LUIGINO CALVI

PENSIERO COMPUTAZIONALE

PROBLEM SOLVING - CODING - ROBOTICA EDUCATIVA

2022

Una *grande* scoperta risolve un grande problema ma c'è un granello di scoperta nella soluzione di ogni problema.
Il tuo problema può essere modesto; ma se esso sfida la tua curiosità e mette in gioco le tue facoltà inventive, e lo risolvi con i tuoi mezzi, puoi sperimentare la tensione e godere del trionfo della scoperta.
Questa esperienza ad una età suscettibile può creare un gusto per il lavoro mentale e lasciare un'impronta nella mente e un carattere per tutta la vita.

G. Polya, *How to solve it*

Indice

Parte I

PROBLEM SOLVING

1

PROBLEMI

Sentiamo in noi l'eterno richiamo: ecco il problema, cercane la soluzione.
D. Hilbert

Il matematico David Hilbert, al Congresso Internazionale di Matematica tenutosi a Parigi nell'agosto del 1900, si espresse con le parole riportate nella citazione sopra per sottolineare che nell'uomo è connaturata la propensione a risolvere le questioni ed i problemi che gli si presentano. In quell'occasione presentò una lista di 23 problemi all'epoca non ancora risolti, proponendoli come sfida per i matematici e con la speranza che potessero essere risolti nel secolo che stava per iniziare. I problemi di quella lista hanno resistito per molti decenni ed alcuni attualmente non sono stati ancora risolti.

In questo capitolo viene definito il contesto di quanto seguirà nei successivi; la catena degli argomenti può essere sintetizzata come segue: si formulano i problemi che opportunamente generalizzati costituiscono delle classi di problemi che vengono risolti dal solutore che descrive un algoritmo che viene eseguito da un esecutore che, in base ai dati forniti, genera il risultato di un'istanza di problema della classe.

1.1 Problemi

Per quanto sarà detto nel seguito, fra le tante possibili, adotteremo la seguente definizione di problema:

> *Un problema è un quesito a cui si vuole dare risposta o un obiettivo che si intende perseguire.*

Da questa definizione risulta implicitamente che ogni problema sottintende uno sforzo volontario per il raggiungimento del risultato finale; pertanto non considereremo come problemi quelli che ammettono palesemente la risposta o quelli ai quali non siamo interessati. Coerentemente con queste ipotesi, un problema è tale relativamente ad un dato soggetto che lo considera. Ad esempio, possiamo considerare come problema il seguente: *Determinare le intensità delle correnti in un dato circuito.* Al contrario, per noi non sarà un problema il seguente (essendo evidente la risposta): *Decidere se un dato numero naturale è pari o dispari.* Analogamente, possiamo non considerare come problema il seguente (in quanto non ci interessa): *Determinare un numero naturale primo composto da cento cifre decimali.* Nonostante l'apparente stravaganza, questioni di questo ultimo tipo rappresentano degli effettivi problemi pratici che si incontrano in applicazioni di codifica/decodifica di informazioni cifrate nella cosiddetta *crittografia a chiave pubblica.*

Nel seguito considereremo solo problemi che sono definibili in modo preciso, univoco e non ambiguo, e che si prestano, quindi, ad essere risolti in modo automatico. Con questa ipotesi si potrà ambientare i problemi in appositi contesti formali, all'interno dei quali la soluzione può essere descritta in modo rigoroso. Rimane così definita una prima classificazione dei problemi, come descritto nella figura 1.1.

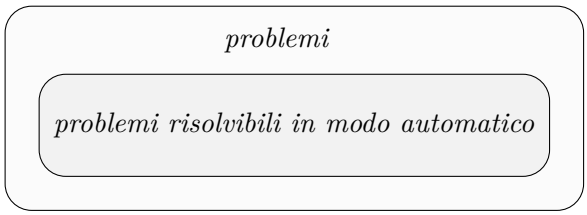


Figura 1.1: Una classificazione dei problemi.

Esempio 1.1.1 - Storicamente i primi problemi si sono presentati quando l'uomo ha iniziato ad operare con i numeri e con le figure elementari del piano. A seguire è riportata una lista di classici problemi che sono stati considerati ed affrontati dai greci più di 2000 anni fa.

1. Stabilire se un numero naturale è primo.
2. Scomporre un numero naturale in fattori primi.
3. Determinare il massimo comune divisore fra due numeri naturali.
4. Determinare la lunghezza della diagonale di un quadrato di lato unitario.
5. Disegnare un poligono regolare di 17 lati con il solo uso di riga e compasso.

1.2 Una lista di problemi

In questo paragrafo viene presentata una variegata gamma di problemi, con l'obiettivo di incuriosire e stimolare a seguire tutto quello che verrà esposto più avanti, quando questi problemi saranno ripresi in considerazione, analizzati e, per quelli risolvibili, ne verrà presentata una soluzione. Si tratta di problemi che riguardano argomenti molto elementari che non richiedono conoscenze avanzate; la loro formulazione è molto semplice ma la loro risoluzione non è mai immediata e banale e risulta interessante in quanto tocca alcuni importanti aspetti delle tecniche di risoluzione algoritmica dei problemi.

Il problema delle 8 regine

Il problema delle 8 regine consiste nel determinare le configurazioni di 8 regine disposte su un'usuale scacchiera in modo tale che nessuna regina sia attaccata da qualche altra; in altri termini, le 8 regine devono trovarsi su righe, su colonne e su diagonali distinte. Nella figura 1.2 è riportata una *non-soluzione* in quanto sono presenti solo 7 regine e non è possibile disporne un'altra senza entrare in conflitto con quelle già presenti.

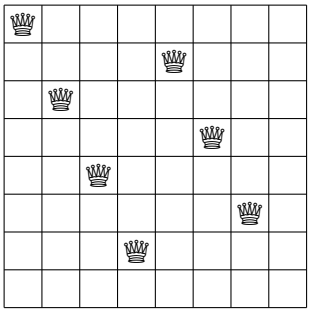


Figura 1.2: Una *non-soluzione* del problema delle 8 regine.

Questo problema fu studiato da Carl Friedrich Gauss (1777-1855) che, nel 1850, riuscì a determinare inizialmente 76 configurazioni e successivamente, con altri tentativi, riuscì a completare il lavoro determinando tutte le 92 possibili configurazioni. Questa soluzione in più tempi fa intuire che all'epoca Gauss non disponesse di un procedimento sistematico di ricerca che conosciamo noi oggi. Nel 1874 J. W. Glaisher propose una generalizzazione di questo problema, considerando una generica scacchiera quadrata di lato n sulla quale disporre n regine; egli giunse ad ipotizzare che su una tale scacchiera fosse sempre possibile disporre n regine. Tale ipotesi venne dimostrata solo nel 1969 [1].

Problemi di questo tipo sono diventati di particolare interesse in quanto, pur non ammettendo delle soluzioni analitiche, si prestano ad essere risolti efficacemente mediante l'uso degli elaboratori. Il problema delle 8 regine è

[1] E. J. Hoffman, J. C. Loessi, R. C. Moore, *Construction for the solution of the n-queens problem*, Mathematical Magazine 42 (1969), 66-72.

divenuto un classico della programmazione in quanto coinvolge tipici ed interessanti aspetti dell'informatica e della programmazione, quali la ricorsività e le strategie di backtracking.

Il problema della torre di Hanoi

Il *problema della torre di Hanoi* è un gioco ideato nel 1883 dal matematico francese Edouard Lucas. Il gioco prende spunto dalla leggenda secondo la quale nel tempio di Benares, in India, i monaci avevano il compito di spostare una piramide di 64 dischi in oro, disposti uno sopra l'altro, dal più grande posto in basso al più piccolo posto in alto, da un piolo di diamante ad un altro, spostando un solo disco alla volta, utilizzando un terzo piolo come appoggio, con il vincolo che ogni disco debba poggiare sopra uno di dimensioni più grandi. Secondo la leggenda il mondo avrà fine quando i monaci avranno finito il proprio lavoro. Sintetizzando quanto tramandato dalla leggenda, il problema è generalmente noto nella seguente formulazione:

> *Sono dati tre pioli e n dischi di diversi diametri. Inizialmente tutti gli n dischi sono impilati su un piolo in modo tale che nessun disco sia disposto sopra di un disco più piccolo. Il problema consiste nello spostare la pila degli n dischi da un piolo ad un altro, muovendo un solo disco alla volta, potendo utilizzare il terzo piolo come supporto ausiliario e rispettando il vincolo che nessun disco sia disposto sopra uno più piccolo.*

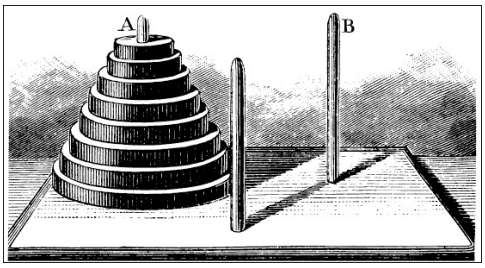


Figura 1.3: Problema della torre di Hanoi con 8 dischi, come proposto nel testo originale di Edouard Lucas.

Nonostante possa sembrare un semplice gioco, il problema suggerisce alcune interessanti questioni tipiche dell'informatica e della programmazione:

- È possibile risolvere il problema per qualsiasi numero n di dischi?
- Determinare il numero di spostamenti necessari per spostare n dischi.
- Determinare la sequenza di spostamenti necessari per spostare n dischi.
- Determinare e descrivere un procedimento per spostare gli n dischi.

Il problema dei ponti di Königsberg

Königsberg è una città della Prussia orientale attualmente situata in territorio russo e nota come Kaliningrad. È attraversata dal fiume Pregel al cui interno sorge un'isola oltre la quale il fiume si spezza nuovamente in due rami. I suoi quartieri sono collegati da sette ponti. Una vecchia mappa della città è riportata nella figura 1.4.

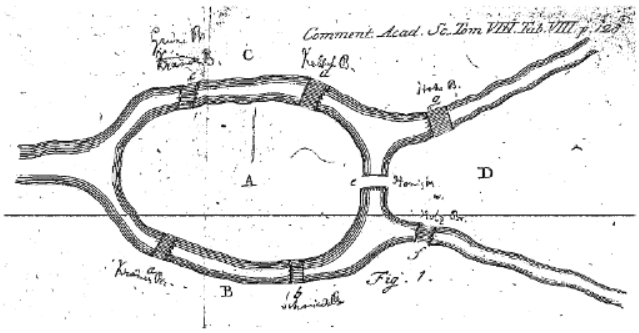


Figura 1.4: Mappa della città di Königsberg.

Agli inizi del XVIII secolo gli abitanti di Königsberg si chiedevano se fosse possibile fare una passeggiata attraversando una sola volta i sette ponti della cittadina e ritornare al punto di partenza. Il problema divenne famoso come *problema dei ponti di Königsberg*, quando il grande matematico svizzero Leonhard Euler (1707-1783), all'inizio di un suo celebre scritto pubblicato nel 1736, lo espose nei seguenti termini:

> *È possibile uscire di casa e farvi ritorno dopo avere percorso tutti e sette i ponti una sola volta?*

Il problema venne approfondito e risolto dallo stesso Euler e le sue disquisizioni sull'argomento diedero avvio alla teoria dei grafi, uno degli ambiti più fecondi della matematica moderna.

Nei suoi scritti Euler affermò che un possibile approccio al problema consiste nell'adottare un procedimento di *forza-bruta* che consiste nell'elencare tutti i possibili percorsi, determinando così i cammini che soddisfano al problema oppure constatando che non esiste alcun cammino soddisfacente. Lo stesso Euler scartò subito questo approccio per vari motivi: il numero di possibili percorsi è enorme e difficilmente trattabile; inoltre, tale approccio crea delle difficoltà indotte che non sono strettamente connesse con la natura del problema; altro aspetto importante è che sarebbe desiderabile un procedimento generale, non adatto solamente al caso specifico in questione, ma valido per una generica configurazioni di ponti.

Il problema dei 4 colori

Il *problema dei quattro colori* può essere formulato nei seguenti termini:

> *È possibile, mediante al più quattro colori, colorare una generica carta geografica piana in modo che regioni contigue abbiano colori differenti?*

In questo problema si ipotizza che ogni regione sia connessa, ossia sia costituita da un unico pezzo. Si pensa che questo problema fosse noto a Möbius già nel 1840. Si è comunque soliti far risalire l'origine di questo problema all'ottobre 1852 quando Francis Guthrie, giovane matematico studente di De Morgan, riuscendo a colorare una cartina delle contee dell'Inghilterra con soli 4 colori, ipotizzò che 4 colori fossero sempre sufficienti per colorare qualsiasi cartina, imponendo l'ovvio vincolo che due regioni contigue abbiano colori diversi. Talvolta, dato il successo della colorazione di tutte le mappe considerate, ci si riferisce a questo quesito con il termine di *Teorema dei* 4 *colori*. De Morgan propose la congettura alla London Mathematical Society, invitando a cercarne una dimostrazione o una refutazione. È davvero sorprendente che un tale problema, all'apparenza così semplice, sia rimasto irrisolto per più di un secolo, nonostante, da subito, moltissimi matematici si siano impegnati nella sua soluzione ed abbiano dedicato anni di lavoro nel tentativo di dimostrare il teorema. Nel 1879 Alfred Bray Kempe, avvocato londinese che studiò matematica a Cambridge, pubblicò una dimostrazione della congettura. Tale dimostrazione venne ritenuta valida per undici anni, fino al 1890 quando Percy John Heawood vi trovò un errore. Nonostante l'errore insito nella sua dimostrazione, Kempe ebbe il merito di introdurre le cosidette *catene di Kempe* che vennero utilizzate un secolo dopo per la dimostrazione definitiva del teorema. Tale risultato conclusivo venne raggiunto nel 1977 da Kenneth Appel e Wolfang Haken, due matematici dell'Università dell'Illinois. La loro dimostrazione si basa sulla riduzione del numero infinito delle mappe possibili a soli 1476 configurazioni per le quali la validità del teorema venne verificata caso per caso, usando un complesso algoritmo informatico la cui esecuzione richiese 1200 ore di elaborazione su diversi computer ad alta velocità. La dimostrazione di Appel ed Haken richiese oltre 500 pagine per essere trascritta, rimanendo comunque impossibile una verifica diretta. Ciò alimentò molte polemiche e molti matematici rifiutarono di accettarla come *dimostrazione* [2]. Successivi miglioramenti della dimostrazione hanno ridotto il numero di configurazioni a 633. Sono state successivamente proposte nuove dimostrazioni alternative, basate sull'uso della teoria dei gruppi, un'importante branca dell'algebra astratta.

Un divertente anneddoto riguardante questo problema risale al 1975, quando Martin Gardner, un famoso esperto e divulgatore di giochi matematici, pubblicò un articolo su un numero di una rivista uscita il 1 aprile, riportando

[2]Vennero all'epoca sollevate profonde discussioni sul cosa sia effettivamente una *dimostrazione*; ma questa questione si insinua nel campo dei fondamenti della matematica e della logica, e non tentiamo degli approfondimenti qui.

una mappa per colorare la quale non sarebbero stati sufficienti 4 colori. Ad alcuni lettori sorse il sospetto, visto anche il giorno dell'uscita dell'articolo, che si trattasse di uno scherzo, come in effetti era. La mappa dello scherzo è riportata nella figura 1.5.

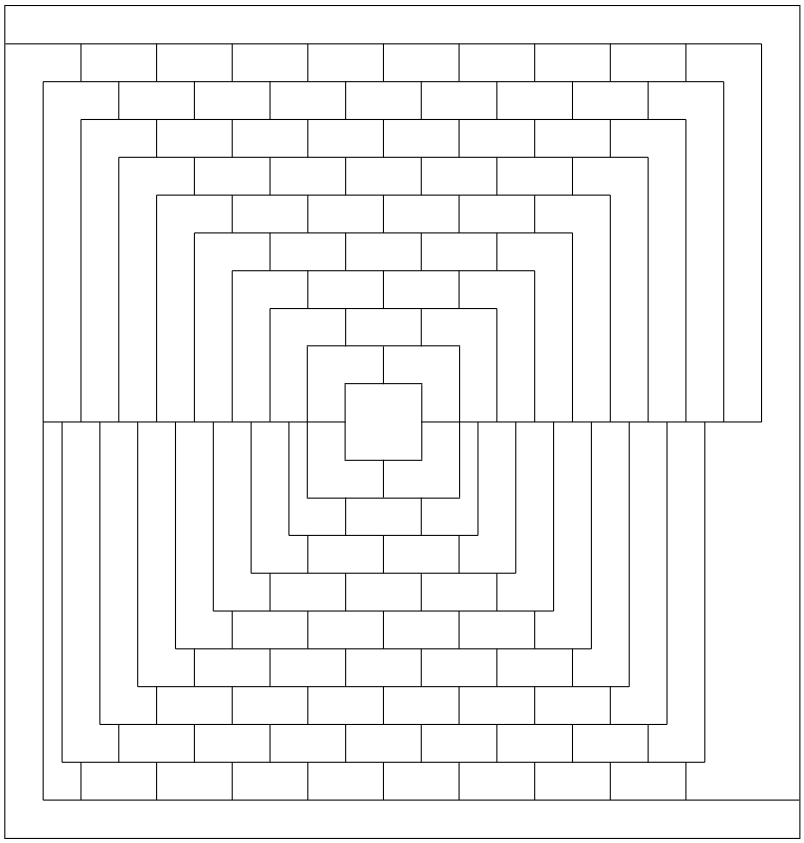


Figura 1.5: Mappa dello scherzo di Martin Gardner.

Il giro del commesso viaggiatore

Il *problema del commesso viaggiatore* fu considerato per la prima volta agli inizi degli anni '30 del secolo scorso, anche se, sotto altra veste, comparve all'interno della teoria dei grafi già nel diciannovesimo secolo. Nella sua tradizionale formulazione il problema si enuncia come segue:

> *Un commesso viaggiatore deve visitare un dato numero di città. Ogni città è collegata a tutte le altre da una strada di cui si conosce la lunghezza. Determinare il percorso più breve che passa per ogni città una sola volta e ritorna alla città di partenza.*

Il problema richiede la formulazione di un procedimento risolutivo generale che non dipenda da una particolare disposizione delle città, né dal numero delle città. Una istanza di questo problema corrispondente a 532 punti è presentata nella figura 1.6.

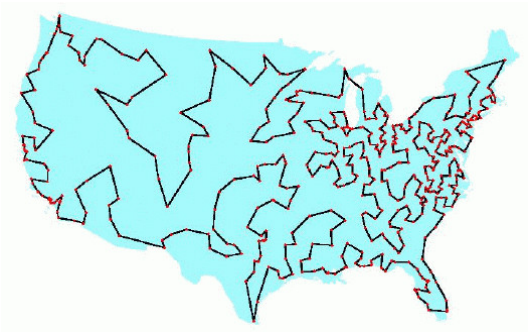


Figura 1.6: Il cammino ottimale sulle 532 centraline della società AT&T negli USA, calcolato nel 1987 da Padberg e Rinaldi.

Al di là della sua formulazione in termini di città da visitare, il problema ha un'importanza pratica in vari settori dell'industria, ad esempio nella stampa dei circuiti elettronici. Un ovvio metodo di soluzione consiste nel determinare tutti i percorsi possibili e poi determinarne il più breve fra questi. La scarsa efficacia di questo approccio discende dal fatto che il numero di possibili percorsi che passano attraverso n città è $(n-1)!/2$ e questo è un numero al di fuori della portata di calcolo anche dei più potenti elaboratori attuali e futuri, anche per modesti valori di n.

Il problema delle equazioni

Moltissimi problemi, di diversa natura, opportunamente trasformati, si riconducono alla soluzione di un'equazione algebrica o ad un sistema di equazioni algebriche. Anche per questo motivo, uno dei problemi più importanti dell'Algebra riguarda la *risolubilità delle equazioni polinomiali di grado n in una sola incognita, mediante delle formule risolutive* costituite da un'espressione contenente solo le quattro operazioni elementari e l'estrazione di radice. Ci limiteremo qui a considerare il caso particolare delle equazioni polinomiali di grado n, per le quali il problema può essere formulato come segue:

> *Determinare una formula risolutiva per risolvere un'equazione polinomiale di grado n, in una sola incognita x:*
>
> $$a_0\,x^n + a_1\,x^{n-1} + \cdots + a_{n-1}\,x + a_n = 0$$

Le equazioni di primo grado ($a\,x+b=0$) e quelle di secondo grado ($a\,x^2+b\,x+c=0$) sono state risolte sin dall'antichità, individuando delle formule che forniscono direttamente le radici. Le equazioni di terzo grado furono risolte solo nei

primi anni del Cinquecento, all'epoca del Rinascimento, ad opera dell'italiano Scipione Dal Ferro (1465 circa - 1526), professore di matematica a Bologna; egli non pubblicò la sua scoperta ma la comunicò ad uno dei suoi allievi. In quegli stessi anni anche il matematico Nicolò Tartaglia (1500 circa - 1557), forse senza conoscere le scoperte di Dal Ferro, scoprì la formula e la comunicò a Gerolamo Cardano (1501-1576) il quale, senza esserne l'ideatore, la pubblicò nella sua opera *Ars Magna*. Pochi anni dopo il matematico italiano Ludovico Ferrari (1522-1565) risolse anche l'equazione generica di quarto grado, trovando un modo per trasformare una generica equazione di quarto grado ad una di terzo grado e potersi quindi avvalere della formula per l'equazione di terzo grado. Al di là della diatriba su chi debba essere considerato il primo scopritore della formula risolutiva per le equazioni di terzo grado, gli splendidi risultati ottenuti in quegli anni stimolarono molti lavori sulla ricerca di formule risolutive per equazioni di grado superiore al quarto. Al noto algebrista Ehrenfried Walter von Tschirnhaus (1651-1708) sembrò persino di aver scoperto un metodo generale per risolvere una generica equazione polinomiale di grado n, riconducendosi ad equazioni di grado inferiore. Ma era solo un'illusione: già per ricondurre un'equazione di quinto grado ad una di quarto grado si richiedeva di risolvere un'equazione ausiliaria di sesto grado, che non si era in grado di risolvere. Nel 1824 il giovane matematico norvegese Niels Henrik Abel (1802-1829) fornì una dimostrazione che non esiste alcuna formula risolvente una generica equazione di grado superiore al quarto. In seguito il matematico francese Evariste Galois (1811-1832), pur nella sua breve vita, basandosi su profondi concetti algebrici, fornì dei criteri per ridurre un'equazione polinomiale al prodotto di polinomi di grado inferiore fornendo così, indirettamente, dei metodi per la sua soluzione.

Il decimo problema di Hilbert

Il *decimo problema di Hilbert* rappresenta uno dei più famosi problemi della matematica. Tale nome deriva dal fatto che questo problema apparve come decimo in una lista di 23 problemi proposti da David Hilbert al secondo congresso dei matematici tenutosi a Parigi nell'agosto dell'anno 1900. Alcuni di questi problemi hanno resistito ai tentativi di molti matematici e risultano a tutt'oggi ancora insoluti. Famoso è il decimo di quella lista. Esso riguarda le equazioni polinomiali a coefficienti interi, ossia equazioni a più incognite legate tra loro da un'espressione polinomiale; ad esempio $3xy - 4y^2 + y = 0$. Il problema proposto da Hilbert può essere formulato nei seguenti termini:

> *Data un'equazione polinomiale a coefficienti interi con un qualunque numero di incognite, descrivere un procedimento per decidere se essa ammetta radici intere.*

Il problema non richiede un procedimento per trovare le soluzioni ma solamente per stabilire se l'equazione ne possegga. Nella formulazione del problema è essenziale il fatto che si cerchino solo radici intere. Un'equazione di questo tipo viene detta *diofantea*, in onore del matematico Diofanto di Alessandria che si occupò di questi argomenti nel terzo secolo d.C.. Per le equazioni di primo grado, ad esempio $3x - 7y + 2z - 14 = 0$, l'esistenza di soluzioni può essere determinata mediante un procedimento ideato ancora da Euclide. Per

le equazioni di secondo grado a due incognite, ad esempio $7x^2 - 4y^2 + xy - 13 = 0$, è possibile stabilire se esistono soluzioni basandosi su una teoria e su dei procedimenti sviluppati all'inizio del XIX secolo dal grande matematico Carl Friedrich Gauss. Per le equazioni di grado superiore al secondo sono stati escogitati alcuni metodi applicabili solo ad alcuni casi particolari.

Nel 1970 il giovane matematico russo Yuri Matijasevic poneva fine alla questione dimostrando che non esiste alcun procedimento generale per stabilire se un'equazione diofantea ammetta radici.

Una famiglia molto nota di equazioni diofantee ha la forma

$$x^n + y^n = z^n, \quad n = 2, 3, 4, \ldots$$

Per $n = 2$ l'equazione risulta soddisfatta dalle lunghezze dei lati di ogni triangolo rettangolo e corrisponde al teorema di Pitagora; ad esempio l'equazione è soddisfatta per $x = 3, y = 4, z = 5$. Per $n \geq 3$ l'equazione è nota come *equazione di Fermat*. Il matematico francese Pierre de Fermat nel diciasettesimo secolo sul margine della sua copia del libro di Diofanto scrisse di aver trovato una dimostrazione veramente mirabile che l'equazione non è soddisfatta per alcuna terna x, y, z) e $n \geq 3$, ma che era troppo lunga per poter essere riportata su quel margine. La dimostrazione di Fermat non venne mai trovata (ammesso che l'avesse effettivamente trovata) e da allora il problema divenne famoso con il nome di l'*ultimo teorema di Fermat*. Nonostante l'appellativo di teorema, tale risultato è stato dimostrato solamente nel 1993 da Andrew Wiles: a contrasto con l'estrema semplicità della formulazione della questione, la dimostrazione è lunga 200 pagine ed ha richiesto nove anni di lavoro.

1.3 Le componenti di un problema

Ogni problema è caratterizzato da tre elementi fondamentali:

1. i *dati*: sono le informazioni iniziali che vengono specificate nella formulazione del problema in base alle quali ricavare i risultati
2. i *risultati*: è quanto si deve determinare in base ai dati iniziali
3. le *condizioni*: specificano come i risultati sono legati ai dati e forniscono un criterio per stabilire la correttezza dei risultati ottenuti

Qui i termini *dati* e *risultati* sono da intendersi in senso lato: possono essere un numero, una sequenza di numeri, la risposta ad un quesito, una figura, un disegno o qualcos'altro di più generale ancora. Dati e risultati vengono spesso indicati anche con i termini *input* ed *output*; nel caso di un problema di tipo matematico spesso i risultati vengono spesso qualificati come *incognite* e vengono tradizionalmente denotate con x, x_1, x_2 e così via.

Esempio 1.3.1 - Consideriamo il problema della risoluzione di un'equazione di secondo grado $ax^2 + bx + c = 0$. In questo caso i dati sono rappresentati dai coefficienti a, b, c dell'equazione, mentre i risultati sono le due radici x_1 e x_2 dell'equazione; la condizione che lega i risultati ai dati è fornita dal fatto che le radici devono soddisfare l'equazione.

1.4 La risoluzione dei problemi

Risolvere un problema significa ricercare e descrivere con un qualche formalismo (*linguaggio*) in modo preciso e non ambiguo un procedimento (*algoritmo*) che, eseguito, consenta, a partire da delle informazioni iniziali (*dati*), di ottenere delle informazioni finali (*risultati*) soddisfacenti ad un *criterio di verifica*. Questa attività viene solitamente detta *problem solving*.

Il *solutore* è il soggetto (in genere l'uomo) che descrive le azioni da eseguire per risolvere un problema. L'*esecutore* è l'entità (in genere una macchina) che esegue le azioni descritte dal solutore. La *soluzione* di un problema è la descrizione delle azioni (*istruzioni*), da parte del solutore e rivolte all'esecutore, che, eseguite, forniscono i risultati del problema. In modo grafico le definizioni appena date possono essere sintetizzate mediante lo schema riportato nella figura 1.7.

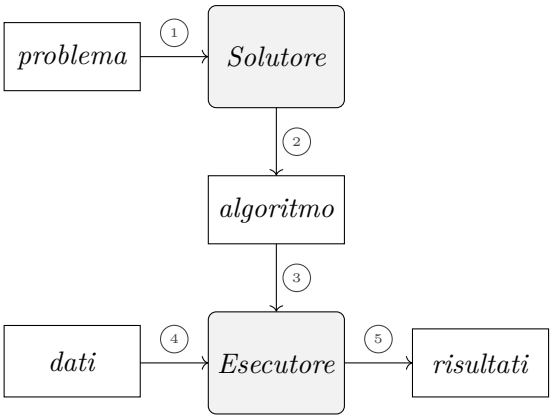


Figura 1.7: Schema del processo risolutivo di un problema: il problema viene analizzato dal solutore che descrive l'algoritmo il quale viene eseguito dall'esecutore che acquisisce i dati in ingresso e produce i risultati in uscita.

Riassumendo le varie definizioni date sopra si può descrivere lo scenario di riferimento dove vengono risolti i problemi come segue:

> *Il solutore descrive, mediante un linguaggio,*
> *la soluzione del problema ad un esecutore*
> *che è capace di eseguire un insieme di operazioni elementari*
> *su un insieme di oggetti elementari.*

Osservazione. La ricerca e la descrizione della soluzione di un problema è un'attività del solutore (e non dell'esecutore, il quale *esegue* il procedimento risolutivo ma non lo *costruisce*). Questa separazione dei ruoli e delle responsabilità tra solutore ed esecutore sarà il filo conduttore di molti delle argomentazioni che seguiranno.

Esempio 1.4.1 - Un procedimento risolutivo per il problema della risoluzione di un'equazione di secondo grado $ax^2 + bx + c = 0$ è fornito dalla ben nota formula risolutiva delle equazioni di secondo grado:

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Con riferimento a questo problema, il solutore è colui che ha ideato la formula risolutiva delle equazioni di secondo grado, mentre chi la applica per trovare, con carta e penna, le radici di una particolare equazione assume il ruolo di esecutore. Nel caso venga utilizzato un programma su calcolatore per trovare le radici di un'equazione, l'esecutore è rappresentato dal calcolatore (e chi utilizza il programma assume il ruolo di semplice utente). Un criterio di verifica consiste nella sostituzione diretta dei valori trovati delle radici e nella verifica dell'identità.

Osservazione. L'impostazione basata sui termini chiave *problema*, *solutore*, *soluzione*, *esecutore*, *risultato* è tipica del cosidetto paradigma *imperativo*, secondo il quale il solutore specifica all'esecutore le direttive di *come* deve agire per ottenere il risultato finale. Ad un più alto livello di astrazione, un'impostazione alternativa, che caratterizza il cosidetto paradigma *dichiarativo*, consiste nel descrivere all'esecutore da parte del solutore, con informazioni e vincoli stringenti, in *cosa* consiste il risultato, delegando all'esecutore il compito di trovare i passi da eseguire per determinarlo.

1.5 Problemi ben formulati

Ai fini della loro risolvibilità i problemi devono soddisfare ai vincoli precisati nella seguente definizione. Si afferma che un problema è *ben formulato* o *formalmente definito* se soddisfa alle seguenti specifiche:

1. Il problema è formulato in una forma comprensibile dal solutore
2. I dati sui quali si deve cercare la soluzione sono sufficienti e coerenti
3. È univoco il criterio che definisce i risultati

Esempio 1.5.1 - Consideriamo il seguente problema:

> *Determinare il barimetro di un poliangolo.*

Questo è un chiaro esempio di problema mal formulato in quanto nella sua formulazione compaiono dei termini che non hanno alcun significato per il solutore.

Esempio 1.5.2 - Un altro problema mal formulato è il seguente:

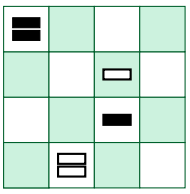
> *Determinare le lunghezze delle diagonali di un trapezio sapendo che le basi maggiore e minore misurano rispettivamente* 18 *e* 12 *unità e che i lati obliqui misurano* 5 *e* 7 *unità.*

In questo caso i dati sono insufficienti. L'aggiunta dell'attributo *rettangolo* al trapezio renderebbe ben formulato il problema. Come conseguenza c'è da

notare che anche un'ipotesi o una condizione può costituire un *dato* e quindi può determinare la ben formulazione di un problema.

Esempio 1.5.3 - Consideriamo il seguente problema:

> *Risolvere la seguente situazione di gioco:*



Questo è un altro esempio di problema mal formulato in quanto non è precisato qual è il gioco, quali sono le possibili mosse e qual è il criterio per determinare se una data posizione finale è vincente. Sono invece ben formulati gli usuali problemi di scacchi che si trovano sui giornali di enigmistica, in quanto le regole del gioco sono ben definite ed è univoco il criterio per valutare se una data posizione finale sia vincente, in base alle regole del gioco degli scacchi, anche se tali ipotesi sono assunte implicitamente e non direttamente riportate nella formulazione del problema.

1.6 Formalizzazione dei problemi

Il processo di *formalizzazione* o *modellizzazione* di un problema consiste nel tradurre un problema reale in una sua formulazione alternativa, semplificata ma equivalente per quanto riguarda la tematica di interesse, in modo tale che esso possa essere risolto automaticamente.

Esempio 1.6.1 - Consideriamo il seguente problema:

> *Si ha un bersaglio delimitato da* 3 *cerchi concentrici. Il cerchio più interno vale* 19 *punti, mentre le due corone esterne valgono rispettivamente* 13 *e* 5 *punti. Il raggio del cerchio interno e lo spessore delle corone sono di* 7 *cm. Decidere se, ed eventualmente come, è possibile totalizzare* 100 *punti, in più tiri.*

Filtrando i dettagli superflui (raggio e spessore delle corone), tale problema può essere formalizzato mediante un'*equazione diofantea*, ossia mediante un'equazione avente coefficienti interi e le cui radici devono essere numeri interi; l'equazione è la seguente:

> *Risolvere l'equazione diofantea* $19x + 13y + 5z = 100$.

In questo caso, data la particolare formulazione del problema, si deve imporre il vincolo che le incognite x, y, z debbano essere numeri naturali. Risolvendo questa equazione si trova la soluzione $(x = 3, y = 1, z = 6)$. Questo risultato ammette la seguente *interpretazione*: centrando 3 volte il cerchio interno, 1

volta la corona intermedia e 6 volte la corona esterna si totalizzano 100 punti. Ci si può chiedere se questa sia l'unica soluzione; la risposta è evidentemente no in quanto si nota la soluzione $(x = 0, y = 0, z = 20)$. Il problema diventa allora più interessante se si impone il vincolo di ridurre al minimo il numero di tiri necessari per totalizzare 100 punti. Ogni formalizzazione e schematizzazione comporta una certa semplificazione, riduzione ed idealizzazione della realtà: nella soluzione sopra non si considera il caso che delle frecce vadano proprio su una linea di separazione fra due zone. Notiamo che se il problema consistesse nel determinare i punti da assegnare alle varie corone del bersaglio, dato il numero di tiri in ciascuna corona ed il totale dei punti da ottenere, il problema potrebbe essere formalizzato mediante un'equazione diofantea simile a quella riportata sopra.

1.7 Dati e risultati

Un passo fondamentale del processo di risoluzione di un problema consiste nell'individuazione dei *dati* e dei *risultati*. Questa fase costituisce l'anello di congiunzione fra il problema e la sua soluzione; in alcuni casi è definita compiutamente nella formulazione del problema; in altri casi può rimanere implicita o latente e richiedere una fase di analisi da parte del solutore. In alcuni casi questa fase può essere offuscata dalla formulazione del problema, in altre può invece essere suggerita dal classico schema *Dati ... determinare ...* .

La descrizione dei dati e dei risultati può essere fatta a diversi livelli di dettaglio che dipendono dal contesto in cui viene formulato il problema e, in definitiva, dalle capacità possedute dall'esecutore al quale verrà rivolto il procedimento risolutivo del problema.

Esempio 1.7.1 - Consideriamo il seguente problema:

> *Dati nel piano un punto ed una retta, determinare il segmento più corto che unisce il punto alla retta.*

Dalla formulazione del problema sembrano evidenziarsi i seguenti dati e risultati:

Dati: retta r, punto P
Risultati: segmento s

Questa analisi risulta imprecisa in base alle seguenti osservazioni: un segmento nel piano è individuato dai suoi estremi; il segmento s ha come estremo il punto P che è noto; pertanto l'unico risultato da determinare è l'estremo Q che sta sulla retta. La corretta individuazione dei dati e risultati è dunque la seguente:

Dati: retta r, punto P
Risultati: punto Q

Questa descrizione dei dati e dei risultati può essere adeguata per un esecutore in grado di operare con i tradizionali strumenti delle costruzioni geometriche come una riga (non graduata) ed un compasso. Se l'esecutore è in grado

di operare con i numeri, si può ambientare il problema in un sistema di assi cartesiani ortogonali, nel qual caso un punto risulta individuato dalla coppia delle sue coordinate ed una retta individuata da un'equazione della forma $a\,x + b\,y + c = 0$; a sua volta un'equazione di questa forma risulta individuata dai tre coefficienti a, b, c. Pertanto, i dati ed i risultati possono essere precisati come segue:

Dati:	coefficienti (a, b, c) dell'equazione cartesiana della retta, coordinate (x, y) del punto
Risultati:	coordinate (x_1, x_2) del punto sulla retta

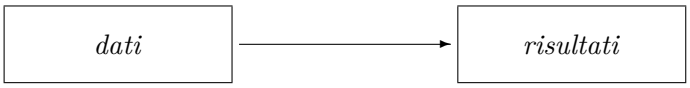
Anche questa formulazione può essere limata, osservando che le coordinate (x_1, x_2) definiscono un punto *sulla retta*; pertanto è sufficiente precisare l'ordinata del punto (e determinare la corrispondente ascissa dall'equazione della retta). Questa formulazione, benché minimale, si espone al caso particolare che la retta sia parallela all'asse y, nel quale caso la conoscenza dell'ascissa non individua più il punto. Per questo motivo ritorna ad essere preferibile la precedente formulazione.

Dopo aver individuato i dati a disposizione ed i risultati da determinare, il passo successivo consiste nella risoluzione del problema. In alcuni casi la soluzione consiste in una semplice formula; in altri casi la soluzione può consistere in un procedimento più articolato, detto *algoritmo*.

1.8 Risolvibilità dei problemi

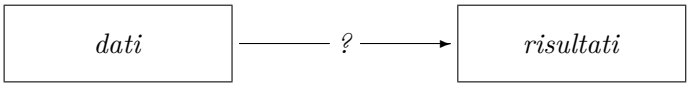
Un procedimento risolutivo di un problema può essere efficacemente raffigurato come un cammino che congiunge i dati iniziali ai risultati finali. Per alcuni problemi non si conosce oppure non esiste alcun cammino risolutivo. A seconda che questo cammino esista o no, si possono classificare i problemi come segue:

1. problemi *risolvibili*: sono quei problemi per i quali esiste un procedimento risolutivo. Questi problemi possono essere descritti dal seguente schema:



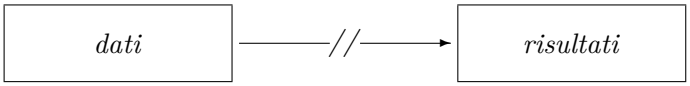
Esempio 1.8.1 - È risolvibile il problema di risolvere un'equazione algebrica polinomiale di secondo grado, della forma $ax^2 + bx + c = 0$.

2. problemi *risolvibili*? : sono quei problemi per i quali non è noto se esista un procedimento risolutivo. Possono essere descritti dal seguente schema:



Esempio 1.8.2 - Un problema di cui non si sa se è risolvibile o meno è fornito dalla congettura che esistano infinite coppie di numeri primi gemelli, ossia numeri naturali che differiscono per due unità (esempi di coppie di numeri primi gemelli: $(3,5)$, $(5,7)$, $(11,13)$, $(17,19)$).

3. problemi *irrisolvibili*: sono quei problemi per i quali è noto che non esiste alcun procedimento risolutivo. Possono essere descritti dal seguente schema:



Esempio 1.8.3 - La soluzione di una generica equazione algebrica polinomiale di grado maggiore di 4 in un'incognita, ad esempio $7x^5-13x^3+x^2+5 = 0$ è un problema irrisolvibile.

Osservazione. Il concetto di *problema risolvibile* è tutt'altro che ovvio e scontato; in realtà sottende due diverse impostazioni filosofiche di fondo: nel XX secolo, quando i problemi considerati erano di tipo prevalentemente matematico, i matematici consideravano *risolvibile* un problema per il quale era stato trovato un procedimento risolvibile o anche quando era stato provato che un tale procedimento *esisteva*; non era necessario, e quasi era una questione marginale, che il procedimento venisse esibito esplicitamente. A partire dal XXI secolo, sotto la sollecitazione del pragmatismo informatico, un problema viene considerato risolvibile se ne è stato descritto un procedimento risolutivo effettivo.

1.9 Tipologie dei problemi

I problemi presentano delle analogie rispetto ad alcune caratteristiche quali la forma del problema, la sua risolvibilità, la tipologia dei dati e dei risultati, l'ambiente di operatività (aritmetico, geometrico, grafico, ...). La variegata gamma dei problemi che si incontrano può essere pertanto classificata in categorie rispetto ad una di queste caratteristiche. Classificare un problema costituisce un primo passo verso la sua comprensione e, quindi, verso la sua soluzione.

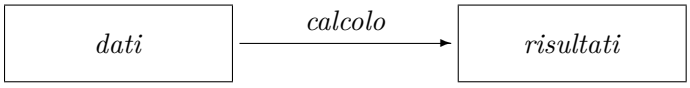
Molto spesso i problemi rappresentano delle analogie con riferimento alle informazioni iniziali che si dispongono ed all'obiettivo finale che si intende raggiungere. Come si vedrà più avanti, queste tre classi di problemi verranno risolte mediante delle corrispondenti classi di procedimenti generalmente definiti *funzioni*, *predicati* e *procedure*. In base a questo criterio i problemi che più frequentemente si incontrano nella programmazione possono essere partizionati nelle classi di seguito descritte.

Problemi computazionali

Una delle categorie più tradizionaali di problemi ammette delle soluzioni che richiedono di eseguire calcoli su dei dati numerici ottenendo dei risultati nu-

merici. Questi problemi vengono classificati come *problemi computazionali* e vengono formulati secondo il seguente schema:

A partire da dei dati numerici iniziali noti,
mediante dei calcoli,
determinare dei risultati numerici.

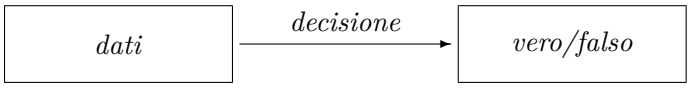


Esempio 1.9.1 - Determinare il *fattoriale* di un numero naturale n, ossia il valore, indicato con $n!$, uguale al prodotto $1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n$ dei primi n numeri naturali.

Problemi decisionali

Una vasta classe di problemi rientra nell'insieme dei *problemi di decisione* che hanno il caratteristico incipit *Decidere se ...* ed hanno come risultato *vero* o *falso*. Questi problemi spaziano dall'ambito della logica, alla matematica ed includono molte questioni tipiche dell'informatica. Un problema *decisionale* ha la seguente struttura:

Fissate delle ipotesi iniziali,
attraverso calcoli e deduzioni logiche,
stabilire se una data proposizione è vera.

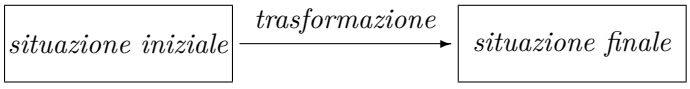


Esempio 1.9.2 - Data una sequenza $(a_1, a_2, \cdots, a_n)$ ordinata, ossia tale che $a_1 \leq a_2 \leq \cdots \leq a_n$, stabilire se nella sequenza è presente un dato elemento x.

Problemi esecutivi

I problemi di tipo *esecutivo* si formulano come segue:

Fissata una situazione iniziale di partenza,
eseguire una sequenza di trasformazioni
in modo da ottenere una situazione finale obiettivo.



Esempio 1.9.3 - Il *problema della torre di Hanoi* consiste nello spostare da un piolo ad un altro, usando come appoggio ausiliario un terzo piolo, una pila di n dischi, di dimensioni decrescenti dal basso verso l'alto. È ammesso spostare un solo disco alla volta, con il vincolo che ogni disco non può essere appoggiato sopra uno più grande.

1.10 Istanze e classi di problemi

Nel suo percorso scolastico uno studente incontra varie tipologie di problemi. Nei primi anni di scuola si imbatte in problemi come il seguente:

> *Moltiplicare* 7413×938.

In una situazione come questa, l'allievo assume il ruolo di esecutore, eseguendo la moltiplicazione in colonna seguendo il procedimento insegnatogli dalla maestra.

In un problema apparentemente simile come il seguente:

> *Calcolare l'area di un rettangolo avente la base e l'altezza di lunghezza pari a* 420 *e* 112 *cm.*

allo studente viene implicitamente richiesto, prima di mettersi nella veste di esecutore dei calcoli, di individuare in veste di solutore la corretta operazione da eseguire.

Un po' più avanti affronta problemi della seguente forma:

> *Calcolare il massimo comune divisore fra* 798 *e* 242.

Anche in questo caso lo studente, sempre in veste di esecutore, applica il procedimento che l'insegnante gli ha precedentemente spiegato; in questa situazione il procedimento si fonda sull'ipotesi che lo studente abbia preventivamente acquisito la capacità di scomporre un numero in fattori primi.

Dopo aver maturato delle ulteriori conoscenze sui numeri e sulle proprietà delle figure del piano, lo studente si trova ad affrontare problemi della forma

> *Determinare il raggio della circonferenza inscritta nel triangolo rettangolo avente le misure dei cateti pari a* 72 *e* 45 *cm.*

In questo caso il risultato viene ottenuto mediante una serie di passaggi eseguendo operazioni ed applicando delle formule e proprietà. In questo caso lo studente assume il doppio ruolo di solutore (individuando il procedimento guida) ed esecutore (eseguendo le operazioni descritte nel procedimento).

Tutti i casi di problemi precedentemente analizzati sono caratterizzati dall'avere come dati dei valori particolari. Si parla in questo caso di *istanze di problemi*. Un netto cambio di contesto avviene considerando dei dati generici, denominati mediante degli identificatori; ad esempio: Determinare il massimo comune divisore fra due numeri naturali m ed n. In questo modo si evidenzia un insieme di problemi aventi la stessa struttura, detto *classe di problemi*, i cui elementi (istanze di problemi) si ottengono per particolari valori assunti dai dati. In situazioni come questa si individuano in modo distinto i ruoli del *solutore* che descrive il procedimento e dell'*esecutore* che esegue il procedimento su una particolare istanziazione dei valori dei dati. Nel seguito, quando si parla di *problema* si intenderà solitamente una *classe di problemi*, che risulta caratterizzata da dei parametri che costituiscono i dati.

Le classi di problemi vengono risolte mediante un procedimento risolutivo generale indipendente dalle particolari istanziazioni dei dati. Ad esempio, un

metodo di soluzione per la ricerca del massimo comune divisore di due numeri naturali m e n deve essere valido per qualsiasi coppia di valori assunti da m e n. Risulta dunque evidente che un metodo risolutivo di un problema deve essere applicabile ad un'intera classe di problemi le cui istanze differiscono soltanto per i particolari valori assunti dai dati iniziali. Con queste premesse, quando nel seguito si userà il termine *soluzione di un problema* ci si riferirà ad un procedimento risolutivo generale, valido per un'intera classe di problemi alla quale appartiene il problema considerato.

Lo schema attraverso il quale si ricava il risultato di una particolare istanza di problema si sviluppa attraverso le seguenti fasi successive:

1. considerazione dell'istanza P del problema
2. individuazione di una classe C alla quale appartiene il problema P
3. ricerca e descrizione di un procedimento risolutivo A per la classe C
4. applicazione del procedimento risolutivo A all'istanza P

Graficamente, queste fasi possono essere evidenziate mediante lo schema riportato in figura 1.8.

Figura 1.8: Schema delle fasi per la soluzione di un'istanza di problema.

1.11 Generalizzazione dei problemi

La generalizzazione di un problema specifico individua una classe di problemi della quale il problema originale costituisce una particolare istanza. Data un'istanza di problema si può ottenere immediatamente una classe di problemi denotando i valori dei dati con dei nomi generici.

Esempio 1.11.1 - L'ultima istanza di problema descritta nel precedente paragrafo può essere generalizzata nella seguente classe:

> *Determinare il raggio della circonferenza inscritta nel triangolo rettangolo avente le misure dei cateti pari a a e b.*

Come è stato sopra suggerito, quando ci si imbatte in un problema particolare (ad esempio, trovare le radici dell'equazione $x^2 + 3x - 5 = 0$) prima

di impostare la soluzione è conveniente, per fare un buon investimento dello sforzo che ci si appresta a compiere, trovare una opportuna generalizzazione del problema. Il punto centrale del procedimento di generalizzazione riguarda le seguenti questioni:

- *Come generalizzare?*
- *Fino a che punto generalizzare?*

Un po' di buon senso (all'inizio) ed un po' di esperienza (quando sarà stata maturata) saranno sufficienti come guida per individuare una buona generalizzazione. La figura 1.9 descrive lo schema di successive generalizzazioni di un'istanza di problema.

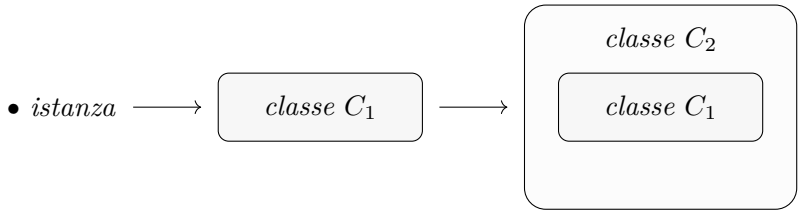


Figura 1.9: Passi di generalizzazione di un'istanza di problema.

Esempio 1.11.2 - Consideriamo la seguente istanza di problema:

> *Determinare la distanza fra i due punti del piano cartesiano* $P(1,2)$, $Q(4,3)$.

Una prima generalizzazione consiste nel prendere dei valori generici per i dati delle coordinate dei punti:

> *Determinare la distanza fra i due punti del piano cartesiano aventi coordinate* (x_1, y_1) *e* (x_2, y_2).

Per individuare delle successive generalizzazioni è sufficiente considerare i termini del problema che si prestano ad essere generalizzati come segue:

$$\begin{aligned} 2 &\longrightarrow n \\ \text{punti} &\longrightarrow \text{elementi} \\ \text{piano} &\longrightarrow \text{insieme } E \text{ generico} \\ \text{distanza} &\longrightarrow \text{distanza } d \text{ generica} \end{aligned}$$

Adottando queste generalizzazioni si ottiene la seguente formulazione molto generale del problema:

> *Determinare il diametro della più piccola sfera contenente un dato insieme* $E = \{a_1, ..., a_n\}$ *di* n *elementi di uno spazio metrico* (E, d).

Per il momento questa formulazione può risultare oscura ma, una volta risolta questa classe di problemi, avremo trovato una soluzione per moltissime sottoclassi, appartenenti a contesti molto diversi.

ESERCIZI

1.1 Individuare i dati ed i risultati di ciascuno dei problemi riportati nel paragrafo 1.2.

1.2 Determinare una configurazione ammissibile di 8 regine.

1.3 Determinare le mosse necessarie per spostare una pila di 3 dischi per il problema della torre di Hanoi.

1.4 Usando solo 4 colori, colorare la mappa riportata nella figura 1.5.

1.5 Spiegare la differenza fra i termini *soluzione* e *risultato*.

1.6 Spiegare il significato dei termini *solutore* ed *esecutore*.

1.7 Spiegare perché non ha senso considerare un procedimento risolutivo senza fare riferimento alle capacità dell'esecutore.

1.8 Spiegare se i problemi *risolvibili?* possono essere qualificati con l'attributo *irrisolvibili?*.

1.9 Fissato un dato problema P, si considerino i seguenti due problemi:

P_1: *Il problema P ammette soluzione ?*

P_2: *Risolvere il problema P.*

Classificare i problemi P_1 e P_2 in base alla loro forma. Stabilire, in funzione della risolvibilità di P, la risolvibilità dei problemi P_1 e P_2.

1.10 Determinare per quali valori di n il seguente problema ammette un unico risultato:

> *Fissati sul piano un insieme di n punti distinti $P_1, \ldots, P_n$, disegnare un poligono di n lati non intrecciati, aventi $P_1, \ldots, P_n$ come vertici.*

1.11 Determinare per quali valori di n il seguente problema è ben formulato:

> *Determinare l'area di un poligono di n lati conoscendone la misura dei lati, in ordine in base alla consecutività dei lati.*

1.12 Stabilire se i seguenti problemi sono ben formulati:

1. Una classe è formata da s studenti. Di questi m sono maschi. Ci sono k studenti che portano gli occhiali. Determinare quante sono le femmine che portano gli occhiali.
2. Determinare il perimetro del rettangolo di cui è nota la lunghezza d della diagonale e l'area a.
3. Determinare le misure dei lati del triangolo di cui è noto il perimetro p e l'area a.
4. Determinarne l'area ed il perimetro del triangolo rettangolo di cui è nota la misura l di un lato ed a dell'ipotenusa.
5. Determinare l'area di un triangolo di cui è noto il perimetro e la lunghezza delle tre altezze.

6. Determinare l'area di un triangolo di cui sono note le misure dei lati.
7. Determinare l'area di un quadrilatero di cui sono note le misure dei lati.
8. Determinare l'area di un rettangolo di cui è noto il perimetro e la lunghezza della diagonale.
9. Determinare l'area di un trapezio di cui sono note le misure delle basi e dei lati obliqui.
10. Date le misure dei lati di due triangoli, stabilire se il primo è più piccolo del secondo.
11. Determinare le lunghezze degli spigoli di un parallelepipedo di cui sono noti il volume e la superficie totale.

1.13 Completare, in più modi, la formulazione del seguente schema di problema in modo da costituire dei problemi ben formulati:

Di un triangolo si conosce la lunghezza di ciascun lato. Determinare ...

1.14 Definire delle ipotesi integrative in modo che il seguente problema risulti ben formulato:

Ordinare una sequenza di triangoli.

1.15 Spiegare perché, senza assumere tacite ipotesi, i seguenti problemi sono mal formulati. Ridefinirli in modo che risultino ben formulati. Con riferimento al problema ben formulato, individuare quali sono i dati ed i risultati e specificare la tipologia del problema.

1. Risolvere l'equazione di secondo grado $ax^2 + bx + c = 0$.
2. Determinare il numero delle cifre di un numero naturale.
3. Determinare la somma delle cifre di un numero naturale.
4. Determinare la somma delle cifre di un numero naturale considerato in base 10.
5. Rovesciare un numero naturale di tre cifre.
6. Determinare i più piccoli dieci numeri naturali che si leggono indifferentemente da sinistra a destra e da destra a sinistra.
7. Date le misure dei lati di un triangolo acutangolo, determinare il più piccolo rettangolo che lo ricopre.
8. Date le misure dei lati di un quadrilatero, decidere se il quadrilatero è convesso.
9. Date le misure dei 6 segmenti congiungenti in tutti i modi possibili i vertici di un quadrilatero, determinare il perimetro del quadrilatero.
10. Fissati sul piano 4 punti distinti e non allineati, disegnare il quadrilatero non intrecciato avente per vertici i 4 prefissati punti.

1.16 È assegnato un insieme $A = \{P_1, P_2, ..., P_n\}$ di n punti del piano. Stabilire se i seguenti due problemi sono ben formulati; in caso affermativo dire in cosa consistono i dati ed i risultati.

1. Determinare l'area del poligono di vertici A.
2. Determinare l'involucro convesso dell'insieme dei punti A, ossia il più piccolo poligono convesso contenente i punti dati.

1.17 Individuare i dati ed i risultati dei seguenti problemi relativi alle equazioni di secondo grado:

1. Determinare le radici di un'equazione di secondo grado.
2. Determinare il numero delle radici reali distinte di un'equazione di secondo grado.
3. Stabilire se un'equazione di secondo grado ammette soluzioni reali.
4. Stabilire se un valore è soluzione di un'equazione di secondo grado.

1.18 Individuare i dati ed i risultati dei seguenti problemi relativi alla codifica dei numeri:

1. Determinare il numero di cifre necessarie per scrivere un numero naturale.
2. Scrivere un numero in una data base.
3. Convertire un numero naturale da una base ad un'altra.

1.19 I seguenti tre problemi sono apparentemente simili ma, ad una attenta analisi, si capisce che sono di differenti complessità:

1. Determinare un numero primo di k cifre decimali.
2. Decidere se un dato numero naturale n è primo.
3. Scomporre in fattori primi un dato numero naturale n.
4. Determinare la somma dei reciproci di tutti i numeri primi.

Assumendo in modo intuitivo il concetto di *complessità di un problema*, intesa come una misura del tempo di esecuzione della miglior soluzione del problema, discutere questi problemi in riferimento alla loro complessità, ordinandoli dal più *semplice* al più *difficile*.

1.20 Descrivere una classe di problemi che includa le seguenti tre classi di problemi sui numeri naturali:

1. Numerare per n da p a q.
2. Determinare i numeri minori di n.
3. Determinare i multipli di un numero contenuti in un intervallo.

1.21 Definire una classe di problemi alla quale appartengano le seguenti due istanze:

1. Valutare $\log_2 9871$
2. Valutare $\log_8 24364$

1.22 Individuare i dati ed i risultati delle seguenti classi di problemi nel contesto del piano della geometria descrittiva e nel contesto del piano cartesiano:

1. Determinare la circonferenza circoscritta ad un triangolo.
2. Determinare la retta passante per due punti.
3. Determinare il punto di intersezione fra due rette.
4. Determinare la circonferenza passante per tre punti.
5. Stabilire se un punto è interno ad una circonferenza.

1.23 Generalizzare le seguenti istanze di problemi, proponendo delle classi di problemi via-via più generali:

1. Determinare l'area di un quadrato di lato 2 cm.
2. Determinare la lunghezza della diagonale di un quadrato di lato 2 cm.
3. Trovare il lato del triangolo equilatero inscritto in un cerchio di raggio 8 cm.
4. Ordinare i tre numeri 5, 4, 6.
5. Nel piano cartesiano, determinare la distanza fra i due punti $P = (1, 2)$, $Q = (3, 4)$.
6. Determinare l'area di un quadrato inscritto in una circonferenza di lato 4 unità.
7. Disegnare un quadrato rosso di lato 1 cm.
8. Determinare la distanza del punto di coordinate $(1, 2)$ dall'origine degli assi cartesiani.
9. Decidere se ci sono domeniche comprese nel periodo dal 30 ottobre 2018 al 2 novembre 2018.
10. Risolvere l'equazione $2x - 8 = 0$ nell'insieme dei numeri naturali. [3]

Per ciascuna classe di problemi che si è individuata, si precisino quali sono i *dati* e quali i *risultati*.

[3] Suggerimento: si possono ottenere delle generalizzazioni sia ampliando l'insieme dove si cercano le soluzioni, sia generalizzando la forma dell'equazione.

2

MACCHINE

L'informatica non riguarda i computer più di quanto l'astronomia riguardi i telescopi.

Edsger Wybe Dijkstra
(*informatico olandese*) [1930-2002]

Il fatto che la ricerca scientifica sia riuscita a riconoscere in modo preciso i limiti delle sue possibilità, ci sembra una prestazione dello spirito umano, più grande della tecnicizzazione del nostro mondo, tanto spesso ammirata.

H. Meschkowsky, *Mutamenti nel pensiero matematico*

In ogni contesto operativo la risoluzione di un problema deve avvalersi di un'entità in grado di agire ed *eseguire* delle azioni definite in base ad un prescritto procedimento risolutivo. A questo scopo l'uomo ha costruito degli esecutori automatici: robot meccanici, elaboratori elettronici, calcolatori biologici ed altre tipologie di marchingegni. A prescindere dalle particolari tecnologie realizzative, tali esecutori vengono genericamente denominati *macchine*.

2.1 Uomini e macchine

Un tratto caratterizzante l'evoluzione dell'uomo è rappresentato dalla costruzione di strumenti sempre più sofisticati e potenti che potessero aiutarlo nello svolgimento delle sue attività. Questa tendenza si è sviluppata fino ad invadere gli aspetti più speculativi del pensiero umano. Addirittura, specialmente nell'alveo della cultura occidentale, si è assistito al tentativo, parzialmente conseguito, di costruire sistemi artificiali in grado di riprodurre comportamenti ed attività mentali che sembravano prerogativa precipua dell'uomo. In modo ampio e generico questi strumenti vengono detti *macchine*; in ambito scientifico, nel caso che vengano adibiti per qualche forma di elaborazione di dati, vengono detti *computer*; in ambito industriale vengono spesso denominati *robot*.

Il computer ha avuto negli ultimi decenni un fortissimo impatto sull'evoluzione delle scienze, dell'ingegneria ed in alcuni settori della matematica, aprendo varchi di indagine su aspetti altrimenti impenetrabili, assumendo un ruolo simile al microscopio per le scienze biologiche. I *frattali*, oggetti matematici definiti mediante funzioni, hanno potuto essere studiati solo grazie al computer che ha reso possibile una mole di calcoli preclusi alla sola operatività dell'uomo; vengono solitamente rappresentati in forma grafica trasformando i risultati dei calcoli in immagini di alta suggestione; furono visti per la prima volta nel 1980: penso che, a quell'epoca, quei primi esploratori del mondo frattale abbiano provato un'emozione simile a quella degli speleologi che scoprono una grotta carsica ricca di meravigliose stallattiti e stallagmiti o a quella degli archeologi che nel XIX secolo entrarono per primi nelle tombe delle piramidi dei faraoni egizi o a quella degli esploratori subacquei che scoprono un fondale ricco di stupende e colorate varietà marine.

Una delle più stimolanti sfide che l'uomo si è posto consiste nel tentativo di costruire delle macchine che, in qualche modo e con gli inevitabili limiti, possano essere ritenute intelligenti, ossia capaci di replicare le facoltà intellettive e pensanti dell'uomo. Sebbene tale sfida sia lungi dall'essere vinta (ma, in definitiva, tutto dipende da cosa si voglia intendere con il termine *intelligente*), purtuttavia essa rappresenta un polo attrattore degli sforzi dell'uomo.

2.2 Le macchine di Turing

Già agli inizi del ventesimo secolo si notò che il concetto di algoritmo e di macchina risultavano inadeguati quando si dovevano affrontare delle questioni delicate relative alla calcolabilità delle funzioni, alla logica matematica, alla potenziale risolubilità di alcuni problemi. Si avvertì allora l'esigenza di disporre di rigorosi modelli di calcolo che eliminassero qualsiasi forma di ambiguità. Uno dei più noti e fortunati modelli di calcolo fu introdotto dal matematico e logico inglese Alan Turing nel 1936, ancora prima che i calcolatori elettronici fossero pensati e realizzati. In onore del suo ideatore, questo modello di calcolo è denominato *macchina di Turing* (in seguito *MdT*). Dal punto di vista delle sue potenzialità la macchina di Turing risulta essere la macchina più semplice possibile in grado di fare qualcosa che può fare qualsiasi altra macchina.

Il termine *macchina* attribuito alle MdT è derivato dal fatto che questo

modello di calcolo si presta in modo naturale ad essere pensato e realizzato come un meccanismo fisico di calcolo. Una MdT, descritta nella figura 2.1, è costituita da un'unità di controllo dotata di una memoria interna che può assumere uno stato fra un insieme finito e prefissato di stati, da un nastro suddiviso in celle ciascuna delle quali può contenere un carattere appartenente ad un alfabeto finito di simboli, da una testina di lettura/scrittura che permette di leggere e scrivere sul nastro un carattere alla volta. Il nastro è illimitato in entrambe le direzioni, nel senso che, all'occorrenza, viene automaticamente allungato con delle celle vuote. Poiché in ogni istante solo una porzione del nastro è utilizzata, si assume la convenzione che la restante parte sia vuota, ossia riempita con il carattere spazio, indicato con uno dei simboli ∄, □, ␣. Si assume inoltre l'ipotesi che tale carattere faccia sempre parte dell'alfabeto della MdT, anche se non viene esplicitamente indicato.

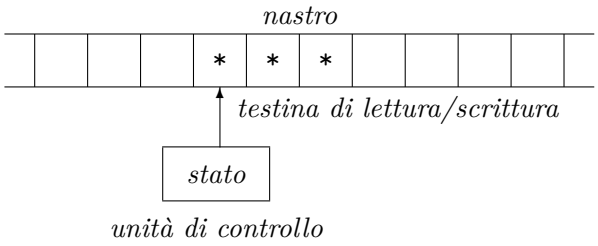


Figura 2.1: Schema di una macchina di Turing.

Ogni azione che viene intrapresa da una MdT risulta deterministicamente definita dalle seguenti informazioni:

- stato corrente dell'unità di controllo
- simbolo corrente sul nastro

Un'azione di una MdT risulta composta dalle seguenti azioni parziali:

- cambiare lo stato dell'unità di controllo
- scrivere un simbolo alla posizione attuale sul nastro
- spostarsi di una posizione a destra o a sinistra (o rimanere ferma)

Si assume la convenzione che quando in una situazione non esista alcuna indicazione che precisi come proseguire, la MdT si fermi. Viene imposta inoltre la condizione che le MdT siano *deterministiche*, ossia che in una stessa condizione di stato interno e di simbolo osservato procedano in modo univoco. Nel seguito saranno tacitamente adottate queste convenzioni.

2.3 Programmi per le macchine di Turing

Esistono diversi formalismi per descrivere il funzionamento di una MdT. Uno dei più semplici consiste nel precisare un'azione mediante una quintupla; precisamente: una quintupla

$$(p, x, q, y, m)$$

Algoritmo 1 - Interpretazione di una quintupla (p, x, q, y, m) di una MdT

1: **if** la MdT è nello stato p e legge il carattere x **then**
2: passa nello stato q
3: scrive sul nastro il carattere y
4: esegue il movimento m
5: **end if**

va interpretata come descritto nell'algoritmo 1.

Il movimento m può essere descritto mediante uno dei seguenti tre simboli:

- `<` : spostamento della testina di una posizione a sinistra
- `>` : spostamento della testina di una posizione a destra
- `-` : nessuno spostamento della testina

In base a queste premesse, un *programma per una MdT* risulta costituito da un *insieme* di quintuple. Per garantire la condizione di determinismo si impone la condizione che quintuple distinte non coincidano sulle prime due componenti.

Con un altro formalismo, una quintupla (p, x, q, y, m) può essere descritta in una tabella a doppia entrata, detta *matrice funzionale*, assumendo lo stato iniziale p ed il simbolo attuale x come indici di riga e di colonna sulla tabella ed inserendo la porzione di quintupla (q, y, m) all'interno della tabella, in corrispondenza dei due indici p e x.

Per alcune argomentazioni risulta comodo rappresentare un programma per una MdT sotto forma di automa descritto mediante un *grafo di transizione di stato*; usando questa notazione, una quintupla (p, x, q, y, m) viene rappresentata come illustrato nella figura 2.2.

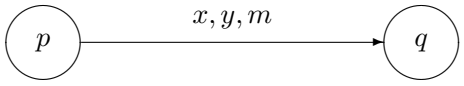


Figura 2.2: Grafo di transizione di stato che descrive la quintupla (p, x, q, y, m).

In un grafo di transizione di stato lo stato iniziale viene denotato mediante una freccia entrante e gli stati finali mediante una circonferenza a doppia linea.

Osservazione. Un programma per una MdT, in qualsiasi delle notazioni sopra descritte deve essere considerato in modo dichiarativo e non in modo imperativo, nel senso che si tratta di un *insieme di indicazioni di comportamento* e non di una *sequenza di istruzioni da eseguire*.

Esempio 2.3.1 - A seguire è descritta, mediante un insieme di quintuple (fig. 2.3), mediante una matrice funzionale (fig. 2.4) e mediante un grafo di transizione degli stati (fig. 2.5), una MdT che, partendo dallo stato iniziale `0` e posizionata sul primo carattere a sinistra di una sequenza di `a` e `b`, trasforma le `a` in `b` e viceversa e, alla fine, si riposiziona sul primo carattere a sinistra. Con `_` si denota il carattere *spazio*.

```
0,a,0,b,>
0,b,0,a,>
0,_,1,_,<
1,a,1,a,<
1,b,1,b,<
1,_,f,_,>
```

Figura 2.3: Programma di una MdT espresso mediante un *insieme di quintuple*.

	a	b	_
0	0 b >	0 a >	1 _ <
1	1 a <	1 b <	f _ >

Figura 2.4: Programma di una MdT espresso mediante una *matrice funzionale*.

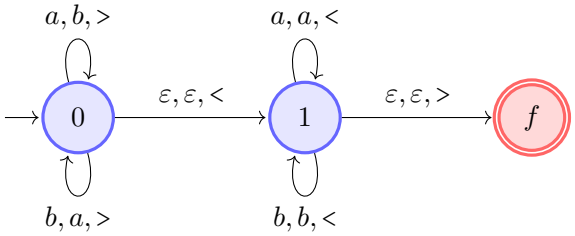


Figura 2.5: Programma di una MdT espresso mediante un *grafo di transizione di stato*.

2.4 Risoluzione di problemi con le MdT

Nonostante una MdT non comprenda il *significato* di quanto c'è scritto sul nastro iniziale e di quanto viene scritto sul nastro durante i passi della computazione, codificando opportunamente i dati sul nastro e decodificando opportunamente i risultati lasciati sul nastro alla fine della computazione, le MdT si prestano a risolvere tutte le forme di problemi finora visti (computazionali, decisionali, trasformazionali). La possibilità di scrivere sul nastro rende le MdT estremamente potenti dal punto di vista computazionale in quanto il nastro può essere utilizzato come una memoria sulla quale registrare dati e risultati intermedi prodotti nel corso dell'elaborazione.

Altre informazioni ricavabili dalla computazione di una MdT sono rappresentate dallo stato di arresto finale e dalla posizione finale sul nastro; in questo modo una MdT, anche senza modificare il nastro, è in grado di risolvere dei problemi di tipo decisionale, ad esempio decidere se la stringa presente inizialmente sul nastro soddisfa a dei prefissati requisiti oppure ricercare uno specifico carattere presente sul nastro.

Lo snodo logico e pratico riguardante l'utilizzo delle MdT risiede nel seguente quesito: *Come fa una MdT a risolvere un dato problema?* Il meccanismo dell'applicabilità di una MdT alla risoluzione di un dato problema è quello generale che sottostà all'utilizzo di un generico esecutore: c'è l'esigenza di passare dallo spazio delle soluzioni, nel quale viene ricercata e sviluppata la soluzione del problema da parte del solutore, allo spazio delle computazioni, cioè all'ambiente nel quale opera l'esecutore; per il caso particolare in cui l'esecutore sia costituito da una MdT il passaggio dallo spazio delle soluzioni allo spazio delle computazioni può essere descritto mediante lo schema riportato nella figura 2.6. Rimane il dubbio se ogni algoritmo descritto in una qualsiasi notazione algoritmica sia traducibile in quintuple per una MdT: la risposta, affermativa, è riportata nell'osservazione che segue.

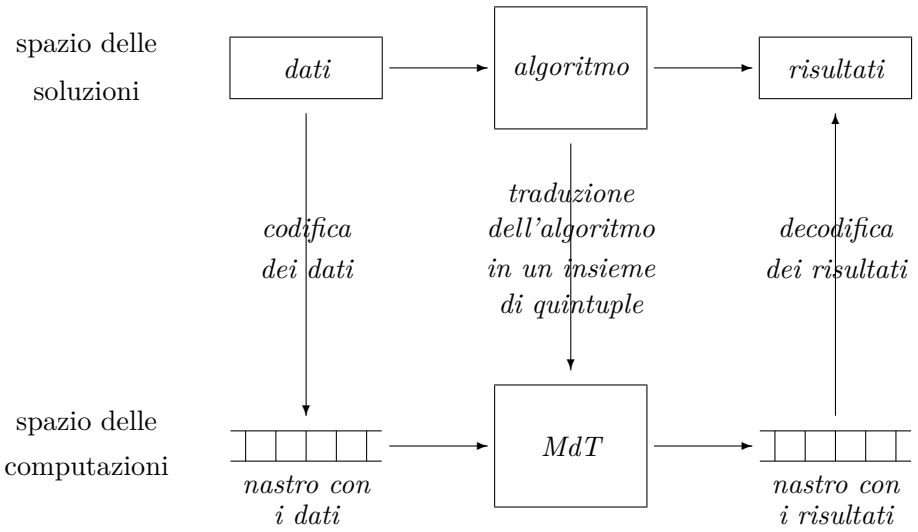


Figura 2.6: Schema del processo di risoluzione di un problema mediante una macchina di Turing.

Osservazione. Dal punto di vista delle azioni svolte, le MdT sono degli esecutori estremamente semplici e rudimentali. La loro importanza non è dovuta al fatto che rappresentino dei modelli di calcolatori reali, ma alla loro estrema semplicità di descrizione e minimalità architetturale che ben si presta a molteplici utilizzi nell'analisi di diverse questioni relative alla computabilità e per indagare sulla risolvibilità teorica dei problemi. D'altra parte, si ritiene che tale semplicità non costituisca una limitazione alla risolvibilità dei problemi, in quanto, da varie argomentazioni, si ritiene che ciò che può essere calcolato da qualsiasi calcolatore esistente o immaginabile, possa essere calcolato anche da una MdT; viceversa, per ogni problema risolvibile, esiste una MdT che lo risolve (tesi di Church-Turing).

2.5 Esempi di macchine di Turing

Negli esempi che seguono viene utilizzato il formalismo delle quintuple; ogni riga contiene una quintupla; le righe che iniziano con un carattere `#` vengono intese come commenti e non hanno alcuna influenza sulla modalità di funzionamento della MdT. Con `_` viene indicato il carattere *spazio*. Come si supporrà sempre nel seguito, la MdT inizia la computazione nello stato 0 e si trova posizionata sul primo carattere non-spazio a sinistra.

Esempio 2.5.1 - In questo esempio viene calcolato il successivo di un numero espresso in notazione unaria, usando il carattere `*`: il numero *zero* viene rappresentato con `*`, il numero *uno* con `**`, il numero *due* con `***` e cosí via. Il numero di cui si vuole calcolare il successivo si trova inizialmente codificato sul nastro; alla fine della computazione sul nastro si troverà il risultato (sempre espresso in notazione unaria).

```
0,*,0,*,>
0,_,H,*,-
```

Osservazione. Bisogna notare la differenza fra il *livello sintattico* e *livello semantico* in quanto la MdT che incrementa un numero espresso in notazione binaria non conosce i concetti di *numero*, *base di rappresentazione di un numero*, *operazione di incremento*, ...); la MdT opera al livello dei *segni*, indipendentemente dal *significato* che si voglia attribuire loro.

Problema 2.5.1 Calcolare il successivo di un numero espresso sul nastro in notazione binaria.

Soluzione. Adotteremo la classica convenzione che il numero sia scritto inizialmente sul nastro e che la MdT, alla fine della computazione lasci sul nastro il risultato. La descrizione del procedimento può essere basata sulla modalità con la quale una persona calcola manualmente il successivo di un numero. Il calcolo inizia sulla destra della rappresentazione del numero e procede verso sinistra, trasferendo verso sinistra l'eventuale riporto. Da questa osservazione emerge che il calcolo, per una MdT, avviene in due fasi consecutive, caratterizzate ciascuna dagli stati 0 ed 1 come descritto a seguire:

- 0 : *spostamento sull'ultima cifra a destra*: nello stato (iniziale) 0, qualunque cifra si incontri (`0` o `1`) la si lascia inalterata e ci si sposta verso destra fino a quando si incontra lo spazio che indica la fine della rappresentazione del numero; giunti alla fine della rappresentazione del numero, ci si posiziona sull'ultima cifra e si assume il nuovo stato 1
- 1 : *calcolo del successivo con trasferimento verso sinistra dell'eventuale riporto*: nello stato 1 se si è in corrispondenza della cifra `0` la si sovrascrive con la cifra `1` e si termina il calcolo; mentre si si è in corrispondenza della cifra `1` si continua trasferendo verso sinistra il riporto; nel caso particolare che il nastro iniziale contenga una sequenza di tutte cifre `1`, nello spostamento verso sinistra si arriva al carattere spazio che precede il primo `1` e si sovrascrive questo spazio con `1`

Tutto ciò viene sintetizzato nel seguente insieme di quintuple:

```
# posizionamento sul carattere piu' a destra
0,0,0,0,>
0,1,0,1,>
0,_,1,_,<
# computazione del successivo
1,0,H,1,-
1,1,1,0,<
1,_,H,1,-
```

Osservazione. Dal confronto fra l'esempio 2.5.1 e la soluzione del problema 2.5.1 si nota che utilizzando un maggior numero di simboli per codificare i dati (per risolvere uno stesso problema) si ha un aumento del numero di quintuple necessario per descrivere la soluzione. Seguendo l'obiettivo di massima semplicità si tende ad usare un numero molto ridotto di simboli del nastro. In tale modo la descrizione di una MdT risulta semplificata. Questa semplicità va ad appesantire l'esecuzione, ma questo fatto risulta un problema marginale nelle questioni che coinvolgono le MdT.

Problema 2.5.2 Realizzare mediante una MdT l'algoritmo di Euclide per il calcolo del massimo comune divisore di due numeri naturali presenti sul nastro, espressi in notazione unaria.

Soluzione. A seguire è riportato un insieme di quintuple per una MdT che realizzano l'algoritmo di Euclide. Si lascia per esercizio la comprensione di come queste quintuple costituiscano la *codifica* dell'algoritmo di Euclide.

```
0,_,0,_,>    3,_,4,_,>    6,_,6,_,<    9,_,2,_,>
0,1,1,1,<    3,1,3,1,>    6,1,1,1,<    9,1,1,1,<
1,_,2,1,>    4,_,4,_,>    7,_,7,_,<    A,1,A,1,<
1,1,1,1,<    4,1,5,_,>    7,1,8,1,<    A,_,H,_,>
2,_,A,_,>    5,_,7,_,<    8,_,9,_,<
2,1,3,_,>    5,1,6,1,<    8,1,8,1,<
```

Osservazione. Come si nota dalla soluzione del problema 2.5.2, le MdT richiedono che venga precisato un consistente numero di quintuple; questa situazione era prevedibile in base al *principio di complessità della soluzione*. Ciò può risultare scoraggiante, specialmente quando si tratta di risolvere un problema di una certa complessità. Ma le MdT non vengono utilizzate per risolvere in pratica problemi significativi e complessi, quanto piuttosto come strumento teorico per indagare e decidere sulla potenziale risolvibilità di alcuni problemi.

2.6 Algoritmi e programmi per una macchina di Turing

Il problema 2.5.2 evidenzia che scrivere un programma per una MdT può risultare impegnativo a causa della distanza esistente fra il linguaggio delle quintuple compreso dalla macchina e l'idea sulla quale si basa la soluzione del problema. Per agevolare la scrittura del programma conviene scrivere un algoritmo, come descritto nella soluzione del problema che segue.

Problema 2.6.1 Stabilire se una sequenza di parentesi tonde è bilanciata secondo le usuali regole di costruzione delle espressioni.

Soluzione. L'algoritmo risolutivo si basa sulla seguente idea: le parentesi vengono scandite da sinistra verso destra; nella scansione il numero delle parentesi aperte incontrate deve essere non inferiore al numero di parentesi chiuse; più semplicemente è sufficiente controllare, mediante un contatore, che il numero di parentesi aperte in eccesso rispetto a quelle chiuse sia maggiore o uguale a 0. Il procedimento è descritto dall'algoritmo 2.

Algoritmo 2 - Controllo se una sequenza di parentesi è bilanciata.

```
1: imposta il contatore delle parentesi a 0
2: scandisci i caratteri del nastro da sinistra a destra
3: for ciascun carattere c del nastro do
4:     if c = ( then
5:         incrementa il contatore di un'unità
6:     else
7:         decrementa il contatore di un'unità
8:     end if
9: end for
10: if si è arrivati sul carattere spazio e lo stato è 0 then
11:     le parentesi sono corrette
12: else
13:     le parentesi sono errate
14: end if
```

L'algoritmo 2 è impostato secondo una logica imperativa, ma può essere immediatamente tradotto in modalità dichiarativa mediante un insieme di quintuple. Per semplificare la soluzione del problema supponiamo che il massimo livello di annidamento delle parentesi sia pari a 3; questa ipotesi semplificativa permette di gestire il contatore delle parentesi mediante degli stati: vengono utilizzati gli stati da 0 (iniziale) a 3 (pari al massimo livello di annidamento concesso). Il risultato viene fornito mediante lo stato finale di arresto costituito da una stringa che descrive l'esito della computazione. Questa stringa descrive anche le situazione di errore che si annidano nelle linee 5, 7 e 13 dell'algoritmo 2. Il programma in quintuple è riportato a seguire.

```
# Test se una sequenza di parentesi tonde e' bilanciata
0,(,1,(,>
0,),Errore: troppe parentesi chiuse,),-
```

```
0,_,Ok: parentesi corrette,_,-
1,(,2,(,>
1,),0,),>
1,_,Errore: mancano parentesi,_,-
2,(,3,(,>
2,),1,),>
2,_,Errore: mancano parentesi,_,-
3,(,Errore: troppe parentesi annidate,(,-
3,),2,),>
3,_,Errore: mancano parentesi,_,-
```

2.7 Forme di elaborazione di una macchina di Turing

Una MdT è sostanzialmente un calcolatore che elabora *stringhe*, ossia sequenze di caratteri. Questa generica caratterizzazione si declina in modi diversi.

Accettazione di stringhe

Una MdT può essere usata per controllare se una stringa è *sintatticamente* corretta, ossia se soddisfa a specifiche regole formali: la stringa da analizzare viene scritta sul nastro, la MdT la analizza e poi fornisce il risultato del controllo svolto. In questo modo la MdT assume il ruolo di *accettatore* di stringhe e definisce una corrispondenza fra la macchina stessa ed il linguaggio formato dalle stringhe accettate dalla macchina. Il risultato di accettazione può essere desunto dallo stato finale in cui la MdT si ferma; si possono a questo scopo utilizzare i due stati `TRUE` (accettazione) e `FALSE` (non accettazione).

Problema 2.7.1 Decidere se una data stringa è formata da una successione di `a` seguite da una successione di `b`. Ad esempio, devono essere riconosciute le seguenti stringhe: `ab`, `aabbbb`.

Soluzione. Si tratta di un problema di tipo *decisionale*: la stringa viene esaminata, da sinistra verso destra, una sola volta, senza essere modificata; alla fine la MdT si troverà nello stato `TRUE` se la stringa scritta inizialmente sul nastro è corretta, altrimenti nello stato `FALSE`. L'analisi della stringa avvine in fasi successive, ciascuna caratterizzata dagli stati 0, 1 e 2 come descritto a seguire:

0 : *superamento della prima* `a`: nello stato (iniziale) 0 si può proseguire, passando nello stato 1, solo se si incontra una `a`; negli altri casi (`b` oppure `_`), la stringa viene riconosciuta come errata, portando la MdT nello stato `FALSE`.

1 : *superamento delle altre eventuali* `a`: nello stato 1 significa che si è incontrata una `a` all'inizio e si continua fino ad oltrepassare tutte le eventuali `a` che seguono, fino ad incontrare la prima `b`, nel qual caso la MdT entra nello stato 2; se, terminate le `a`, si incontra uno spazio la stringa è errata e la MdT entra nello stato finale `FALSE`.

2 : *superamento delle* `b`: nello stato 2 significa che si è incontrata una `b` dopo una sequenza di `a` e si continua fino ad oltrepassare tutte le eventuali altre `b` che seguono, fino ad incontrare un carattere diverso da `b`: se si incontra una `a` significa che la stringa è errata mentre se si incontra un carattere spazio significa che la stringa è corretta (stato finale `TRUE`).

Quanto descritto sopra può essere codificato in un insieme di quintuple descritte nel seguente programma

```
# superamento della prima a
0,a,1,a,>
0,b,FALSE,b,>
0,_,FALSE,_,-
# superamento delle altre eventuali a fino alla prima b
1,a,1,a,>
1,b,2,b,>
1,_,FALSE,_,-
# superamento delle b
2,a,FALSE,a,-
2,b,2,b,>
2,_,TRUE,_,-
```

oppure, equivalentemente, mediante la seguente matrice funzionale.

	a	b	_
0	1 a >	FALSE b >	FALSE _ -
1	1 a >	2 b >	FALSE _ -
2	FALSE a -	2 b >	FALSE _ -

□

Trasformazioni di stringhe

Nella sua forma più generale, com'è stata descritta nei paragrafi precedenti, una MdT T può essere pensata come un *trasformatore di stringhe*: la stringa s presente come dato scritto sul nastro all'inizio della computazione viene trasformata in un'altra stringa $s' = T(s)$ presente sul nastro alla fine della computazione (nel caso la MdT si fermi), secondo lo schema di trasformazione illustrato nella figura 2.7.

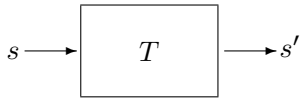


Figura 2.7: Processo di trasformazione di una stringa s in una stringa s' mediante una MdT T.

Esistono molti ed interessanti problemi che possono essere inquadrati e risolti mediante un processo di trasformazione di stringhe. Ad esempio, *rovesciare una stringa*. Questa accezione di MdT come *trasformatore di stringhe*

non è comunque limitante in quanto qualsiasi dato di altra natura (numero, sequenza di numeri, espressione simbolica, ...) può essere trasformato in stringa e successivamente la stringa lasciata sul nastro alla fine della computazione può essere interpretata come numero o altra tipologia di valore.

Con queste premesse una MdT può calcolare funzioni della forma $\mathbb{N}^k \to \mathbb{N}$ partendo con il nastro iniziale sul quale è impressa la codifica di k numeri naturali spaziati, ad esempio, mediante un carattere *spazio*.

Il processo di trasformazione di stringhe mediante le MdT può essere descritto in modo funzionale, assimilando la MdT T ad una funzione parziale:

$$s \quad \mapsto \quad s'$$

Questa modalità di interpretazione suggerisce che più MdT $T_1, T_2, \ldots, T_n$ (possibilmente diverse) possano essere fatte agire in sequenza, ognuna agente sul nastro lasciato scritto dalla MdT precedente; il risultato finale, presente sul nastro alla fine dell'elaborazione della MdT T_n, sarà uguale a

$$T_n(T_{n-1}(\ldots T_2(T_1(s))))$$

ottenuto dalla catena di trasformazioni $s_0 \to s_1 \to \cdots \to s_n$, indicando con s_0 la stringa presente sul nastro all'inizio della computazione e con s_i la stringa lasciata scritta sul nastro dalla MdT T_i. Per rendere univoca l'attivazione in sequenza di più MdT, si assume l'ipotesi che ogni MdT inizi la propria fase di elaborazione partendo dal primo carattere non-spazio a sinistra. L'idea della concatenazione di MdT viene utilizzata anche nei moderni linguaggi di programmazione e trova riscontro nello strumento dei sottoprogrammi.

2.8 La macchina di Turing universale

Da quanto è stato esposto nei precedenti paragrafi, una MdT è caratterizzata da uno specifico insieme di quintuple e quindi è in grado di risolvere una specifica classe di problemi. Una generalizzazione di questa situazione consiste nell'avere un'unica MdT programmabile, in grado di simulare il comportamento di ogni qualsiasi altra MdT, leggendo dal nastro, come input, le quintuple, consultando il nastro durante il processo di computazione per scegliere la quintupla da *eseguire* man mano che procede il processo di computazione. Tale MdT viene detta *macchina di Turing universale*. Al di là della complessità di realizzazione pratica, una tale MdT è facilmente immaginabile; tale idea di generalizzazione ha avuto importanti ricadute pratiche ed ha tracciato la linea dello sviluppo dei calcolatori attuali secondo l'architettura di von Neumann, caratterizzata dall'avere in memoria sia il programma che i dati che vengono elaborati, come avviene nella stragrande maggioranza degli attuali calcolatori.

2.9 Computabilità

Ogni disciplina e più in generale ogni sistema di pensiero, una volta raggiunto uno stadio di maturità nel suo sviluppo, intraprende un percorso di analisi introspettiva alla ricerca ed alla definizione dei propri fondamenti con l'obiettivo di darsi uno status di teoria o di scienza. È successo con la Matematica all'inizio del XX secolo quando Hilbert intraprese il suo progetto di definire in modo rigoroso ed assoluto i fondamenti della Matematica, senza peraltro riuscirvi. Per l'informatica questo percorso ebbe inizio verso gli anni '30 del secolo scorso quando logici e matematici iniziarono ad indagare sulle modalità del ragionamento umano e sui meccanismi di calcolo. Tutti questi studi e ricerche rientrano nella tematica denominata *computabilità*, una branca di studio che ha come obiettivo primario la definizione delle potenzialità e dei limiti del ragionamento umano e delle macchine. La teoria della computabilità permette di tracciare una linea di demarcazione fra ciò che è computabile e ciò che non lo è, ossia fra i problemi che (almeno in linea teorica) sono risolvibili e problemi che non lo sono.

Le tematiche inerenti alla computabilità coinvolgono varie discipline: informatica, tecnologia, matematica, filosofia, epistemologia, ed altre ancora. L'argomento si presta, quindi, ad essere trattato ed indagato, dall'interno di ciascuna disciplina, in modi diversi e con diversi accenti. In ambito informatico e nella scienza dei calcolatori la computabilità ha trovato un efficace strumento di indagine nelle *macchine di Turing* che possono essere considerate dei precursori degli attuali meccanismi automatici di calcolo (calcolatori) e di azione (robot).

La Tesi di Church-Turing

Esistono moltissimi problemi per i quali è stata fornita una rigorosa dimostrazione del fatto che essi non sono risolvibili. Tale limitazione non è dovuta all'incapacità del solutore nel formulare un adeguato algoritmo, ma alla natura intrinseca del problema ed ai limiti della logica, della matematica e del linguaggio. Nella dimostrazione di non risolvibilità di questi problemi gioca un ruolo fondamentale la seguente ipotesi, formulata da Church e Turing nel 1937, che, da varie argomentazioni, è plausibile ritenere che sia vera, cosicché si è meritata l'appellativo di *tesi*.

TEOREMA 1 (Tesi di Church-Turing)**.** Tutti i meccanismi di calcolo (MdT, personal computer, mainframe, calcolatori paralleli, ...) finora realizzati (dato di fatto dimostrato) e realizzabili in futuro (ipotesi plausibile) sono equivalenti, cioè possono risolvere gli stessi problemi.

Dalla tesi di Church-Turing discende il seguente corollario.

TEOREMA 2 (Corollario della tesi di Church-Turing)**.** Ogni problema è risolvibile se e solo se esiste una MdT che lo risolve.

In base a questo risultato si può parlare di *problemi risolvibili* e *problemi non risolvibili* senza fare alcun riferimento all'esecutore. Operativamente, si dimostra che un problema è non risolvibile dimostrando che non esiste alcuna

MdT (e quindi nessun'altra macchina, per la tesi di Church-Turing) che lo risolve. La tesi di Church-Turing evidenzia i limiti del calcolo automatico tracciando una linea di separazione tra ciò che una macchina può fare e ciò che non può fare.

Osservazione. I risultati di Church e Turing presentati in questo paragrafo segnano una sorta di ideale confine invalicabile dai metodi e dagli strumenti della scienza.

Il Problema della fermata

Un altro famoso problema, considerato inizialmente da Turing e per il quale lo stesso Turing ha fornito una dimostrazione di non decidibilità, è noto come *Problema della fermata* :

> *Dato un generico programma P (insieme di quintuple di una MdT, programma scritto in un qualche linguaggio di programmazione o altro formalismo) ed un insieme di dati di input I, stabilire se il programma P, avendo come input I, si fermerà o no.*

La non risolvibilità di questo problema è l'oggetto del seguente teorema.

TEOREMA 3. Il problema della fermata è indecidibile, ossia non esiste alcun procedimento in base al quale, dato un programma P ed un insieme di dati di input I sui quali opera il programma P, si possa decidere se il programma P si fermerà.

Il problema della fermata acquista particolare rilevanza nell'ambito della teoria della programmazione in quanto afferma che è impossibile stabilire se un generico programma giungerà o no al termine dell'esecuzione; esistono comunque dei programmi per i quali si riesce a dimostrare la terminazione o la non terminazione. La non risolvibilità del problema della fermata acquista anche una valenza strumentale per la soluzione di altre classi di problemi che si dimostra essere non risolvibili seguendo una linea di ragionamento come la seguente: si dimostra che tali classi di problemi sono riconducibili al problema della fermata e pertanto, se ammettessero soluzione, risulterebbe risolvibile anche il problema della fermata. Da questa contraddizione si deduce che queste classi di problemi sono non risolvibili. Ad esempio, si dimostra in questo modo che la decisione se due programmi risultino equivalenti è non risolvibile.

Osservazione. Il quadro avvilente, presentato in questo paragrafo, riguardante la risolvibilità di importanti classi di problemi, viene in parte riscattato dal fatto che spesso si considerano particolari sottoinsiemi di queste classi di problemi e, per questi sottoinsiemi, si riesce a trovare un'adeguata soluzione e si è in grado di dimostrare che l'esecuzione giungerà al termine per ogni possibile istanziazione dei dati di input.

Funzioni computabili

Una funzione dicesi *computabile* o *calcolabile* se esiste un procedimento meccanico (algoritmo) che permette di determinare il valore della funzione per *ogni*

possibile valore degli argomenti. Nel caso in cui il processo di calcolo, per qualche valore dei dati di input, non termini si parla di *funzioni parzialmente computabili*.

Per le considerazioni che seguiranno ci limiteremo a considerare funzioni sui numeri naturali.

Esempio 2.9.1 - Sono esempi di funzioni computabili le seguenti:

$$\begin{aligned} n &\mapsto 2^n \\ (m,n) &\mapsto \text{massimo comune divisore fra } m \text{ ed } n \\ n &\mapsto n\text{-esima cifra decimale di } \sqrt{2} \end{aligned}$$

La distinzione fra funzioni computabili e non computabili non avrebbe ragione d'esistere fintantochè non si sia dimostrato che esistono funzioni non computabili oppure non si sia prodotto un esempio di funzione non computabile. È questo l'oggetto del teorema che segue.

TEOREMA 4. Esistono funzioni non computabili.

Dimostrazione. Ogni MdT è descritta mediante il suo programma (insieme di quintuple); tale programma può essere convertito in una stringa (ad esempio concatenando le stringhe che codificano ciascuna quintupla). Le stringhe che codificano le MdT possono essere enumerate in sequenza, ad esempio in ordine alfabetico. Le funzioni fra numeri naturali (della forma $\mathbb{N} \to \mathbb{N}$) non sono numerabili. Quindi esistono funzioni non computabili. □

Nella dimostrazione del precedente teorema abbiamo ottenuto un risultato più forte di quanto ci eravamo prefissi: le funzioni non computabili sono la stragrande maggioranza fra tutte le funzioni ossia, in altri termini, "quasi" tutte le funzioni fra numeri naturali sono non computabili. Il fatto che questa conclusione contrasti con l'opinione comune che le funzioni siano tutte computabili dipende dal fatto che le funzioni che si incontrano spontaneamente e che si costruiscono sono definite apposta per essere computabili.

Esibire degli esempi di funzioni non computabili non è facile e serve un po' di inventiva. Un classico esempio di funzione non computabile [1] è rappresentato dalla funzione $f : \mathbb{N} \to \mathbb{N}$ definita come segue:

$$f(n) = \begin{cases} 1 & \text{se nella rappresentazione decimale di } \pi \\ & \text{c'è una successione di } n \text{ cifre 7 consecutive} \\ 0 & \text{altrimenti} \end{cases}$$

Esssendo che un algoritmo è concettualmente assimilabile ad una funzione, si può parlare indifferentemente di *problemi risolvibili* e *funzioni computabili*.

[1] Riportato in N. Cutland, *Computability*, Cambridge Univ. Press, 1980.

2.10 Macchine combinatorie e sequenziali

Dal punto di vista di un utente, astraendo da qualsiasi dettaglio implementativo e considerando solamente la sua funzionalità verso l'esterno, una macchina può essere considerata come una scatola nera che non lascia trasparire all'esterno il meccanismo di funzionamento interno. La macchina manifesta all'esterno, verso l'utente, i risultati attraverso dei dispositivi di output, in una qualche modalità (testo, grafica, audio, ...). Lo schema spesso utilizzato per descrivere una tale macchina è riportato nella figura 2.8: l'utente sollecita la macchina fornendo in ingresso valori, espressioni, comandi, interrogazioni, regole, istruzioni e la macchina risponde fornendo dei risultati in uscita.

Figura 2.8: Schema di una macchina generica.

Lo schema descritto nella figura 2.8 può riferirsi ad una situazione fisica in cui gli ingressi e le uscite sono oggetti concreti manipolati e prodotti da un meccanismo automatico oppure ad una situazione astratta in cui la macchina corrisponde ad un operazione (sui valori in ingresso) che fornisce il risultato in uscita.

Osservazione. Negli schemi a blocchi simili a quello descritto nella figura 2.8 si assume la convenzione che gli ingressi siano posti a sinistra e le uscite a destra, con il verso di flusso da sinistra verso destra. Adottando questa convenzione vengono omessi i versi delle freccie che vengono indicate mediante un segmento.

Esempio 2.10.1 - Un distributore automatico di bevande può essere visto come una macchina in cui gli ingressi sono le monete e la scelta della bevanda desiderata e come uscita la bevanda scelta e le monete di resto.

Figura 2.9: Schema di un distributore di bevande secondo l'ottica del cliente.

Uno stesso meccanismo può essere visto in modi diversi, a seconda di chi lo considera; il distributore di bevande visto sopra, dal punto di vista dell'azienda proprietaria viene visto come una macchina avente gli ingressi costituite dalle materie prime e prodotti che vengono inseriti come ingredienti e come uscita le monete di ricavo inseriti dai vari utenti:

Figura 2.10: Schema di un distributore di bevande secondo l'ottica del proprietario.

Macchine combinatorie

Una *macchina combinatoria* è caratterizzata dal fatto che le uscite dipendono unicamente dagli ingressi e non sono influenzate dalla storia degli ingressi precedenti; in altri termini, una macchina combinatoria è senza *memoria*.

Esempio 2.10.2 - Consideriamo una delle più semplici macchine combinatorie: una macchina in grado di addizionare due numeri in ingresso e dare in uscita la loro somma; possiamo descriverla come indicato nella figura 2.11.

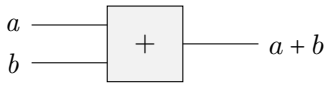


Figura 2.11: Macchina addizionatrice a 2 ingressi.

Esempio 2.10.3 - Nella figura 2.12 sono descritte due equivalenti macchine combinatorie che risolvono una generica equazione di primo grado $ax + b = 0$, con $a \neq 0$. Sono composte da un operatore che inverte i due ingressi, da uno che cambia il segno dell'ingresso e da un operatore aritmetico di divisione.

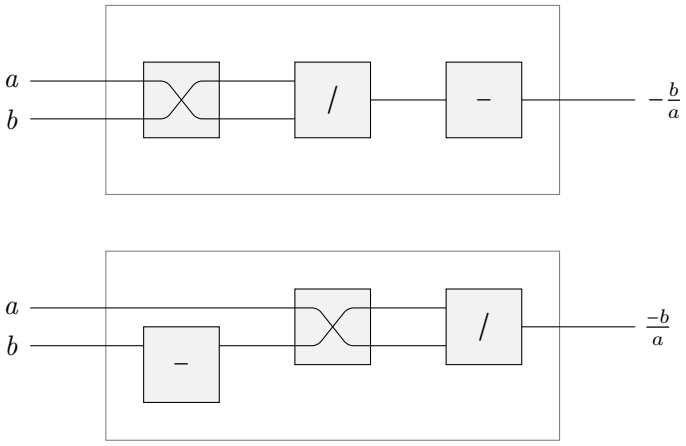


Figura 2.12: Due equivalenti macchine che risolvono una generica equazione di primo grado della forma $ax + b = 0$.

Macchine sequenziali

Adottando l'impostazione tipica dei linguaggi di programmazione di tipo imperativo, il solutore ha a disposizione come esecutore una macchina con memoria alla quale vengono rivolti dei comandi che, eseguiti, modificano lo stato della macchina stessa in un modo che dipende dallo stato precedente della macchina. Una macchina con questa struttura è detta *macchina sequenziale* ed è descrivibile mediante lo schema riportato nella figura 2.13 dove gli ingressi modificano lo stato della macchina stessa ed i risultati dipendono dagli ingressi e dallo stato della macchina; inoltre, lo stato viene modificato assumendo un valore dipendente dagli ingressi e dallo stato precedente.

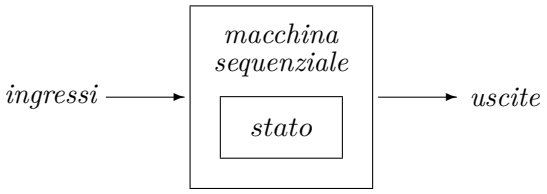


Figura 2.13: Schema di una macchina sequenziale.

In una macchina sequenziale gli ingressi assumono generalmente la forma descritta nella figura 2.14, dove c è un comando e $a_1, a_2, \ldots, a_n$ sono gli argomenti che specificano gli argomenti del comando.

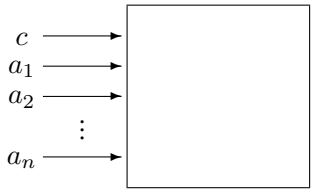


Figura 2.14: Struttura di un comando inviato ad una macchina sequenziale.

Un comando come quello appena descritto viene solitamente scritto nella seguente notazione funzionale:

$$c(a_1, a_a, \ldots, a_n)$$

Osservazione. Il fatto che le macchine sequenziali, a parità di comportamento, abbiano un *loro* stato le rende distinguibili le une dalle altre. Per questo motivo, quando se ne utilizzi più d'una si rende necessario denotarle mediante un nome identificativo.

2.11 Costruire una macchina

Per realizzare una macchina si possono sostanzialmente adottare due strategie: assemblare opportunamente dei componenti elementari di base (dotati

di una loro ben determinata funzionalità) oppure prendere una macchina già costruita, non dotata di alcuna specifica funzionalità, ma in grado di recepire un insieme di istruzioni (*programmi*) che ne regoleranno il funzionamento. La versatilità di questa seconda tipologia di macchine si fonda proprio sulla possibilità di variare il loro modo di agire cambiando il programma. Queste due strategie possono essere adotatte contemporaneamente, *costruendo* delle *macchine programmabili*.

Esempio 2.11.1 - Costruiamo ora una macchina addizionatrice a 3 ingressi, avente la specifica funzionale descritta dalla figura 2.15.

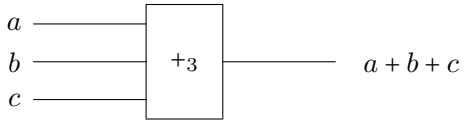


Figura 2.15: Macchina addizionatrice a 3 ingressi.

L'idea di costruire un'altra macchina, con una struttura interna specifica corrispondente alla funzionalità che deve espletare, risulta poco produttiva. La strategia migliore consiste nel costruire piccole macchine specializzate (ad esempio per addizionare 2 numeri) e di comporle per costruire macchine più complesse (ad esempio per addizionare più di 2 numeri). La costruzione della macchina addizionatrice a 3 ingressi può avvalersi della precedente macchina a 2 ingressi. Una possibile implementazione è riportata nella figura 2.16.

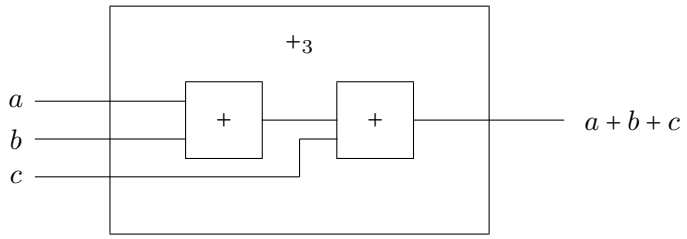


Figura 2.16: Macchina addizionatrice a 3 ingressi.

Osservazione. La strategia costruttiva descritta nel precedente esempio è tipica dell'informatica (sia per l'hardware che per il software): si costruiscono piccole macchine o funzioni software elementari componendo le quali si realizzano macchine o programmi complessi.

Una macchina per disegnare

Un interessante e noto esempio di macchina sequenziale è costituito dalla *tartaruga* dell'omonima della *grafica*. La tartaruga è un automa che si muove su un piano, lasciando una linea dove passa; ha attaccata una penna che può essere alzata ed abbassata sul piano di disegno; ha una direzione di avanzamento

e può essere manovrato mediante degli appositi comandi, i più importanti e caratteristici dei quali sono elencati a seguire:

forward(n)	avanza di n passi
left(a)	gira a sinistra di un angolo a
right(a)	gira a destra di un angolo a
penup	alza la penna dal foglio
pendown	abbassa la penna sul foglio

Il comando *forward*(n), se la penna è abbassata sul foglio, ha l'effetto di tracciare un segmento di lunghezza n a partire dalla posizione corrente e lungo la direzione di avanzamento della tartaruga. L'angolo a di rotazione nei comandi *left*(a) e *right*(a) viene espresso in gradi sessagesimali. Questa macchina disegnatrice è un esempio di macchina sequenziale in cui lo stato è costituito dai seguenti valori:

- posizione dove si trova la tartaruga
- direzione di avanzamento della tartaruga
- stato su/giù della penna

Usando questo automa è possibile disegnare figure anche molto complesse. Limitandosi ad utilizzare i pochi comandi elencati sopra, mediante l'algoritmo 3 si può disegnare la figura composta da 10 triangoli equilateri di lato 100, incentrati in uno stesso punto, ciascuno ruotato di 36 gradi rispetto al successivo (figura 2.17).

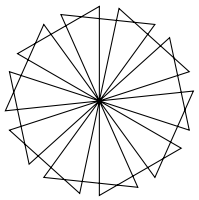


Figura 2.17: Figura composta da 10 triangoli equilateri di lato 100.

La figura 2.17 può essere generata mediante l'algoritmo 3.

Algoritmo 3 - Disegno di una stella di triangoli equilateri

```
1: for 10 times
2:    for 3 times
3:       forward(100)
4:       left(120)
5:    end for
6:    left(36)
7: end for
```

Il controllo della forma **for** n **times** *azioni* **end for** che compare nell'algoritmo 3 ha l'effetto di far ripetere ciclicamente per n volte le *azioni* riportate all'interno del ciclo.

Una macchina per memorizzare

Nell'ambito della programmazione un utilizzatissimo esempio di macchina sequenziale è costituito da una *variabile*: è una macchina avente la funzione di memorizzare un valore e renderlo accessibile successivamente, all'occorrenza. A questo scopo su una variabile sono predisposti due comandi di base:

set: memorizza un valore nello stato della variabile (l'eventuale valore precedentemente memorizzato viene sovrascritto e perso)

get: fornisce in uscita il valore precedentemente memorizzato variabile

La figura 2.18 descrive lo schema dei comandi *set* e *get* applicati ad una variabile.

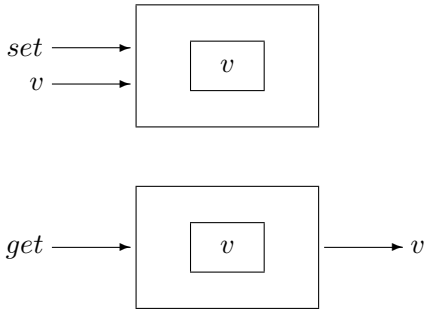


Figura 2.18: I comandi *set* e *get* su una variabile.

I comandi *set* e *get* applicati ad una generica variabile x vengono solitamente espressi come segue:

$x \leftarrow v$ memorizza nella variabile x il valore v
x valore memorizzato nella variabile x

Esempio 2.11.2 - Con le convenzioni di scrittura date sopra, per esprimere l'istruzione che incrementa di un'unità il valore memorizzato nella variabile x si scriverà

$$x \leftarrow x + 1$$

2.12 Macchine programmabili

Consideriamo una macchina $\mathcal{M}$ del tipo di quelle esaminate nei paragrafi precedenti, che risolve dei problemi. Se $\mathcal{M}$ fosse in grado di risolvere tutti i problemi (risolvibili) risulterebbe troppo complicata. Se risolvesse solo i problemi di una classe ben determinata, per risolvere tante classi di problemi servirebbero troppe macchine (una per ciascuna classe di problemi). La strategia vincente consiste nell'avere un'unica macchina universale $\mathcal{M}_{\mathcal{U}}$ che, di volta in volta, viene istruita a risolvere una ben specifica classe di problemi. Per far sì che la macchina universale si adegui alla soluzione di diverse classi di problemi, anziché dotarla di un meccanismo intrinseco che ne governa il funzionamento, viene fornita alla macchina, dall'esterno, la descrizione del procedimento da eseguire, (*programma*), che risolve una classe di problemi. Un programma può essere

visto come una scheda intercambiabile da inserire in una macchina universale in modo da renderla capace di risolvere una specifica classe di problemi. Tali macchine vengono dette *macchine programmabili* e costituiscono una naturale evoluzione delle macchine sequenziali con stato: al momento dell'attivazione iniziale le macchine programmabili vengono dotate di uno specifico programma che ne definisce lo stato iniziale ed inoltre determina le successive fasi dell'elaborazione. La struttura di una macchina programmabile che risolve una classe di problemi alla quale appartiene l'istanza di problema P della classe $\mathcal{C}$ può essere descritta mediante lo schema riportato nella figura 2.19.

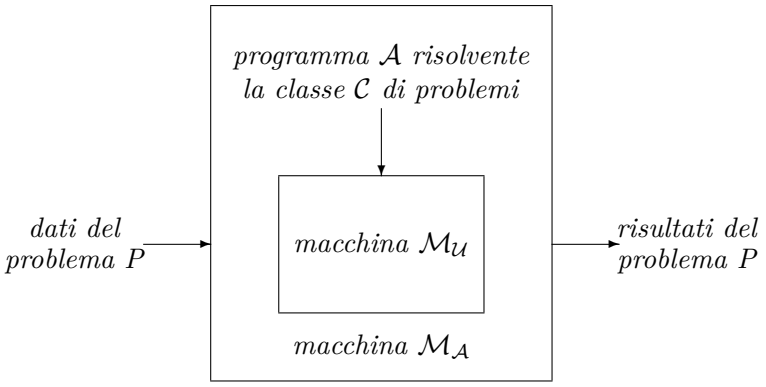


Figura 2.19: Schema di una macchina programmabile.

Nella figura 2.19 il blocco tratteggiato costituito dal programma $\mathcal{A}$ unitamente alla macchina universale $\mathcal{M}_{\mathcal{U}}$ può essere pensato come una macchina $\mathcal{M}_{\mathcal{A}}$ (a logica cablata) univocamente individuata dal programma $\mathcal{A}$. Per le macchine programmabili si presenta l'esigenza di avere un linguaggio $\mathcal{L}$ mediante il quale descrivere alla macchina $\mathcal{M}_{\mathcal{U}}$ il programma $\mathcal{A}$. Ovviamente, in questo caso il programma $\mathcal{A}$ deve essere descritto nel linguaggio $\mathcal{L}$ compreso dalla macchina $\mathcal{M}_{\mathcal{U}}$.

2.13 Macchine virtuali

Con riferimento ad una macchina programmabile, se il programma $\mathcal{A}$ viene considerato come facente parte della macchina che lo esegue, la macchina si configura come una *macchina virtuale* in grado di risolvere i problemi della classe della quale il programma $\mathcal{A}$ costituisce una soluzione. È questa la situazione in cui si trova un utente di un dato programma applicativo. Una macchina virtuale è costituita da un insieme di risorse hardware che rappresentano la macchina fisica e da un programma o insieme di programmi che costituiscono il software mediante il quale si riesce a far eseguire alla macchina le azioni per raggiungere i risultati desiderati. Ad un utente la distinzione fra la macchina fisica ed i programmi che la controllano risulta del tutto trasparente: le risorse hardware (macchina fisica) e software (programmi) si configurano come un esecutore virtuale che agisce in base a quanto definito dal programma che si sta utilizzando. Il punto di vista più interessante e positivo di questa

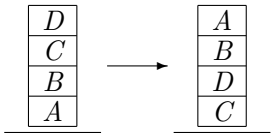
impostazione sta nel fatto che un utente di una macchina virtuale non si deve preoccupare (e dal punto di vista logico non dovrebbe neanche pensare) di *come* è fatta la macchina virtuale che utilizza, ma deve solamente sapere *cosa* fa. Naturalmente, la macchina virtuale vista dall'utente può essere composta da più strati. La suddivisione in strati risulta comunque trasparente all'utente, il quale vede un'unica macchina virtuale.

L'idea di macchina virtuale ha trovato di recente importanti realizzazioni nelle macchine virtuali Java, Python ed altre, sulle quali si basano gli omologhi linguaggi.

2.14 Il principio di complessità

In questo paragrafo, prendendo spunto da un esempio, si analizzerà la relazione che intercorre tra la potenza di un esecutore e la complessità della soluzione per risolvere un dato problema.

Esempio 2.14.1 - Consideriamo il problema di trasformazione di una configurazione di blocchi, come illustrato dallo schema che segue:



Possiamo pensare che qui l'esecutore sia una persona, un robot o qualcos'altro. Per rendere il problema un po' interessante supponiamo, per il momento, che l'esecutore abbia la capacità di spostare un solo blocco alla volta. Con questa ipotesi, una soluzione del problema è fornita dall'algoritmo 4.

Algoritmo 4 - Algoritmo spostamento blocchi (caso a)

1: sposta il blocco D sul piano
2: sposta il blocco C sul piano
3: sposta il blocco D sopra a C
4: sposta il blocco B sopra a D
5: sposta il blocco A sopra a B

In questo contesto, il *solutore* è colui che ha scritto la sequenza delle azioni, l'*esecutore* è l'entità che effettua lo spostamento dei blocchi, gli *oggetti* che vengono manipolati sono costituiti dai blocchi, le *azioni* consistono nello spostare i blocchi, i *dati* sono rappresentati dalla configurazione iniziale ed i *risultati* dalla configurazione finale che si vuole ottenere.

Se ammettiamo che l'esecutore abbia la capacità di spostare una qualsiasi pila di blocchi, senza capovolgerla, la soluzione del problema può essere formulata come descritto nell'algoritmo 5.

Algoritmo 5 - Algoritmo spostamento blocchi (caso b)

1: sposta la pila di blocchi CD sul piano
2: sposta il blocco B sopra a D
3: sposta il blocco A sopra a B

Da questa semplice variazione abbiamo la riconferma che la soluzione dipende dalle capacità dell'esecutore.

Se ammettiamo che l'esecutore abbia anche la capacità di spostare una qualsiasi pila di blocchi e possa, eventualmente, appoggiarla capovolta, una soluzione può essere più semplicemente espressa come descritto nell'algoritmo 6.

Algoritmo 6 - Algoritmo spostamento blocchi (caso c)

1: sposta la pila di blocchi CD sul piano (senza capovolgerla)
2: sposta la pila di blocchi AB sopra D capovolgendola

Le diverse soluzioni presentate nell'esempio precedente evidenziano che la soluzione di un problema non solo dipende dalle capacità dell'esecutore ma che la soluzione è tanto più breve e semplice quanto più potenti sono le capacità dell'esecutore. La lunghezza di una soluzione, misurata come numero delle istruzioni elementari che la compongono, rappresenta un'indicazione della complessità della soluzione. Se la soluzione viene espressa in una forma non sequenziale ma strutturata mediante delle strutture (condizionali e/o cicliche) di controllo delle azioni, il concetto di complessità deve essere espresso in un modo più elaborato. Assumendo in modo intuitivo il concetto di *complessità di una soluzione*, possiamo affermare il seguente principio:

Principio di complessità della soluzione. La complessità della soluzione di un problema diminuisce all'aumentare della potenza delle azioni che l'esecutore è in grado di compiere.

In base a questo principio possiamo dedurre che è molto più laborioso far eseguire una moltiplicazione fra due numeri naturali ad un esecutore che è capace solamente di addizionare un'unità alla volta che non ad un esecutore capace di addizionare due numeri qualsiansi. Il caso estremo è quando l'esecutore è capace di moltiplicare due numeri; in tale caso la soluzione del problema può essere espressa semplicemente in una forma del tipo *moltiplica x per y*. Il discorso può essere generalizzato come segue. Dato un problema P, l'ideale sarebbe disporre di un esecutore così potente da poter esprimere la soluzione mediante la sola istruzione *Risolvi il problema P*. Evidentemente, la realtà è ben diversa; inoltre, il procedimento risolutivo è tanto più lungo e complesso quanto più il linguaggio con il quale ci si rivolge all'esecutore è "lontano" dal problema trattato. Per diminuire la distanza fra il linguaggio usato e problema trattato, i linguaggi di programmazione offrono degli strumenti appositi: definizioni di sottoprogrammi, definizioni di tipi di dato, definizioni di classi ed altro ancora.

ESERCIZI

2.1 Spesso si usano con lo stesso significato le seguenti due locuzioni:

1. La MdT descritta dall'insieme di quintuple X
2. Il programma per una MdT costituito dall'insieme di quintuple X

Dire quale delle due forme sopra è preferibile. Dire perché, comunque, possono essere accettate entrambe le forme.

2.2 Spiegare perché le MdT costituiscono una conferma del *principio di complessità della soluzione*.

2.3 Il nastro di una MdT contiene un solo asterisco. Individuare una strategia per portarsi sull'asterisco trovandosi su uno spazio e non sapendo da quale parte rispetto all'asterisco. Scrivere un programma per una MdT corrispondente alla strategia individuata.

2.4 Decidere se un nastro è completamente vuoto. Si noti che, in questo caso, la MdT potrebbe non fermarsi.

Nota. Nei seguenti esercizi si richiede di scrivere dei programmi per una MdT, assumendo l'ipotesi che la macchina parta dallo stato iniziale 0 e sia posta sul primo carattere non-spazio a sinistra. Descrivere il procedimento dapprima a parole e successivamente codificarlo mediante un formalismo per le MdT.

2.5 Aggiungere un $*$ alla fine di una sequenza di $*$.

2.6 Spostare a destra di una posizione una sequenza di $*$.

2.7 Il nastro di una MdT contiene una sequenza contigua di asterischi. *Parificare* il numero di asterischi aggiungendone eventualmente uno nel caso in cui il nastro iniziale contenga un numero dispari di asterischi.

2.8 Calcolare in successione tutti i successivi del numero presente inizialmente sul nastro, senza fermarsi. Si risolva l'esercizio adottando le seguenti notazioni di codifica dei numeri: unaria, binaria e decimale. Risolvere l'esercizio adottando le seguenti notazioni di codifica dei numeri: unaria, binaria e decimale.

2.9 Calcolare la somma di due numeri naturali scritti in notazione unaria, secondo le due trasformazioni del nastro esemplificate a seguire:

1. $******** \rightarrow ******$
2. $******** \rightarrow ******$

2.10 Realizzare le seguenti funzioni sui numeri naturali, codificati in notazione unaria, in notazione binaria ed in notazione decimale:

successivo:	$n \mapsto n + 1$
precedente:	$n \mapsto 0$ se $n = 0$ altrimenti $n - 1$
doppio:	$n \mapsto n * 2$
metà:	$n \mapsto \lfloor n/2 \rfloor$
logaritmo:	$n \mapsto \lfloor \log_2 n \rfloor$

Discutere in quali casi e come la base di rappresentazione influenzi la facilità di scrittura di un programma per una MdT.

2.11 Assumendo come alfabeto del nastro l'insieme $A = \{\texttt{0}, \texttt{1}\}$ e l'ipotesi che il nastro contenga una sequenza compatta di caratteri, senza spazi, risolvere i seguenti problemi:

1. Eliminare tutti i caratteri presenti sul nastro.
2. Eliminare tutti gli `0` e compattare gli `1` verso sinistra.
3. Eliminare gli `0` iniziali e finali in una stringa binaria. Nel caso in cui la stringa sia composta da tutti `0` deve essere lasciato sul nastro un solo carattere `0`.
4. Spostare alla fine il carattere iniziale.
5. Spostare a destra di una posizione i caratteri presenti sul nastro.
6. Decidere se una stringa rappresenta un numero pari (in notazione binaria).
7. Decidere se una stringa è formata da un numero pari di `0`.
8. Decidere se una stringa è formata da simboli alternati; ad esempio: `0`, `10`, `01010`.
9. Decidere se una stringa è composta da un uguale numero di `0` ed `1`.
10. Eseguire l'addizione fra due numeri espressi in notazione binaria; ad esempio:

nastro prima della computazione:	`101+1100`
nastro alla fine della computazione:	`10001`

2.12 Decidere se un numero naturale è pari. Risolvere l'esercizio adottando le seguenti notazioni di codifica dei numeri: unaria, binaria e decimale.

2.13 Convertire un numero dalla notazione binaria a quella decimale.

2.14 Convertire un numero dalla notazione decimale a quella binaria.

2.15 Decidere se un numero naturale è pari. La MdT deve lasciare sul nastro la stringa com'era all'inizio della computazione fermandosi sullo stato Y se la stringa è composta da un numero pari di asterischi, nello stato N altrimenti. Risolvere l'esercizio adottando le seguenti notazioni di codifica dei numeri: unaria, binaria e decimale. Discutere quale notazione risulta più comoda per la scrittura del programma.

2.16 Decidere se una stringa di `+` e `-` è palindroma, ossia se si legge indifferentemente da sinistra a destra e da destra a sinistra. La MdT deve lasciare sul nastro la stringa com'era all'inizio della computazione; deve fermarsi sullo stato Y se la stringa è palindroma, nello stato N altrimenti.

2.17 Il nastro di una MdT contiene una sequenza contigua composta da `a` e `b`. Decidere se la sequenza è composta da almeno due caratteri consecutivi uguali.

2.18 Decidere se una data stringa è formata da caratteri `a` e `b` alternati. Ad esempio, devono essere riconosciute le seguenti stringhe: `a`, `ba`, `ababa`, `bababa`.

2.19 Decidere se una data stringa è formata da una successione di `a` seguite da una successione di `b`. Ad esempio, devono essere riconosciute le seguenti stringhe: `ab`, `aabbbb`.

2.20 Decidere se una data stringa è formata da una successione di `a` seguite da una successione di `b`, in uguale numero. Ad esempio, devono essere riconosciute le seguenti stringhe: `ab`, `aabb`, `aaabbb`.

2.21 Il nastro di una MdT contiene una stringa di caratteri `a` o `b` . Determinare l'ultimo carattere, eliminando dal nastro tutti i caratteri escluso l'ultimo. Descrivere dapprima il procedimento a parole e successivamente codificarlo mediante un insieme di quintuple e mediante un grafo di transizione di stato.

2.22 Decidere se una sequenza di parentesi è bilanciata a forma di buccia di cipolla, ossia, ad esempio, della forma `((()))`. Generalizzare al caso di parentesi bilanciate generiche, ossia della forma `(()())()`.

2.23 Usando 3 stati (oltre allo stato finale) ed il solo simbolo `*`, descrivere una MdT che, partendo da un nastro vuoto, scriva il maggior numero possibile di asterischi e poi si fermi. Una MdT come quella appena descritta viene detta *castoro a 3 stati*. Un castoro massimale che genera il maggior numero di asterischi viene detto *alacre castoro*.

2.24 Date le seguenti tre MdT:

$\mathcal{M}_1$ che trasforma un numero da notazione unaria a notazione decimale

$\mathcal{M}_2$ che trasforma un numero da notazione decimale a notazione unaria

$\mathcal{M}_3$ che calcola il massimo comune divisore fra due numeri naturali espressi in notazione unaria

comporle opportunamente in modo da ottenere una MdT $\mathcal{M}$ che calcola il massimo comune divisore fra due numeri naturali espressi in notazione decimale.

2.25 Approfondire e discutere la seguente architettura (di *von Neuman*) di macchina di Turing (*macchina di Turing Universale*):

La MdT opera secondo un prefissato programma (non modificabile); la particolarizzazione del procedimento (per adattarsi alla soluzione di diverse classi di problemi) avviene precisando (come dato di input) sul nastro sia l'insieme delle quintuple (che permettono di risolvere i problemi di una classe di problemi) che (come succede per le MdT viste finora) i dati di input per risolvere una particolare istanza di problema (della classe considerata).

2.26 Oltre alle macchine di Turing sono stati ideati altri meccanismi di calcolo. La macchina UMR (ideata nel 1963 da Shepherdson e Sturgis e successivamente modificata da Cutland nel 1980) è composta da un numero illimitato di registri, indicati con $R_1, R_2, \ldots$. Ogni registro, in un dato istante, contiene un numero naturale. Il contenuto dei vari registri può essere modificato mediante le seguenti istruzioni:

- Istruzione di *azzeramento Z(n)* : viene posto uguale a zero il contenuto del registro R_n
- Istruzione di *successivo S(n)* : viene aumentato di uno il contenuto del registro R_n
- Istruzione di *trasferimento T(m,n)* : viene sostituito il valore del registro R_m con il valore del registro R_n

- Istruzione di *salto J(m,n,p)* : se il contenuto del registro R_m è uguale a quello del registro R_n la macchina va alla p-esima istruzione del programma

Discutere comparativamente i livelli di proceduralità e dichiaratività delle MdT e delle macchine URM. Scrivere programma per una macchina URM che esegua il prodotto fra due numeri naturali memorizzati nei registri R_1 ed R_2; il risultato deve essere memorizzato nel registro R_3.

2.27 Individuare gli ingressi, le uscite e l'eventuale stato di una macchina automatica che cambia monete.

2.28 Determinare le informazioni interne all'automa tartaruga che ne costituiscono lo stato.

2.29 Definire una classe di problemi corrispondente all'istanza di figura 2.17 disegnata mediante l'algoritmo 3. Scrivere un algoritmo per disegnare una generica figura della classe.

2.30 Determinare, senza provare al computer, la figura che viene disegnata mediante algoritmo 3 mettendo 1 al posto di 3 nell'istruzione 2.

2.31 Descrivere la struttura di una macchina addizionatrice a 4 ingressi.

2.32 Descrivere la struttura di una macchina a 4 ingressi in grado di calcolare il valore di un'espressione della forma $a + b - c - d$.

3

ALGORITMI

Per troppo tempo, la gente ha tentato di formulare gli algoritmi in un linguaggio di programmazione, piuttosto che decomporre attentamente il problema in problemi più semplici e verificare le loro relazioni prima di procedere ad una specificazione più dettagliata. [...] L'esperienza ha dimostrato che [il secondo approccio] permette di sviluppare algoritmi con molti meno errori che non il vecchio metodo linea-per-linea.

S.Alagic, M.A.Arbib,
The design of well-structured and correct programs

Nelle pagine seguenti sarà definita e precisata la nozione di procedimento risolutivo di un problema, in modo che sia adattabile ai calcolatori. Sarà presentato il concetto di *algoritmo* che rappresenta un'evoluzione in senso dinamico del concetto di *formula risolutiva* della matematica.

I primi algoritmi furono ideati per risolvere problemi di tipo numerico: moltiplicazione e divisione fra due numeri, massimo comune divisore fra due numeri naturali. Oggigiorno gli algoritmi vengono applicati per risolvere problemi di varia natura ed operano sulle più svariate tipologie di entità.

3.1 Algoritmi

La parola "algoritmo" deriva dal nome del matematico persiano Abu Ja'far Muhammad ibn Musa *al-Khwarizmi* che nell'anno 825 d.C. scrisse un importante ed influente testo di matematica. Dalla storpiatura latina medievale del suo nome derivò il termine "algorismus" e successivamente la parola moderna "algoritmo". Nel medioevo il termine indicava un procedimento di calcolo, basato sull'uso delle cifre arabe, descritto mediante un numero finito di regole esplicite che conducono al risultato finale dopo un numero finito di applicazioni delle regole. Con lo sviluppo dei calcolatori il termine algoritmo è stato utilizzato per indicare la descrizione di una sequenza di istruzioni che, eseguite da parte di un sistema di calcolo, permette di risolvere un problema. Più in generale e più precisamente si assume la seguente definizione.

DEFINIZIONE 1. Un *algoritmo* è la descrizione della soluzione di una classe di problemi, ossia la descrizione delle azioni, da eseguirsi da parte di un dato esecutore, mediante le quali risolvere una classe di problemi; inoltre, un algoritmo deve soddisfare alle seguenti condizioni:

- *definitezza*: ogni azione deve essere deterministica ed univocamente interpretabile dall'esecutore.
- *eseguibilità*: ogni azione deve essere effettivamente eseguibile da parte dell'esecutore.
- *finitezza*: le istruzioni devono essere in numero finito, ciascuna azione deve essere eseguita in un tempo finito ed ogni azione deve essere eseguita un numero finito di volte.
- *generalità*: la descrizione del procedimento risolutivo deve applicarsi a tutte le istanze della classe dei problemi.

Le stringenti condizioni elencate nella precedente definizione differenziano quelli che genericamente sono dei *procedimenti risolutivi* da quei procedimenti che possono essere classificati come *algoritmi*. Ad esempio, una ricetta di cucina può essere classificata come un *procedimento risolutivo* ma non come un *algoritmo*. Comunque, quando nel seguito si utilizzerà il termine *procedimento risolutivo*, è da intendersi con significato di *algoritmo*, secondo la definizione data sopra.

Questa definizione di algoritmo è una fra le tante possibili. Tutte le altre che si possono trovare sui testi di informatica, anche se apparentemente diverse ed espresse in modo più formale (ad esempio, macchina di Turing, sistema canonico di Post, lambda-calcolo di Church) sono equivalenti a questa.

Osserviamo che, con le condizioni di generalità e finitezza, si impone che un algoritmo termini per ogni istanziazione dei dati di ingresso. Nella definizione di algoritmo si impone che le istruzioni devono essere eseguite un numero infinito di volte; in molti casi questa condizione di finitezza risulta troppo forte e, per poter trattare una classe più ampia di procedimenti risolutivi, risulta conveniente alleggerire la condizione di finitezza, ammettendo che delle istruzioni possano essere eseguite per un numero infinito di volte. In tali condizioni un procedimento risolutivo è detto *metodo computazionale*.

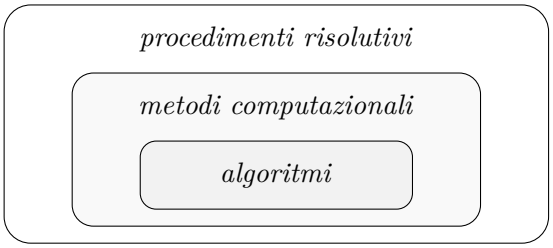


Figura 3.1: Algoritmi, metodi computazionali e procedimenti risolutivi.

La nozione di algoritmo è strettamente connessa con la nozione di esecutore: basta considerare i termini *univocamente interpretabile* e *realmente eseguibile*. È evidente quindi che un algoritmo presuppone che esista un esecutore che lo interpreti ed esegua le istruzioni specificate.

Anche se la finalità di un algoritmo è quella di essere eseguito da un calcolatore e necessita pertanto di essere tradotto in uno specifico linguaggio di programmazione, un algoritmo è indipendente dai linguaggi di programmazione nei quali verrà tradotto.

3.2 Livelli descrittivi degli algoritmi

Gli algoritmi vengono descritti con un linguaggio e con livello di dettaglio delle azioni da eseguire che dipendono dalle capacità dell'esecutore. Questo fatto è evidenziato nella soluzione dei due problemi che seguono

Problema 3.2.1 Note le misure a e b dei due lati di un rettangolo, determinare la misura d della diagonale.

Soluzione. Se l'esecutore è uno studente che abbia frequentato almeno le scuole secondarie inferiori, un procedimento risolutivo potrebbe essere:

Determina d usando il teorema di Pitagora.

Se l'esecutore non conosce il teorema di Pitagora ma conosce la notazione algebrica, un procedimento risolutivo, espresso in modo algebrico, potrebbe essere:

$$d \leftarrow \sqrt{a^2 + b^2}$$

Se l'esecutore non conosce nemmeno il simbolismo algebrico che compare nella formula precedente o non è in grado di eseguire le operazioni indicate, il procedimento risolutivo dovrebbe essere descritto ad un livello ancora più elementare. Ad esempio, se l'esecutore è in grado di eseguire singolarmente le quattro operazioni aritmetiche elementari ed il calcolo della radice quadrata si usa una notazione algoritmica come la seguente, dove vengono indicati i dati sui quali opera l'algoritmo ed il risultato fornito, come descritto nell'algoritmo 1.

Algoritmo 1 - Diagonale di un rettangolo

Input: misure a e b dei lati del rettangolo
Output: diagonale d del rettangolo

1: $p \leftarrow a * a$
2: $q \leftarrow b * b$
3: $s \leftarrow p + q$
4: $d \leftarrow \sqrt{s}$
5: **return** d

L'istruzione **return** indica il risultato del calcolo. Questo semplice problema evidenzia che una stessa soluzione può essere espressa con notazioni ed a livelli descrittivi diversi, in funzione delle capacità e delle conoscenze dell'esecutore al quale è rivolta. □

Problema 3.2.2 Determinare l'area a di un trapezio date le misure B della base maggiore, b della base minore ed h dell'altezza.

Soluzione. Per questo problema il procedimento risolutivo è immediato e può essere espresso per mezzo della semplice formula

$$a \leftarrow (B + b)h/2$$

Anche qui, comunque, la formula risolutiva maschera un procedimento risolutivo che può essere esplicitato in modo discorsivo come segue:

> *Considera i valori delle misure delle basi e dell'altezza; somma le misure delle due basi; moltiplica il risultato per l'altezza; dividi il risultato per* 2*; il risultato ottenuto rappresenta l'area del trapezio.*

Il precedente procedimento viene solitamente scritto in una notazione più precisa, come descritto nell'algoritmo 2.

Algoritmo 2 - Area di un trapezio

Input: misure B e b delle basi ed h dell'altezza
Output: area a del trapezio

1: $s \leftarrow B + b$
2: $d \leftarrow s * h$
3: $a \leftarrow d/2$
4: **return** a

□

Osservazione. Dai due precedenti problemi 3.2.1 e 3.2.2 sembra che gli algoritmi corrispondano ad una singola formula risolutiva e sono stati esplicitati mediante un'elencazione sequenziale di passi. Per molti problemi, anche limitandosi ai problemi di tipo computazionale, la soluzione non è così semplice e lineare; servono, in generale, dei costrutti linguistici più articolati. In altre

parole, non esiste più una corrispondenza biunivoca fra le frasi della descrizione delle istruzioni e l'esecuzione delle istruzioni. Per convincersi di quanto appena affermato consideriamo il seguente altro problema:

> *Determinare il massimo comune divisore di due numeri naturali m ed n, ossia il più grande numero naturale che divide sia m che n.*

In questo caso il risultato non è direttamente ricavabile mediante una singola formula risolutiva e neanche come sequenza di passi, come nei problemi precedenti, ma necessita di un algoritmo più articolato. Questo problema sarà ripreso e risolto in più modi nel seguito.

3.3 Schemi di algoritmi

Le azioni descritte in un algoritmo vengono strutturate secondo degli schemi (detti *controlli*). I formati più frequenti sono descritti a seguire.

Controllo sequenziale

Per indicare che le azioni devono essere eseguite una dopo l'altra, dalla prima all'ultima, è sufficiente scriverle sequenzialmente, nell'ordine desiderato.

Esempio 3.3.1 - Il seguente algoritmo, rivolto ad un esecutore in grado di eseguire le operazioni aritmetiche singolarmente, determina la superficie di un parallelepipedo, date le misure dei 3 spigoli.

Algoritmo 3 - Superficie di un parallelepipedo

Input: misure a, b e c degli spigoli del parallelepipedo
Output: superficie del parallelepipedo

1: $p \leftarrow a + b$
2: $p \leftarrow 2 * p$
3: $q \leftarrow a * b$
4: $p \leftarrow p * c$
5: $q \leftarrow 2 * q$
6: $s \leftarrow p + q$
7: **return** s

Controllo condizionale

Spesso accade che, a seconda del valore di verità di una condizione, debbano essere svolte delle azioni oppure altre; similmente, sempre a seconda del valore di verità di una condizione, accade che certe azioni debbano essere eseguite o meno. In questi casi si ricorre al controllo condizionale, come descritto nel seguente esempio.

Esempio 3.3.2 - Il seguente algoritmo determina il massimo fra tre numeri.

Algoritmo 4 - Massimo fra 3 numeri

Input: numeri a, b e c
Output: massimo dei 3 numeri
1: **if** $a > b$ **then**
2: $max \leftarrow a$
3: **else**
4: $max \leftarrow b$
5: **end if**
6: **if** $c > max$ **then**
7: $max \leftarrow c$
8: **end if**
9: **return** max

Controllo ciclico

In molte situazioni serve ripetere delle azioni in modo ciclico. Lo schema più semplice consiste nel ripetere delle azioni per un numero di volte noto a priori.

Esempio 3.3.3 - Il seguente algoritmo disegna un quadrato di dato lato utilizzando l'automa tartaruga.

Algoritmo 5 - Disegno di un quadrato

Input: lunghezza a del quadrato da disegnare
1: **for** 4 **times**
2: $forward(a)$
3: $left(90)$
4: **end for**

Si incontrano spesso situazioni in cui serve eseguire ciclicamente delle azioni in un numero di volte non predeterminato, ma dipendente da condizioni che si modificano dinamicamente durante il processo di elaborazione.

Esempio 3.3.4 - Il seguente algoritmo, rivolto ad un esecutore in grado di eseguire le operazioni aritmetiche sui numeri naturali, determina il numero di cifre (in base 10) di un numero naturale; ad esempio il numero di cifre del numero 74262 è 5.

Algoritmo 6 - Numero di cifre di un numero naturale

Input: numero naturale n
Output: numero di cifre di n
1: $k \leftarrow 0$
2: **while** $n \neq 0$ **do**
3: $n \leftarrow n \,\mathsf{div}\, 10$
4: $k \leftarrow k + 1$
5: **end while**
6: **return** k

3.4 Moltiplicare due numeri

Consideriamo il problema della moltiplicazione fra due numeri naturali. Quando noi, con carta e penna, eseguiamo una moltiplicazione fra due numeri non stiamo risolvendo un problema ma stiamo semplicemente applicando, in veste di esecutori, il procedimento che ci ha insegnato la maestra alle scuole elementari. Le conoscenze operative richieste da questo procedimento consistono nella conoscenza delle tabelline fino al 9 e di essere in grado di sommare dei numeri scritti in colonna. Queste operazioni risultano di facile applicabilità per le persone che riescono a destreggiarsi con disinvoltura su un foglio a quadretti gestendo l'incolonnamento dei numeri, ma di difficile traduzione per un calcolatore.

Benché il termine "algoritmo" sia balzato alla ribalta solo negli ultimi decenni, la sua storia è millenaria. Esempi di algoritmi risalgono alle epoche degli egizi e dei babilonesi. Presso il British Museum di Londra è conservato un antico papiro, datato attorno al 1650 a.C., detto *papiro di Rhind* (dal nome dell'antiquario scozzese che, a metà del XIX secolo, lo acquistò in Egitto e lo donò al museo) o *papiro di Ahmes* (dal nome dello scriba che lo redasse), che contiene molti problemi di tipo geometrico ed aritmetico e molte regole e procedimenti di calcolo, derivati da una lunga tradizione risalente al XX secolo a.C.. Fra questi è descritto un ingegnoso metodo per moltiplicare due numeri naturali. Il procedimento di calcolo consiste nel raddoppiare iterativamente un fattore mentre si dimezza l'altro, tenendo conto dove il valore del primo fattore è dispari. Il passo finale consiste nell'addizionare i multipli del fattore che è stato raddoppiato, in corrispondenza dei punti in cui il valore del primo fattore è dispari. Ad esempio, l'applicazione di questo algoritmo per calcolare il prodotto fra i due numeri $m = 22$ e $n = 13$ è descritto nella tabella riportata nella figura 3.2, dove i due numeri in grassetto alla seconda riga (22 e 13) sono i dati, i numeri sottolineati corrispondono alla condizione *m dispari* e costituiscono gli addendi che, sommati, producono il risultato finale 286. La correttezza del procedimento è facilmente dimostrabile osservando che

$$m * n = \begin{cases} (m \div 2) * (2 * n) & \text{se } m \text{ è pari} \\ (m-1) * n + n = (m \div 2) * (2 * n) + n & \text{se } m \text{ è dispari} \end{cases}$$

m	m dispari ?	n
22	no	**13**
11	si	<u>26</u>
5	si	<u>52</u>
2	no	104
1	si	<u>208</u>
	prodotto:	**286**

Figura 3.2: Tavola di traccia del calcolo 22×13 mediante l'algoritmo di moltiplicazione egizio.

Le fasi operative per l'applicazione di questo procedimento sono (specialmente se i fattori sono grandi) più laboriose rispetto all'usuale metodo di moltiplicazione, ma in questo procedimento le capacità coinvolte sono più elementari. Un esecutore deve infatti essere in grado di eseguire le seguenti operazioni:

- duplicare un numero naturale
- dimezzare un numero naturale
- decidere se un numero naturale è dispari
- addizionare numeri naturali

Notiamo inoltre che questo *procedimento* di moltiplicazione non è commutativo e che, poiché *l'operazione* di moltiplicazione è commutativa, in un prodotto è preferibile prendere come fattore da dimezzare quello minore, per ottimizzare l'efficienza del procedimento, ossia per ridurre il numero di operazioni che vengono eseguite nel processo di calcolo; e questo numero è proporzionale al numero di righe della tavola di traccia.

Una tabella come quella descritta nella figura 3.2 è detta *tavola di traccia*; non costituisce la soluzione del problema in quanto corrisponde all'algoritmo applicato ad un'istanza di problema e descrive nel tempo i valori associati alle entità coinvolte nell'algoritmo. Una tavola di traccia è paragonabile ad una sequenza di fotogrammi che descrive l'avanzamento del processo di elaborazione.

Ricorrendo agli schemi di algoritmi descritti nel precedente paragrafo, dalla tavola di traccia riportata nella figura 3.2 si può derivare l'algoritmo 7.

Algoritmo 7 - Moltiplicazione di due numeri naturali (algoritmo egizio)

Input: numeri naturali m e n
Output: prodotto fra m e n

```
1: inizializza p a 0
2: while m ≠ 0 do
3:    if m è dispari then
4:       incrementa p di n
5:    end if
6:    dimezza m
7:    raddoppia n
8: end while
9: return p
```

Il procedimento di moltiplicazione egizio è rimasto in uso in Russia ed in alcuni paesi confinanti fino a tempi molto recenti in virtù del fatto che in questi paesi venivano utilizzati abaci con i quali era facile eseguire le operazioni di divisione e moltiplicazione per 2; per questo motivo il procedimento è noto anche con il nome di *algoritmo del contadino russo*. L'algoritmo viene ancora oggi utilizzato nelle unità aritmetico logiche degli attuali computer dove, per

la particolare architettura operativa e per la rappresentazione dei numeri in notazione binaria, le operazioni di moltiplicazione e divisione per due risultano particolarmente efficienti in quanto svolte mediante uno slittamento di un posto delle cifre del numero. È interessante notare questo filo logico che lega un procedimento ideato quattro millenni fa ed un aspetto della tecnologia di oggigiorno.

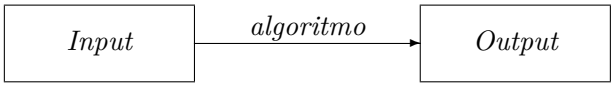
3.5 Algoritmi come funzioni

Risolvere un problema significa sostanzialmente definire una funzione che opera sui dati iniziali di input e produce dei risultati finali di output. Ad esempio, nel caso del problema dell'area di un trapezio esaminato nel precedente paragrafo, la soluzione può essere espressa mediante l'espressione funzionale

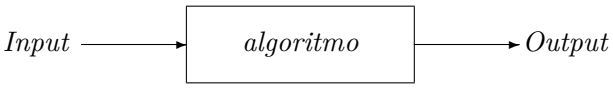
$$area = f(B, b, h) = (B + b)\, h/2$$

Anche quando l'algoritmo non sia esprimibile mediante una singola espressione, esso può sempre comunque essere considerato come una funzione che, operando sui dati di input, genera i risultati di output (in questo caso è solo più laboriosa la descrizione e la valutazione della funzione).

Adottando la tradizionale convenzione di denotare con il termine *Input* l'insieme dei dati e con il termine *Output* l'insieme dei risultati, descriveremo un algoritmo mediante uno schema a blocchi come il seguente, dove viene sottolineato il ruolo dell'algoritmo che stabilisce l'associazione fra dati e risultati; in questo caso l'algoritmo viene visto come una funzione che, in base ai dati iniziali di input, genera i risultati finali di output.



Nel caso si voglia mettere in evidenza che l'algoritmo si comporta come una macchina che trasforma i dati che riceve in ingresso nei risultati in uscita si ricorre ad una figura come la seguente:



L'interpretazione di un algoritmo come funzione porta a delle naturali definizioni e concetti.

DEFINIZIONE 2. Due algoritmi A_1 ed A_2 si dicono *funzionalmente equivalenti* se hanno lo stesso dominio e calcolano lo stesso risultato y per ogni possibile istanziazione dei dati di input x, ossia

$$A_1(x) = A_2(x) = y \text{ per ogni input } x$$

Due algoritmi si dicono *equivalenti* se fanno compiere all'esecutore la stessa sequenza di azioni. È evidente che l'equivalenza implica l'equivalenza funzionale.

Data la definizione di equivalenza fra algoritmi, acquista importanza pratica il seguente problema: *Decidere se due dati algoritmi sono equivalenti.* La soluzione di questo problema dovrebbe essere un algoritmo che, dati due algoritmi come suoi dati di input, decida se questi algoritmi sono equivalenti. È stato dimostrato che un tale algoritmo di decisione non esiste.

3.6 Analisi e complessità degli algoritmi

L'analisi e la valutazione della qualità degli algoritmi in relazione alla quantità di risorse impiegate viene detta *complessità computazionale.* I criteri in base ai quali stabilire la qualità di un algoritmo non vengono basati sulle caratteristiche intrinseche dell'algoritmo (lunghezza, leggibilità) bensì sulle risorse richieste dall'algoritmo (tempo impiegato, memoria occupata). Fra questi il fattore più significativo è costituito dal tempo di elaborazione impiegato per l'esecuzione dell'algoritmo. Il parametro *tempo* risulta però poco interessante in quanto fa riferimento ad una particolare esecuzione di una istanza del problema su una data macchina. È chiaramente poco significativa, infatti, una frase del tipo: *L'algoritmo di ordinamento A impiega* 6.2 *secondi ad essere eseguito* in quanto non è precisato né il numero di elementi che vengono ordinati, né la loro disposizione iniziale, né il linguaggio di programmazione usato per codificare l'algoritmo in un programma, né la macchina sulla quale viene eseguito il programma. Questo approccio richiederebbe sempre delle prove per poter confrontare due algoritmi. Il problema della valutazione della complessità degli algoritmi non ammette una soluzione di tipo pragmatico della forma "prova e vedi" ma deve essere affrontato sulla carta; in questo modo si riesce a valutare la qualità di un algoritmo indipendentemente dal sistema di elaborazione, anche nei casi in cui il tempo di esecuzione è elevato (un'ora o addirittura un secolo!). Ed è proprio in questi casi estremi che è utile sapere (senza provare!) la complessità di un algoritmo per stimare il tempo di esecuzione, senza eseguire l'algoritmo. Per stimare il tempo di esecuzione di un algoritmo senza fare riferimento ad alcuna macchina particolare si assume l'ipotesi (plausibile) che il tempo di esecuzione di un imprecisato numero k di operazioni elementari sia proporzionale a k e che il numero k sia una funzione f crescente della misura n dei dati da elaborare. Pertanto, tale funzione $f(n)$ può essere assunta come misura del tempo impiegato. Il problema viene così ricondotto al conteggio dei passi elementari che vengono svolti nell'algoritmo; questo numero rappresenta indirettamente, senza riferirsi ad una particolare macchina, una stima del tempo di elaborazione impiegato per l'esecuzione dell'algoritmo. A seguire sono descritti i concetti cardine mediante i quali si definisce in modo preciso la complessità.

Operazione dominante

Spesso negli algoritmi si incontrano dei passi elementari di diversa natura e computazionalmente non uniformabili; ci possono essere delle assegnazioni, delle addizioni, delle moltiplicazioni, delle valutazioni di condizioni. Per semplificare la valutazione della complessità si prende in considerazione l'operazione che più incide sul tempo di elaborazione, come specificato di seguito.

DEFINIZIONE 3. Si chiama *operazione dominante* di un algoritmo l'operazione che viene eseguita con maggiore frequenza; nel caso di operazioni eseguite con la stessa frequenza si considera quella più onerosa in termini di tempo impiegato per la sua esecuzione.

In molti casi l'operazione dominante è deducibile dal problema in quanto tutti gli algoritmi che lo risolvono avranno questa come operazione dominante.

Esempio 3.6.1 - Nella tabella che segue sono riportati alcuni problemi e la corrispondente operazione dominante.

Problema	Operazione dominante
calcolo del fattoriale di un numero	moltiplicazione fra due numeri
ricerca in una sequenza	confronto fra due elementi
ordinamento di una sequenza	confronto fra due elementi
moltiplicazione fra due matrici	moltiplicazione fra due numeri
torre di Hanoi	spostamento di un disco

Tabella 3.1: Problemi e corrispondenti operazioni dominanti.

Dimensione di un problema

Per un dato algoritmo il numero di passi elementari che vengono eseguiti dipende quasi sempre dalla quantità e dalla grandezza dei dati coinvolti nella specifica istanza di problema che si considera. Per tener conto di questo parametro si fa riferimento alla seguente definizione.

DEFINIZIONE 4. La *dimensione di un problema* è un numero naturale che esprime un'opportuna misura dei dati di input del problema.

Nonostante l'apparente vaghezza del termine *opportuna* che compare nella definizione precedente, la dimensione di un problema è facilmente identificabile, come riportato nell'esempio che segue.

Esempio 3.6.2 - La tabella 3.2 descrive la dimensione di alcuni problemi.

Problema	Dimensione del problema
calcolo del fattoriale di un numero	numero di cui si calcola il fattoriale
ricerca in una sequenza	numero di elementi della sequenza
ordinamento di una sequenza	numero di elementi della sequenza
moltiplicazione fra due matrici	dimensione delle matrici
torre di Hanoi	numero di dischi da spostare

Tabella 3.2: Problemi e corrispondenti dimensioni.

Complessità di un algoritmo

In funzione dell'operazione dominante e della dimensione di un problema si definisce la complessità di un algoritmo.

DEFINIZIONE 5. La *complessità di un algoritmo* è una funzione che esprime il numero di operazioni dominanti in funzione della dimensione del problema.

Esempio 3.6.3 - Il problema della moltiplicazione fra due matrici quadrate di lato n ha dimensione n. L'operazione dominante è la moltiplicazione (l'addizione non viene considerata in quanto è computazionalmente meno onerosa in tempo rispetto alla moltiplicazione). Il tradizionale algoritmo di moltiplicazione *righe-per-colonne* ha complessità n^3 in quanto richiede l'esecuzione di n moltiplicazioni per il calcolo di ciascuno degli n^2 elementi della matrice risultante.

Spesso la precisazione dell'operazione dominante e della dimensione del problema sono sottaciuti in quanto sono deducibili dalla formulazione del problema stesso.

Esempio 3.6.4 - Affermare che un algoritmo di ordinamento ha complessità pari a n^2 significa che l'algoritmo svolge un numero di confronti proporzionale a n^2 per ordinare una sequenza di n elementi.

Complessità di un problema

Basandosi sul concetto di complessità di un algoritmo si definisce la complessità di un problema, come segue.

DEFINIZIONE 6. La *complessità di un problema* è la complessità del miglior algoritmo che lo risolve.

Esempio 3.6.5 - Il problema della torre di Hanoi per lo spostamento di una pila di n dischi ha dimensione n. L'operazione dominante è data dallo spostamento di 1 disco da un piolo ad un altro. Il miglior algoritmo che risolve il problema richiede lo spostamento di $2^n - 1$ dischi. Pertanto la complessità del problema è $2^n - 1$.

Esempio 3.6.6 - Il problema dell'ordinamento di una sequenza di dimensione n ha complessità pari a $n \log_2 n$, in quanto esistono algoritmi di ordinamento che hanno una tale complessità, ossia che richiedono $n \log_2 n$ confronti per ordinare una sequenza di n elementi; d'altra parte si può dimostrare che, in generale, non si può ordinare una sequenza con meno di $n \log_2 n$ confronti.

3.7 Complessità asintotica

Poiché la complessità di un algoritmo risulta importante e decisiva per valori grandi della dimensione del problema, si danno le seguenti definizioni.

DEFINIZIONE 7. Date due funzioni f e g, reali a valori reali, si dice che la funzione f *è O grande* di g, e si scrive f è $O(g)$ oppure $f \in O(g)$, se esiste una costante reale $c > 0$ ed un numero reale x_0 tale che per $x \geq x_0$ si abbia $f(x) \leq c\,g(x)$.

Esempio 3.7.1 - Con la notazione O grande una funzione della forma $f(n)$ definita come somma di più addendi risulta caratterizzata dall'addendo che per

valori grandi di n risulta prevalente, trascurando eventuali coefficienti moltiplicativi degli addendi. Ad esempio, la funzione $f(n) = 3n^5 + 72n^3 + 10^6$ è $O(n^5)$; la funzione $f(n) = 4n^2 + 100n \log_2 n$ è $O(n^2)$; la funzione $f(n) = 2^n + n^{100}$ è $O(2^n)$.

DEFINIZIONE 8. La *complessità asintotica di un algoritmo o di un problema* è la complessità per valori grandi della dimensione del problema; viene denotata con la notazione O (*o grande*).

Il tecnicismo matematico della notazione O *grande* può essere evitato accontentandosi di una riformulazione più approssimativa; ad esempio la frase *L'algoritmo A ha una complessità asintotica* $O(f(n))$. significa che per valori grandi della dimensione n del problema il numero di passi elementari richiesti dall'algoritmo è proporzionale a $f(n)$.

Esempio 3.7.2 - L'affermazione: "*Il problema della Torre di Hanoi ha una complessità computazionale asintotica pari a* $O(2^n)$. significa che per spostare n dischi da un piolo ad un altro, per n grande, si esegue un numero di spostamenti proporzionale a 2^n.

Esempio 3.7.3 - La seguente tabella riporta una lista di problemi e le loro corrispondenti complessità asintotica.

Problema	Complessità asintotica del problema
calcolo del fattoriale di un numero	$O(n)$
ricerca in una sequenza	$O(n)$
ordinamento di una sequenza	$O(n \log n)$
moltiplicazione fra due matrici	non è nota
torre di Hanoi	$O(2^n)$

Esempio 3.7.4 - Il problema della moltiplicazione di due matrici quadrate di ordine n ha una complessità non nota. Il tradizionale algoritmo di moltiplicazione *righe-per-colonne* ha una complessità asintotica pari a n^3; il migliore algoritmo conosciuto ha una complessità asintotica pari a $n^{2,78}$ ma attualmente non si conosce una limitazione inferiore per la complessità asintotica degli algoritmi di moltiplicazione di matrici.

Problema 3.7.1 Calcolare la complessità del seguente schema di algoritmo, dove n denota la *dimensione del problema* ed *op* l'*operazione dominante*.

Soluzione. La complessità è data dal numero di operazioni *op* che vengono eseguite dall'algoritmo ed è pari a

$$n(2n + 1) + 3n = 2n^2 + 4n$$

Di conseguenza la complessità asintotica è pari a

$$O(n^2)$$

□

```
1: for n times
2:    for 2n times
3:       op
4:    end for
5:    op
6: end for
7: for 3n times
8:    op
9: end for
```

3.8 Problemi facili e difficili

Un problema viene classificato come facile o difficile non in base alla difficoltà che incontra il solutore per escogitare una soluzione, bensì in base alla complessità del problema, ossia in base alla complessità del miglior algoritmo che lo risolve. Nel contesto della complessità computazionale i problemi facili vengono detti *trattabili* mentre i problemi difficili vengono detti *intrattabili*. I risultati dei problemi trattabili si ottengono in tempi ragionevoli mentre i problemi intrattabili superano la potenza di calcolo di ogni calcolatore esistente e futuro. La linea di demarcazione fra problemi trattabili e problemi intrattabili dipende dalla tipologia della funzione di complessità: i problemi trattabili sono caratterizzati da funzioni di complessità polinomiali, mentre i problemi intrattabili hanno funzioni di complessità di categoria esponenziale. La figura 3.3 dispone in linea di complessità crescenti un insieme di funzioni che si incontrano spesso nell'analisi degli algoritmi e definisce una linea di separazione fra problemi trattabili ed intrattabili. Esistono problemi per i quali non è attualmente noto a quale di queste due categorie appartengono.

complessità polinomiali							complessità esponenziali		
$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	n^{10}	2^n	$n!$	n^n

Figura 3.3: Complessità polinomiali ed esponenziali di alcune funzioni che si incontrano frequentemente nell'analisi degli algoritmi.

Nella tabella 3.3 è riportato il tempo in secondi che impiega un calcolatore che esegue 1 milione di operazioni al secondo a risolvere un problema di dimensione n. L'analisi dei dati riportati nella tabella motiva che è coerente designare come *polinomiali* i problemi trattabili e come *esponenziali* i problemi intrattabili.

$f(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	50	$n = 100$
$\log_2 n$	3.3 μs	4.3 μs	4.9 μs	5.3 μs	5.6 μs	6.6 μs
$\sqrt{n}$	3.1 μs	4.5 μs	5.5 μs	6.3 μs	7.0 μs	10 μs
n	10 μs	20 μs	30 μs	40 μs	50 μs	100 μs
$n \log_2 n$	33 μs	86 μs	147 μs	213 μs	282 μs	664 μs
n^2	100 μs	400 μs	900 μs	1600 μs	2500 μs	10000 μs
n^3	1 ms	8 ms	27 ms	64 ms	125 ms	1 sec
2^n	1 ms	1 sec	17 min	12 g	35 a	$4 \cdot 10^{16}$ a
$n!$	3.6 sec	$8 \cdot 10^4$ a	$8 \cdot 10^{18}$ a	$3 \cdot 10^{34}$ a	$9 \cdot 10^{50}$ a	$3 \cdot 10^{154}$ a
n^n	2.8 h	$3 \cdot 10^{12}$ a	$6 \cdot 10^{35}$ a	$4 \cdot 10^{50}$ a	$3 \cdot 10^{71}$ a	$3 \cdot 10^{186}$ a

Tabella 3.3: Tempi delle funzioni di complessità di alcune funzioni. Le unità di misura dei tempi sono: μs = microsecondi, ms = millisecondi, sec = secondi, min = minuti, g = giorni, a = anni.

Nella tabella 3.4 è riportata la dimensione del problema risolvibile in una data quantità di tempo, sempre nell'ipotesi di disporre di un calcolatore che esegue 1 milione di operazioni al secondo. Un'analisi, anche superficiale, di questa tabella evidenzia che l'evoluzione tecnologica dei calcolatori non potrà intaccare il carattere di intrattabilità dei problemi.

$f(n)$	1 sec	1 min	1 ora	1 $giorno$	1 $mese$	1 $anno$	1 $secolo$
$\log_2 n$	10^{400000}	10^{10^7}	10^{10^9}	$10^{10^{10}}$	$10^{10^{12}}$	$10^{10^{13}}$	$10^{10^{15}}$
$\sqrt{n}$	10^{12}	$4 \cdot 10^{15}$	10^{19}	$7 \cdot 10^{21}$	$7 \cdot 10^{24}$	10^{27}	10^{31}
n	10^6	$6 \cdot 10^7$	$4 \cdot 10^9$	$9 \cdot 10^{10}$	$3 \cdot 10^{12}$	$3 \cdot 10^{13}$	$3 \cdot 10^{15}$
$n \log_2 n$	$9 \cdot 10^4$	10^6	$2 \cdot 10^8$	$4 \cdot 10^9$	10^{11}	10^{12}	10^{14}
n^2	10^3	$8 \cdot 10^3$	$6 \cdot 10^4$	$3 \cdot 10^5$	$2 \cdot 10^6$	$6 \cdot 10^6$	$6 \cdot 10^7$
n^3	100	392	1533	4414	13715	31551	146452
2^n	20	26	32	36	41	45	51
$n!$	10	11	13	14	15	16	18
n^n	7	8	9	10	11	12	13

Tabella 3.4: Dimensioni dei problemi risolvibili in una data quantità di tempo in funzione della complessità $f(n)$ dei problemi.

La figura 3.4 descrive una classificazione dei problemi in funzione del loro carattere di risolvibilità, evidenziando, all'interno dei problemi risolvibili, i problemi trattabili e quelli intrattabili.

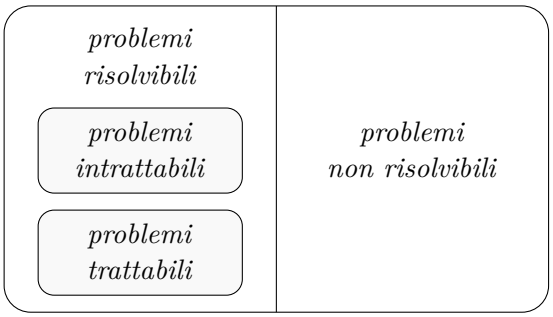


Figura 3.4: Una classificazione dei problemi in funzione del loro carattere di risolvibilità.

3.9 Problemi intrattabili

Il più potente calcolatore che si possa immaginare di costruire non potrà essere più grande dell'universo, non potrà essere fatto di componenti di elaborazione più piccoli dei protoni, non potrà trasmettere le informazioni fra le sue componenti ad una velocità superiore alla velocità della luce. Un semplice calcolo porta alla conclusione che un tale ipotetico calcolatore non potrebbe essere composto da più di 10^{126} componenti. Anche ammettendo che le informazioni di questo calcolatore venissero trasmesse fra le sue componenti ad una velocità uguale a quella della luce, esistono numerosi problemi di grande interesse pratico che, anche per moderati valori della dimensione del problema, tengono in scacco questo calcolatore estremo per decine di miliardi di anni, ossia per un tempo paragonabile a quello dell'età dell'universo. Esistono problemi che, pur risolvibili sul piano teorico in quanto si conoscono degli algoritmi risolutivi, non sono effettivamente risolvibili in quanto richiederebbero tempi esagerati anche per il più potente calcolatore immaginabile; per questo motivo vengono detti *intrattabili*.

L'esistenza di problemi intrattabili può sembrare, a prima vista, un aspetto negativo, un'evidenziazione di un limite della scienza e della tecnica. Sembra paradossale ma, invece, questi problemi costituiscono l'elemento fondante di alcune tecnologie: l'intrattabilità del problema della fattorizzazione in primi di un numero naturale è alla base di tutti i sistemi basati sulla crittografia; l'esplosione dell'albero delle mosse del gioco degli scacchi comporta l'intrattabilità del problema di determinare una strategia vincente nella conduzione del gioco in una partita a scacchi a causa dell'intrattabilità del problema di un'analisi esaustiva dell'albero delle mosse e questo, nonostante l'eccezionale velocità e complessità dei calcolatori attuali, lascia margini di battaglia alla pari all'uomo che gioca a scacchi contro il computer, almeno ai grandi maestri degli scacchi; ed anche se un computer batte agevolmente anche un buon scacchista, le mosse brillanti, ingegnose, artistiche rimangono prerogativa dell'uomo.

ESERCIZI

3.1 Si ha un sacco pieno di chicchi di grano ed uno pieno di chicchi di frumento. Il problema consiste nel decidere quale dei due sacchi contiene più chicchi. Precisare in cosa consistono i *dati* ed i *risultati* del problema. Descrivere un algoritmo che richieda all'esecutore delle capacità il più possibile elementari. Precisare le capacità minime richieste all'esecutore dall'algoritmo che si è descritto. Suggerimento: non serve che l'esecutore sappia *contare*.

3.2 Per i problemi che seguono si individuino i dati ed i risultati e si stabilisca se il problema è ben formulato o meno. Nel caso il problema sia ben formulato, descrivere un algoritmo, evidenziando quali devono essere le capacità dell'esecutore al quale è rivolto.

1. Dato il perimetro e l'area di un triangolo, decidere se esso è un triangolo rettangolo.
2. Data l'area ed il perimetro di un rettangolo, determinarne la diagonale.
3. Dato il perimetro di un rettangolo, determinarne l'area.
4. Data l'area e la base di un rettangolo, determinarne l'altezza.
5. Data l'area ed il perimetro di un rettangolo, determinarne i lati.
6. Dati le lunghezze dei cateti di un triangolo rettangolo, determinarne l'altezza relativa all'ipotenusa.

3.3 Dire, motivando le affermazioni, perchè il seguente problema non è ben formulato: *Determinare l'area di un quadrilatero note le misure a, b, c, d* dei lati. Riformulare il problema in modo da renderlo ben formulato. Disponendo di un esecutore in grado di eseguire le usuali operazioni aritmetiche ($+$, $-$, $*$, $/$, $\surd$, ...) descrivere un algoritmo per risolvere il problema in questione.

3.4 Si consideri la seguente istanza di problema:

> *Suddividere un segmento di* 10 *cm in* 2 *parti di cui una sia i* 3/7 *dell'altra. Si supponga di disporre degli usuali attrezzi da disegno: due squadrette a* 45^o *e* 30^o, 60^o, *una matita, un compasso, con l'ipotesi che le squadrette siano sufficientemente lunghe ma non siano graduate.*

1. Generalizzare il problema proposto.
2. Descrivere un procedimento, da riportare su un testo scolastico di disegno tecnico, per risolvere la classe di problemi che si è individuata.
3. Motivare perchè nella formulazione del quesito precedente si è sentita l'esigenza di precisare "*da riportare su un testo scolastico di disegno tecnico*"?

3.5 Un metodo elementare per effettuare la divisione fra due numeri naturali consiste nel sottrarre ripetutamente il divisore al dividendo fino ad ottenere un valore più piccolo del divisore. Tale valore costituisce il resto della divisione mentre il numero di divisioni eseguite fornisce il quoziente della divisione. Descrivere il procedimento di divisione mediante un algoritmo.

3.6 Supponendo di disporre di un esecutore capace di applicare il teorema di Pitagora, descrivere un procedimento risolutivo per determinare la diagonale di un parallelepipedo rettangolo di cui si conoscono le lunghezze degli spigoli.

3.7 Definire in modo preciso e non ambiguo le operazioni lecite nelle costruzioni con riga e compasso. Con le operazioni definite, determinare un segmento avente per misura la radice quadrata della misura di un dato segmento.

3.8 Si è in un deserto e si dispone di una corda lunga 47 metri e delle forbici. Costruire un recinto di forma quadrata.

3.9 Descrivere il procedimento grafico ed il corrispondente algoritmo per suddividere un dato segmento in 4 parti isometriche.

3.10 Descrivere il procedimento grafico ed il corrispondente algoritmo per determinare il quadrato del quale è data una diagonale.

3.11 Valutare la complessità e la complessità asintotica delle seguenti porzioni di algoritmi, essendo *op* l'operazione dominante ed n la dimensione del problema.

```
1: for n^2 times
2:     for 2n times
3:         op
4:     end for
5:     op
6: end for
7: for n times
8:     op
9: end for
```

```
1: for i from 1 to n do
2:     for j from i to n do
3:         op
4:     end for
5:     op
6: end for
```

```
1: k ← n
2: while k > 0 do
3:     for i from 1 to n do
4:         op
5:     end for
6:     k ← k div 2
7: end while
```

3.12 Siano A_1 ed A_2 due algoritmi funzionalmente equivalenti aventi rispettivamente complessità $f_1(n) = n \log n$ e $f_2 = n^{\sqrt{2}}$. Stabilire quale dei due algoritmi è preferibile, per n sufficientemente grande.

3.13 Determinare per quale valore n_0 (*punto di taglio*) della dimensione n del problema un algoritmo A_1 di complessità polinomiale $f_1(n) = n^{10}$ risulta preferibile rispetto ad un equivalente algoritmo A_2 di complessità esponenziale $f_2(n) = (1.1)^n$.

3.14 Ordinare le seguenti funzioni di complessità:

$$\sqrt{n} \quad \log n^{10} \quad n! \quad n \log n \quad n^n \quad 10n \quad 10^n \quad \log n \quad n^{10} \quad 10^{100} \quad \log^2 n$$

4

RISOLVERE

Risolvere problemi è il compito specifico dell'intelligenza e l'intelligenza è la dote specifica dell'uomo. La capacità di aggirare un ostacolo, di prendere una via indiretta quando non si presenta nessuna via diretta, innalza l'animale intelligente al di sopra di quello ottuso, eleva l'uomo enormemente al di sopra degli animali più intelligenti e gli uomini di talento al di sopra dei loro simili.

G. Polya, *La scoperta matematica*

In questo capitolo vengono presentate alcune metodologie per risolvere i problemi mediante degli algoritmi. Vengono forniti dei suggerimenti alla seguente domanda: *"Come si sviluppa una soluzione algoritmica di un dato problema?"* Verranno analizzate delle situazioni di problemi per la cui soluzione non è sufficiente una semplice applicazione di varie conoscenze e regole, ma si richiede di elaborare delle strategie risolutive in base alle conoscenze acquisite, all'esperienza accumulata ed ad una buona dose d'intuito, creatività e fantasia.

4.1 Soluzione dei problemi

La ricerca di una soluzione di un problema è un'attività complessa ed articolata che si colloca in uno scenario popolato da due tipologie di personaggi: solutori, esecutori che si parlano mediante un linguaggio. Solitamente, il ruolo del solutore è interpretato dall'uomo che, grazie alle sue capacità logiche e creative gioca il ruolo attivo di ideazione del procedimento risolutivo del problema; gli esecutori sono delle entità passive, gli strumenti che eseguono il procedimento: spesso si tratta di macchine programambili che, grazie all'elevata velocità e precisione di elaborazione, forniscono prontamente il risultato del problema.

Molti sono stati i filosofi e gli informatici che hanno suggerito delle indicazioni per la soluzione dei problemi. A seguire sono riportati alcuni stralci che, benché estrapolati da contesti filosofici e scientifici, meritano di essere meditati e riconsiderati nelle situazioni che si incontrano nella soluzione dei problemi.

Dividete ciascun problema che state esaminando in tante parti quante potete e quante ve ne occorrono per risolverlo più facilmente.

[Cartesio, *Discorso sul metodo*]

Questa regola di Cartesio serve poco finché l'arte di dividere [...] rimane inspiegata [...]. Dividendo il suo sottoproblema in parti non convenienti, il risolutore di problemi inesperto può aumentare la sua difficoltà.

[Leibnitz, *Scritti filosofici*]

Le tecniche di costruzione dei programmi si basano su un unico principio: scomporre l'azione necessaria per risolvere un problema in azioni più semplici, e suddividere (di conseguenza) il problema in sottoproblemi.

[N.Wirth, *Principi di programmazione strutturata*]

Molte sono le strade per risolvere tantissimi problemi e molte sono anche le soluzioni degli stessi. Questa situazione non rende più facile il compito di risolvere i problemi: messi dinanzi a una molteplicità di linee di attacco possibili, è normalmente difficile riuscire a capire rapidamente quale percorso è probabilmente infruttuoso e quale può essere produttivo.

[R.G.Dromey, *How to solve it by computer*]

4.2 La metodologia top-down

La maggior parte dei problemi che si incontrano presenta notevoli difficoltà ed una prima analisi non consente di individuare subito un percorso risolutivo. Una valida strategia risolutiva che si applica alla soluzione di generici problemi è nota con il nome di *metodologia top-down* o dei *raffinamenti successivi*; consiste nella risoluzione dei problemi affrontandoli, dall'alto verso il basso, scomponendoli in sottoproblemi più semplici che vengono via-via risolti per approfondimenti successivi. Tale approccio risulta particolarmente indicato quando i problemi non sono di immediata soluzione.

La metodologia top-down per la soluzione di un problema (classe di problemi) P può essere descritta come segue. Indicheremo con $\mathcal{A}(P)$ la struttura di una scomposizione di un problema P in sottoproblemi, ossia un algoritmo in cui compaiono delle istruzioni non direttamente comprensibili dall'esecutore.

Queste istruzioni costituiranno i sottoproblemi da raffinare e dettagliare in fasi successive. Adottando queste notazioni, la soluzione di un problema mediante la metodologia top-down può essere espressa come una sorta di meta-algoritmo rivolto ad un solutore e può essere formulata come descritto nell'algoritmo 1.

Algoritmo 1 - Meta-algoritmo di risoluzione di problemi

Input: problema P da risolvere
Output: procedimento per la soluzione di P
1: **if** la soluzione del problema P è direttamente esprimibile **then**
2: esprimi la soluzione del problema P
3: **else**
4: scomponi il problema P secondo una struttura $\mathcal{A}(P)$
5: risolvi mediante la metodologia top-down i sottoproblemi P_i in $\mathcal{A}(P)$
6: ricomponi le soluzioni $\mathcal{A}(P_i)$ innestandole nella soluzione $\mathcal{A}(P)$
7: **end if**

Nello sviluppo degli algoritmi mediante la metodologia top-down dobbiamo assumere un atteggiamento ottimistico, pensare di avere a disposizione un esecutore sufficientemente potente ad eseguire delle azioni complesse (e sperare di poter esprimere tali azioni in termini delle azioni elementari effettivamente eseguibili da un esecutore reale).

I vantaggi più evidenti indotti dall'applicazione della metodologia top-down per la soluzione dei problemi possono essere riassunti nei seguenti punti:

- Un problema viene scomposto in sottoproblemi più piccoli e quindi più facilmente risolvibili
- In ogni istante si ha una visione complessiva dell'intero problema e della struttura dell'algoritmo risolutivo
- Aumenta la leggibilità dell'algoritmo
- La soluzione di un problema può essere affidata a più persone diverse
- La soluzione può essere ripresa in tempi successivi e sviluppata a blocchi
- Se nella scomposizione vengono evidenziate delle parti di problema simili fra loro, esse possono essere risolte da uno stesso algoritmo
- Le varie fasi di sviluppo dell'algoritmo fungono da documentazione per le successive fasi di maggior dettaglio
- Viene semplificata la localizzazione e la correzione di eventuali errori

Osservazione. Sopra si è preferito usare il termine *metodologia* anziché il termine più deterministico *metodo* in quanto quest'ultimo dà l'idea che si possa cercare la soluzione a colpo sicuro. La ricerca di una soluzione richiede, invece, dei tentativi, delle prove di avvicinamento, dei ritorni sui propri passi e pertanto il termine *metodologia* meglio si adatta ad esprimere questa ricerca della soluzione che si realizza circuendo il problema fino a vincerlo. Usando

un linguaggio figurato, si tratta di catturare una preda che tenta di fuggire, piuttosto che un tiro al bersaglio. In altri termini potremmo esprimerci affermando che non esiste un algoritmo (generale) per sviluppare gli algoritmi ma un algoritmo viene esplicitato seguendo una sorta di fiuto felino che porta alla preda. E neanche il fiuto felino può essere sufficiente a raggiungere la meta ma deve essere unito ad una buona dose d'inventiva, guidato da intelligenza e sorretto da costanza. Il problema di base consiste nel riuscire ad individuare una *buona scomposizione*. A questo proposito risultano appropriate le seguenti parole di Hofstadter tratte dal suo *Gödel, Escher, Bach*.

Non vi è alcuna garanzia che il metodo della riduzione a sottoproblemi funzionerà. Vi sono molte situazioni nelle quali fallisce. Si consideri, per esempio, questo problema. Sei un cane e un amico uomo ti ha appena gettato il tuo osso preferito in un altro giardino al di là di una rete metallica. Tu vedi l'osso attraverso la rete: eccolo là sull'erba, che sembra dire "mangiami, mangiami!". Lungo la rete, a circa venti metri dall'osso, c'è un cancello aperto. Che cosa fai? Alcuni cani corrono semplicemente fino alla rete e restano lí ad abbaiare. Altri si lanciano verso il cancello aperto e poi tornano indietro verso l'osso ambito. Si può dire che entrambi i cani hanno applicato la tecnica della riduzione a sottoproblemi. Essi, però, rappresentano il problema nella loro mente in modi diversi, e in ciò sta tutta la differenza. Il cane che abbaia vede come sottoproblemi

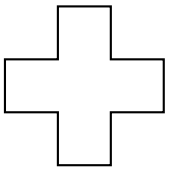
1. *correre alla rete*
2. *attraversarla*
3. *correre all'osso*

ma quel secondo sottoproblema è un osso duro: da cui l'abbaiare. L'altro cane vede come sottoproblemi

1. *raggiungere il cancello*
2. *attraversare il cancello*
3. *correre verso l'osso*

Si noti come tutto dipende dal modo in cui si rappresenta lo spazio dei problemi: cioè, da cosa si percepisce come riduzione del problema (movimento in avanti verso l'obiettivo globale) e che cosa si percepisce come ingrandimento del problema (movimento all'indietro che allontana dall'obiettivo). Alcuni cani cercano di correre direttamente verso l'osso e, quando incontrano la rete, qualcosa scatta nel loro cervello; cambiano subito direzione e corrono verso il cancello. Questi cani si rendono conto che ciò che a prima vista sembrava dover aumentare la distanza tra la situazione iniziale e quella desiderata (cioè, allontanarsi dall'osso per correre verso il cancello aperto) in realtà l'avrebbe diminuita. Inizialmente confondono la distanza fisica con la distanza nel problema. Ogni movimento che allontani dall'osso sembra per definizione una Cosa Cattiva. Ma poi, in qualche modo, si rendono conto che possono spostare la percezione di ciò che li porterà più vicini all'osso. In uno spazio astratto opportunamente scelto, il muoversi verso il cancello è una traiettoria che porta il cane più vicino all'osso! In ogni istante, il cane si sta avvicinando, nel nuovo senso, all'osso. Quindi l'utilità della riduzione a sottoproblemi dipende da come ci si rappresenta mentalmente il problema stesso. Ciò che in uno spazio sembra un indietreggiamento può in un altro spazio apparire un rivoluzionario passo in avanti.

Problema 4.2.1 Disegnare, mediante la grafica della tartaruga, una figura a croce come quella riportata nella figura che segue.



Soluzione. Adottando la metodologia top-down la figura può essere disegnata mediante l'algoritmo 2.

Algoritmo 2 - Disegno di una croce

Input: lunghezza l del lato della croce

1: **for** 4 **times**
2: disegna $\sqcap$ $[P_1]$
3: ruota α $[P_2]$
4: **end for**

Le istruzioni 2. e 3. nell'algoritmo 2 costituiscono due sottoproblemi che possono essere risolti con le seguenti porzioni di algoritmo:

1: $\triangleright$ soluzione sottoproblema P_1:
2: **for** 3 **times**
3: *forward*(l)
4: *left*(90)
5: **end for**

1: $\triangleright$ soluzione sottoproblema P_2:
2: *right*(180)

Ricomponendo le soluzioni dei sottoproblemi P_1 e P_2 all'interno dell'algoritmo 2 si ottiene il definitivo algoritmo 3.

Algoritmo 3 - Disegno di una croce

Input: lunghezza l del lato della croce

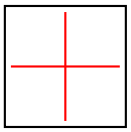
1: **for** 4 **times**
2: **for** 3 **times**
3: *forward*(l)
4: *left*(90)
5: **end for**
6: *right*(180)
7: **end for**

4.3 Forme di scomposizione

Un oggetto (problema, soluzione, figura, ...) può essere scomposto (metodologia top down) o composto (metodologia bottom up) in diverse modalità, descritte a seguire mediante degli esempi nell'ambiente delle figure geometriche che può essere usato come una palestra di allenamento per queste metodologie.

Scomposizione sequenziale

Un quadrato suddiviso in quattro quadrati mediante due linee che uniscono i punti medi dei lati opposti può essere visto come l'*unione* di due figure: il quadrato esterno e la croce interna, come si vede nella figura che segue.

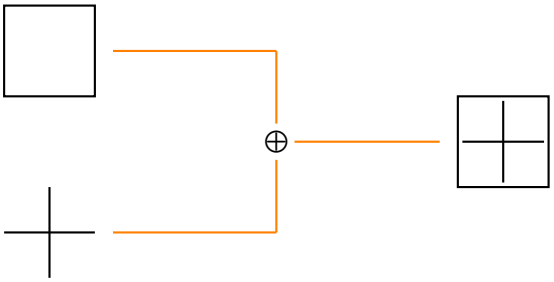


Questa scomposizione corrisponde all'algoritmo 4.

Algoritmo 4 - Scomposizione sequenziale

1: disegna □
2: disegna +

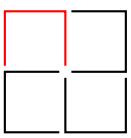
La scomposizione sequenziale può essere vista come un'operazione, denotata con il simbolo ⊕, e può essere descritta con lo schema che segue:



Uno schema di questo tipo può essere letto ed interpretatato in due modi: secondo la metodologia bottom-up (da sinistra verso destra: delle figure elementari vengono composte per formare una figura più complessa); secondo la metodologia bottom-up (da destra verso sinistra: una figura complessa viene scomposta in figure più semplici).

Scomposizione iterativa

La figura del quadrato suddiviso in quattro quadrati come descritto sopra può essere scomposta in modo alternativo come si vede nella figura che segue.

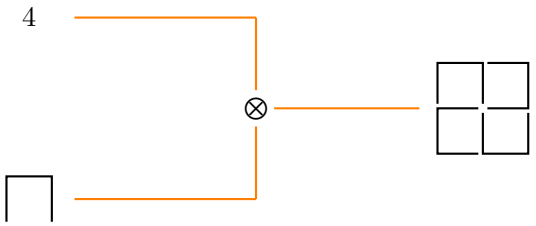


Questa scomposizione corrisponde all'algoritmo 5.

Algoritmo 5 - Scomposizione iterativa

```
1: for 4 times
2:     disegna ⊓
3: end for
```

Similmente al precedente caso di scomposizione sequenziale, la scomposizione iterativa può essere vista come un'operazione, denotata con il simbolo $\oplus$, e può essere descritta con lo schema che segue:



Nella composizione/ripetizione di figure, a differenza di quanto avviene per le operazioni sui numeri, è decisivo, oltre agli 'operandi', anche il loro posizionamento nel piano. Ad esempio l'operazione

$$4 \otimes \sqcap$$

può produrre come risultato due diverse figure a seconda di come vengono disposte le varie componenti:

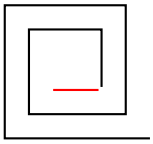
Nella grafica della tartaruga questo fatto si traduce nella definizione di un'azione di raccordo (rotazione, spostamento della tartaruga) interposta fra i disegni delle varie componenti della figura.

Scomposizione ricorsiva

La metodologia ricorsiva per la soluzione dei problemi consiste nel suddividere il problema in sottoproblemi più semplici della stessa forma. In altri termini il problema viene ridefinito in termini di se stesso. Svolto questo passaggio decisivo, il problema risulta quasi risolto: basta solamente definire direttamente la soluzione nel caso di una semplice istanza. Sembra quasi un metodo magico ed inconsistente in quanto la definizione del problema diventa la sua stessa soluzione. Il problema da risolvere è generalmente definito in termini di un numero naturale n che ne rappresenta la dimensione; il nocciolo della tecnica ricorsiva consiste nello scomporre il problema $P(n)$ in un problema $P(n-1)$ di

dimensione $n-1$. A sua volta il problema $P(n-1)$ viene scomposto in un problema $P(n-2)$ e così via. La soluzione si concretizza definendo direttamente la soluzione per l'istanza minima del problema, generalmente $P(0)$.

Talune figure hanno la caratteristica di avere al proprio interno una parte simile alla figura stessa; ad esempio, in una spirale, come quella raffigurata nella figura che segue, si nota che essa è costituita da un segmento e da una spirale (con un lato in meno).



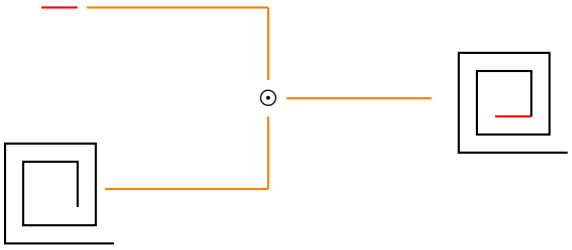
In casi come questo si può ricorrere, oltre che ad un algoritmo iterativo, ad un algoritmo ricorsivo che richiama se stesso, come si vede nell'algoritmo 6.

Algoritmo 6 - Scomposizione ricorsiva: $spirale(n, l, k)$

Input: numero n di segmenti, lunghezza l del lato iniziale, incremento k

```
1: if n > 0 then
2:     forward(l)
3:     left(90)
4:     spirale(n − 1, l + k, k)
5: end if
```

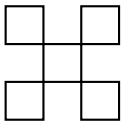
La scomposizione ricorsiva può essere vista come un'operazione, denotata con il simbolo $\odot$, e può essere descritta con lo schema che segue:



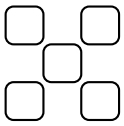
4.4 Criteri di scomposizione

Abbiamo visto che una figura può essere scomposta in più modi. Emerge così l'esigenza di individuare dei criteri in base ai quali scegliere la scomposizione *migliore*. Nel caso di disegno di figure con la grafica della tartaruga si può adottare il criterio di minimizzare la lunghezza del percorso della tartaruga. Come conseguenza di questa scelta risultano preferibili i percorsi che non ripassano su una linea già tracciata. A parità di lunghezza del percorso, dal punto di vista della programmazione risultano preferibili i percorsi definibili mediante la ripetizione di un pattern di base.

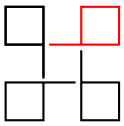
Esempio 4.4.1 - Consideriamo la seguente figura:



La forma più spontanea di scomposizione consiste nell'individuare cinque quadrati, come evidenziato nella figura che segue.



Questa scomposizione, per quanto risulti la più naturale e la più immediata non è la più conveniente in quanto richiede una manovra di raccordo fra i disegni dei quadrati costituita da un'opportuna combinazione di *pennasu* / *avanza* / *pennagiu* / *ruota*. Risulta più comoda la seguente



che corrisponde ai seguenti due algoritmi 7 e 8.

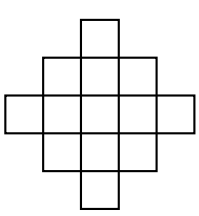
Algoritmo 7 - Figura

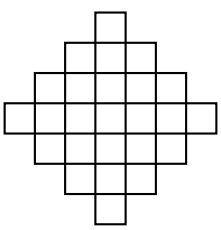
```
1: for 4 times
2:     disegna ⌐□
3: end for
```

Algoritmo 8 - Figura ⌐□

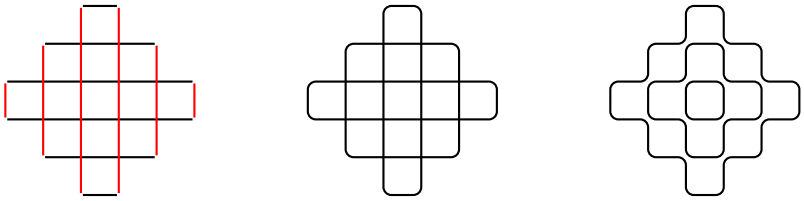
```
1: disegna –
2: disegna □
```

Problema 4.4.1 Mediante la grafica della tartaruga disegnare la classe delle seguenti istanze di figure.





Soluzione. Ciascuna istanza delle figure della classe può essere scomposta in vari modi, come evidenzia la figura che segue:



(a) (b) (c)

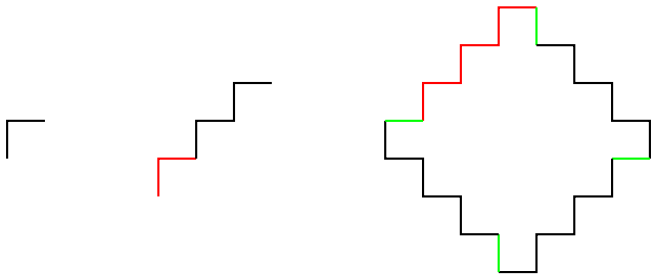
Queste scomposizioni possono essere denominate come segue:

(a) fasci di segmenti paralleli

(b) sequenza di rettangoli

(c) sequenza di cornici

Una soluzione ottimale, corrispondente alla scomposizione (c), evidenzia che la figura è composta da una sequenza di *cornici* concentriche. Una cornice può si basa sulla composizione delle seguenti forme di elementi:



$gradino(l)$ $scala(n,l)$ $cornice(k,l)$

Osservazione. L'aspetto più caratterizzante della metodologia top-down consiste nel trovare delle *buone* ed *utili* scomposizioni dei problemi. Le maggiori difficoltà per la soluzione di un problema risiedono, infatti, non tanto nell'esplicitazione della soluzione dei singoli sottoproblemi elementari, quanto nella capacità di trovare delle adeguate scomposizioni in sottoproblemi. La fase di scomposizione deve tendere a ridurre (in un qualche senso) la complessità del problema originale.

4.5 La metodologia bottom-up

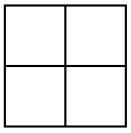
Diversamente da quanto avviene per la metodologia top-down, l'approccio ad un problema mediante la ***metodologia bottom up*** consiste nell'iniziare dal livello dei problemi più elementari che vengono composti per risalire verso il livello del problema da risolvere. Adottando questa metodologia si cerca di individuare e risolvere dapprima i problemi che serviranno per soluzione del problema generale la quale viene costruita ricomponendo le soluzioni dei problemi elementari già risolti.

La metodologia bottom-up viene solitamente utilizzata in una situazione 'matura', quando si ha un repertorio di problemi elementari già risolto; può trattarsi di un bagaglio di esperienze personali pregresse oppure, in un contesto di programmazione, quando si hanno a disposizione delle 'librerie' di funzioni già realizzate oppure ancora, in ambito elettronico, quando si hanno a disposizione dei componenti elettronici elementari che devono essere *scelti* e *composti* per realizzare un *nuovo* componente con una diversa funzionalità.

L'applicazione della metodologia bottom-up dipende da diversi fattori fra i quali:

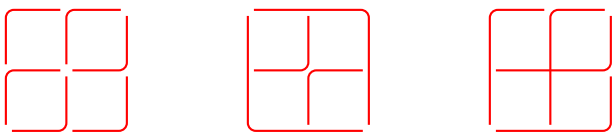
- componenti elementari utilizzabili
- operazioni ammesse sulle singole componenti
- modalità di composizione delle componenti

Esempio 4.5.1 - Vogliamo disegnare mediante la grafica della tartaruga la seguente figura, adottando un approccio bottom-up.



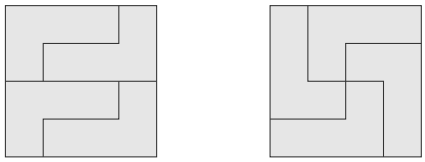
Nella figura che segue sono descritte tre possibili soluzioni corrispondenti alle seguenti tre tipologie di componenti elementari costituite da:

1. figure ad *elle* con lati di segmenti unitari e non scalabili
2. figure ad *elle* scalabili
3. figure ad *elle* costituite da due segmenti uno di lunghezza unitaria e l'altro di lunghezza doppia



Esempio 4.5.2 - Consideriamo un esempio tratto dalla classe di problemi nota come *tassellazioni di figure piane*. Vogliamo comporre un reticolo quadrettato di dimensioni 4×4 utilizzando delle tessere composte da 4 quadretti unitari disposti ad *elle*, come descritto nella figura che segue.

Due possibili soluzioni sono presentate nella figura che segue.



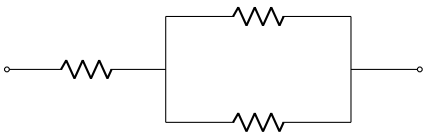
Esempio 4.5.3 - Avendo a disposizione resistenze R di valore unitario, comporle, in serie ed in parallelo per ottenere una resistenza equivalente pari a 1.5 unità. Applicando le regole per il calcolo della resistenza equivalente R_e di più resistenze $R_1, R_2, \ldots, R_n$ collegate in serie:

$$R_e = R_1 + R_2 + \cdots + R_n$$

ed in parallelo:

$$\frac{1}{R_e} = \frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_n}$$

si può costruire la soluzione costituita dal circuito che segue:



4.6 Confronto fra le due metodologie

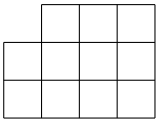
La metodologia bottom-up prevede un approccio intellettualmente diverso dalla metodologia top-down in quanto si risolvono dapprima i problemi elementari preparando i blocchi risolutivi che, assemblati, forniranno la soluzione del problema complessivo. Ma anche qui si vede che la metodologia top-down interviene (almeno in modo latente) in quanto ci si deve preventivamente porre la domanda: *Quali sono i problemi elementari da risolvere?*

La metodologia top-down offre il vantaggio di delineare subito la scomposizione del problema; ciò può comportare lo svantaggio che delle scelte errate

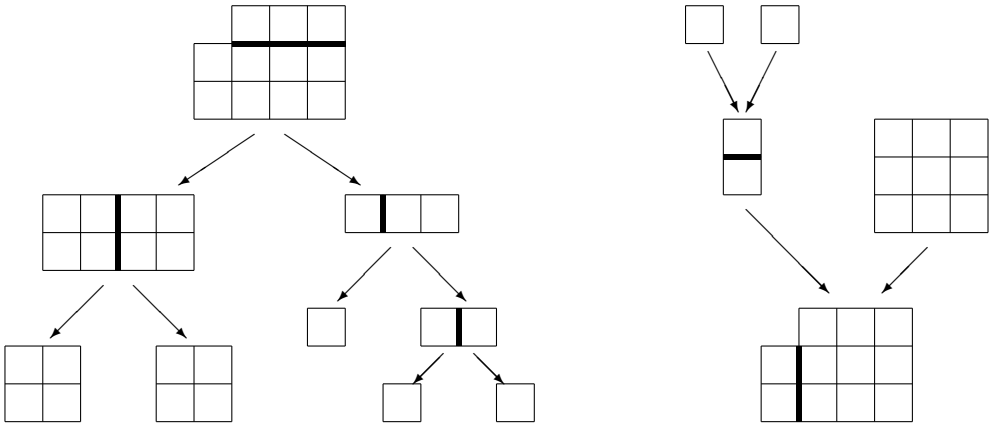
ad un livello elevato compromettano tutte le fasi successive dei livelli più elementari. La metodologia bottom-up, invece, incentiva il riutilizzo di soluzioni di (sotto)problemi già affrontati e risolti in precedenza.

Le metodologie top-down e bottom-up non devono essere considerate alternative: l'atteggiamento più proficuo consiste nel considerarle come degli strumenti che si integrano. Ad esempio, quando si evidenzia un sottoproblema si deve generalizzarlo opportunamente affinché possa risultare utile per situazioni future. È questo un atteggiamento bottom-up.

Esempio 4.6.1 - Il seguente problema illustra in modo schematico un confronto fra la metodologia top-down e la metodologia bottom-up nella soluzione di un problema; il problema consiste nel suddividere in quadrati una data figura piana composta da quadrati unitari.



Nel primo caso (soluzione top-down) si può pensare di scomporre la figura in quadrati usando delle forbici, nel secondo (soluzione bottom-up) si può pensare di ricostruire la figura incollando assieme dei tasselli quadrati di diverse dimensioni. Dalle due soluzioni riportate nella figura 4.1 si può notare, pur dalla banalità dell'esempio proposto, che le metodologie top-down e bottom-up possono portare a procedimenti risolutivi diversi.

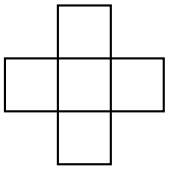


Scomposizione top-down Composizione bottom-up

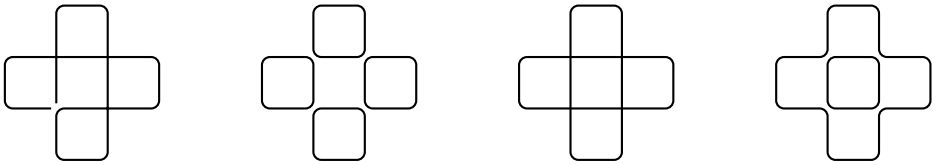
Figura 4.1: Illustrazione della *scomposizione top-down* e della *composizione bottom-up*. Con segmenti più marcati sono indicate le linee di taglio (nella scomposizione top-down) e di unione (nella composizione bottom-up).

La scomposizione di un problema di disegno si traduce nella scomposizione della figura da disegnare, come descritto nel seguente problema 4.6.1.

Problema 4.6.1 Disegnare, mediante la grafica della tartaruga, la figura che segue.



Soluzione. Questa figura può essere disegnata mediante diverse scomposizioni, come illustrato nella figura che segue in cui viene descritta la traiettoria della tartaruga.



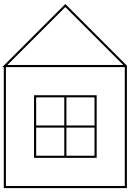
(a) (b) (c) (d)

Queste diverse scomposizioni riflettono diverse gradazioni di applicazione delle metodologie top-down e bottom-up, come descritto ai punti che seguono.

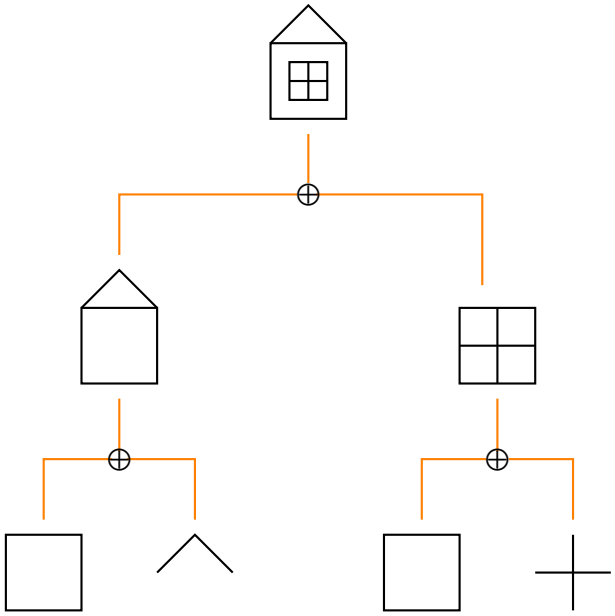
- (a) scomposizione infelice, in quanto non evidenzia gli elementi di simmetria e di ripetitività della figura; addirittura non si evidenzia alcuna scomposizione in sottofigure
- (b) scomposizione efficace in quanto la figura viene scomposta in sottofigure uguali (quadrati); questa soluzione assomma una parte di metodologia bottom-up in quanto il disegno di un quadrato è una figura elementare già risolta
- (c) la scomposizione in due rettangoli è una soluzione altamente top-down in quanto il disegno di un rettangolo è una figura non ancora risolta e richiede quindi un ulteriore passo di analisi per arrivare alla soluzione
- (d) scomposizione altamente bottom-up in quanto sia il quadrato interno e la croce esterna sono delle sottofigure già risolte

Osservazione. Le metodologie top-down e bottom-up vengono spesso usate congiuntamente; addirittura la qualificazione della metodologia (top-down e bottom-up) dipende dal contesto, ossia dalle precedenti soluzioni dei sottoproblemi; di più, l'uso dell'una o dell'altra metodologia dipende dalle intenzioni del solutore: se si scompone il problema P in un sottoproblema Q, se il solutore non dispone della soluzione Q, allora si sta adottando la metodologia top-down; se invece il solutore evidenzia nella scomposizione un sottoproblema Q perché lo ha già risolto si tratta di metodologia bottom-up.

Esempio 4.6.2 - Utilizzando la metodologia top-down costruiamo la seguente 'casetta':



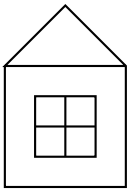
Combinando le operazioni *unione* $\oplus$ e *ripetizione* $\otimes$ precedentemente descritte, si può definire una sorta di *algebra* che permette di esprimere una figura mediante un'*espressione*: la figura della 'casetta' può essere definita mediante la seguente *espressione* ad albero:



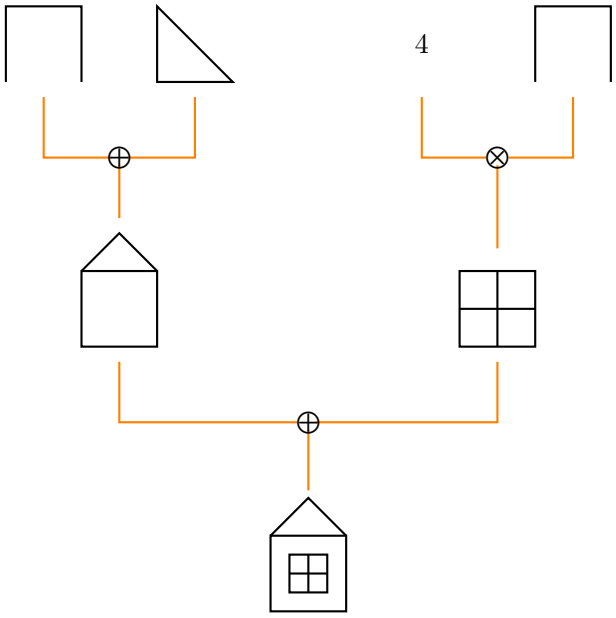
In questa scomposizione emerge, in due diversi punti, il sottoproblema di costruire un quadrato; queste due istanze di problema sono risolvibili mediante lo stesso algoritmo.

Esempio 4.6.3 - Avendo a disposizione più istanze delle seguenti tipologie di forme di componenti lineari ruotabili e scalabili:

costruiamo la seguente 'casetta':



La struttura di composizione della 'casetta' può essere definita mediante la seguente *espressione* in notazione ad albero:



4.7 Un problema numerico

Consideriamo il problema di determinare il numero r definito dalla rappresentazione decimale inversa di un numero naturale n (esempio: $n = 4372$, $r = 2734$). Supponiamo, per il momento, che il numero n non sia divisibile per 10, ossia che il numero, nella sua rappresentazione decimale, non termini con 0. Nell'ipotesi di disporre di un esecutore in grado di eseguire le 4 operazioni aritmetiche di base $+$, $-$, $*$, $\div$, un possibile procedimento risolutivo per questo problema può essere espresso mediante l'algoritmo 9.
Nella formulazione dell'algoritmo 9 si nota che le varie istruzioni informali sono direttamente traducibili in un generico linguaggio di programmazione, ad eccezione del sottoproblema *trasporta la cifra delle unità di n alla fine di r*. Tale sottoproblema può essere ulteriormente scomposto nei seguenti due:

P_1: togli la cifra delle unità dal numero n

P_2: accoda tale cifra alla destra del numero r

Algoritmo 9 - Rovesciamento di un numero naturale in base 10 - ver. A

Input: numero naturale n da rovesciare
Output: numero r rovesciato
1: inizializza r a 0
2: **while** ci sono cifre in n **do**
3: trasporta la cifra delle unità di n alla fine di r
4: **end while**
5: **return** r

Algoritmo 10 - Rovesciamento di un numero naturale in base 10 - ver. B

Input: numero naturale n da rovesciare
Output: numero r rovesciato
1: $r \leftarrow 0$
2: **while** ci sono cifre in n **do**
3: $c \leftarrow$ cifra delle unità di n
4: $n \leftarrow n$ privato della cifra delle unità
5: $r \leftarrow r$ con aggiunta la cifra sulla destra
6: **end while**
7: **return** r

La forma quasi definitiva è riportata nell'algoritmo 10.
Raffinando ulteriormente l'algoritmo 10, si perviene alla forma definitiva (algoritmo 11).

Algoritmo 11 - Rovesciamento di un numero naturale in base 10 - ver. C

Input: numero naturale n da rovesciare
Output: numero r rovesciato
1: $r \leftarrow 0$
2: **while** $n > 0$ **do**
3: $q \leftarrow n \div 10$
4: $p \leftarrow q * 10$
5: $c \leftarrow n - p$
6: $n \leftarrow n \div 10$
7: $r \leftarrow r * 10 + c$
8: **end while**
9: **return** r

In una forma più compatta, le istruzioni di assegnazione interne al ciclo nell'algoritmo 11 potrebbero essere scritte come segue:

$$r \leftarrow (r * 10) + (n - (n \div 10) * 10)$$
$$n \leftarrow n \div 10$$

Si lascia per esercizio l'analisi del comportamento dell'algoritmo precedente nei casi in cui il numero da invertire termini con delle cifre 0.

ESERCIZI

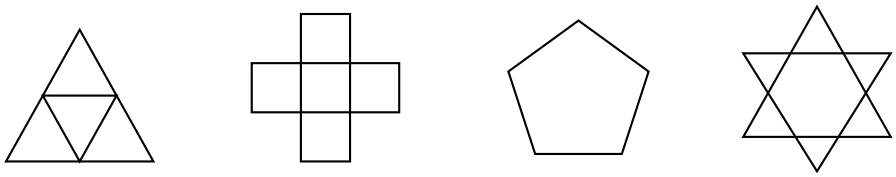
4.1 Illustrare, mediante degli esempi, in cosa consistono le *metodologia top-down* e *bottom-up* per la soluzione dei problemi, evidenziando i vantaggi che esse offrono.

4.2 Si consideri il caso di un bambino che gioca con i mattoncini Lego per costruire una casetta. Si spieghi, in questo contesto, in cosa consistono le *metodologie top-down* e *bottom-up*.

4.3 Componendo, in serie ed in parallelo, delle resistenze unitarie, realizzare un circuito di resistenza equivalente pari a 0.75 unità.

4.4 Una qualità desiderabile per un algoritmo di disegno di una figura mediante la grafica della tartaruga consiste nel disegnare l'intera figura mediante un *percorso euleriano*, ossia senza ripassare su un tratto già tracciato e senza alzare la penna dal piano di tracciamento. Analizzare ed individuare quali delle scomposizioni $(a), (b), (c)$ e (d) presentate nell'esempio 4.6.1 soddisfano a questa condizione.

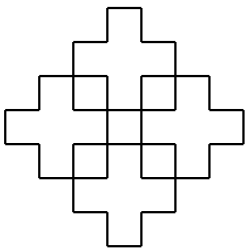
4.5 Individuare delle possibili scomposizioni per ciascuna delle seguenti figure e scrivere dei corrispondenti algoritmi per disegnarle mediante la grafica della tartaruga.



Disegnare le diverse tipologie di figure mediante un unico algoritmo valido per una classe di figure che includa tutte le istanze descritte.

4.6 Definire una classe di figure che comprenda come casi particolari le tre tipologie di figure descritte nel precedente esercizio 4.5. Scrivere un algoritmo per disegnare una generica figura della classe. Particolarizzare i dati dell'algoritmo in modo che produca il disegno di ciascuna delle tre tipologie di figure sopra descritte.

4.7 Mediante la grafica della tartaruga ed applicando le metodologie top-down e bottom-up, disegnare la figura riportata sotto.



4.8 Analizzare e discutere comparativamente le due soluzioni proposte negli esempi 4.6.2 e 4.6.3.

4.9 Con riferimento alla figura 'casetta' descritta nell'esempio 4.6.2, scomporre la figura nelle seguenti due situazioni:

1. usando il minor numero possibile di tipologie di forme
2. usando tutte quattro le tipologie di forme

4.10 Disegnare mediante la grafica della tartaruga la casetta presentata nell'esempio 4.6.2, senza ripassare su un tratto già disegnato ed utilizzando il minor numero possibile di spostamenti con la penna alzata.

4.11 Avendo a disposizione degli elementi di ferro ad "elle", di tutte le dimensioni necessarie, costruire la casetta presentata nell'esempio 4.6.2, utilizzando il minor numero possibile di 'saldature'.

Parte II

CODING

5

LINGUAGGI

Il linguaggio ci permette di trattare i nostri pensieri più o meno come se fossero cose qualsiasi.

M. Minsky, *La società della mente*

Abbiamo visto, nei precedenti capitoli, due categorie di agenti: il solutore e l'esecutore i quali interagiscono in modo cooperativo per risolvere i problemi: il solutore costituisce la mente e l'esecutore il braccio. La separazione dei ruoli e delle responsabilità fra solutore ed esecutore e la contestuale esigenza di cooperazione, comporta la necessità di comunicazione fra questi due attori: in pratica serve un linguaggio, più o meno formale, mediante il quale il solutore possa descrivere all'esecutore il procedimento da eseguire. Il linguaggio viene utilizzato anche per descrivere le fasi di sviluppo del procedimento risolutivo, prima di arrivare alla versione finale da sottomettere all'esecutore. In questo caso risulta uno strumento ad uso interno del solutore.

L'evoluzione storica delle notazioni e dei linguaggi per descrivere gli algoritmi segue una linea parallela all'evoluzione degli strumenti di calcolo e di elaborazione. All'inizio, quando ancora gli algoritmi venivano eseguiti dall'uomo, la descrizione dei procedimenti era spesso basata sulla descrizione, in modo informale e discorsivo, del processo di calcolo riferito a delle specifiche istanze del problema. Quando si iniziarono ad utilizzare degli esecutori automatici, ed in particolare con l'uso dei computer, grosso modo alla metà del secolo scorso, sorse l'esigenza di usare dei linguaggi rigorosi e non ambigui che vennero qualificati come *linguaggi di programmazione*.

5.1 Algoritmi e processi

In prima approssimazione, un *algoritmo* è la *descrizione sulla carta* delle istruzioni da elaborare, mentre un *processo* consiste nell'*elaborazione nel tempo* delle istruzioni. Per comprendere la sostanziale differenza esistente fra il concetto di algoritmo e di processo basta riflettere sulla differenza che esiste fra i due termini *descrizione* ed *elaborazione*: la prima attività si svolge nello spazio, la seconda nel tempo. In altri termini si può dire che un processo è una particolare sequenza di elaborazione delle istruzioni, compatibile con la struttura dell'algoritmo. In termini figurati si può affermare che l'algoritmo descrive le possibili strade che connettono le istruzioni, mentre un processo è il compimento di un particolare percorso lungo le strade tracciate dall'algoritmo. I due grafici riportati nella figura 5.1 descrivono dei nodi di biforcazione collegati da linee del flusso potenziale dell'algoritmo e da linee del flusso esecutivo di un processo.

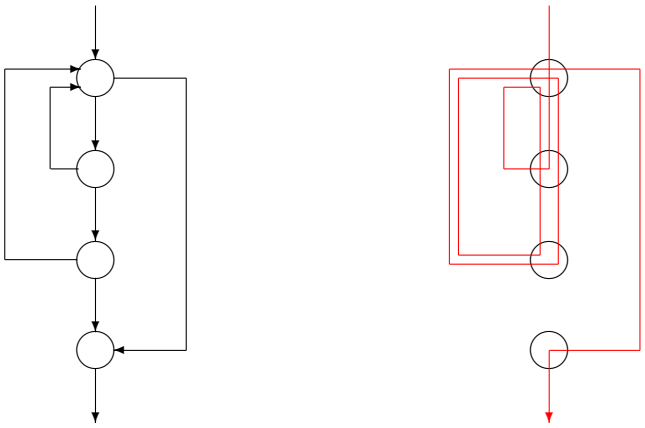


Figura 5.1: Comparazione fra *algoritmo* e *processo*: (*a*) linee del flusso potenziale dell'algoritmo; (*b*) linee del flusso esecutivo del processo.

5.2 Controlli sequenziali, condizionali e ciclici

L'elaborazione sequenziale delle istruzioni è quella più naturale e viene intrapresa dagli esecutori in assenza di altre direttive esplicite. Questa semplice struttura sequenziale di elaborazione risulta però inadeguata per la descrizione degli algoritmi in quanto comporta che l'elaborazione delle istruzioni avvenga solo in modo sequenziale, e che, quindi, ogni istruzione specificata in un algoritmo venga elaborata una sola volta. Per superare questi limiti e rendere il linguaggio per la descrizione degli algoritmi più potente servono dei *controlli* che consentano di *ripetere* alcune istruzioni e di *evitarne* altre.

La forma più elementare di algoritmo corrisponde al caso di istruzioni *descritte sequenzialmente* (sulla carta) dal solutore ed *elaborate sequenzialmente* (nel tempo) dall'esecutore. Questa modalità di descrizione ed elaborazio-

ne degli algoritmi è troppo povera e non consente la descrizione di algoritmi interessanti che possano dar vita a processi complessi per i quali non esista una corrispondenza biunivoca fra la sequenza delle istruzioni descritte nell'algoritmo e la sequenza di elaborazione del processo. La possibilità di avere flussi esecutivi più articolati che non la semplice elaborazione sequenziale si fonda sull'uso combinato della possibilità di *evitare*, in base a delle condizioni, l'elaborazione di alcune istruzioni e sulla possibilità di *ripetere* più volte l'elaborazione di alcune istruzioni. A questo scopo vengono utilizzati dei *controlli*, ossia delle particolari direttive che, opportunamente intercalate fra le istruzioni dell'algoritmo, specificano le possibili linee di percorrenza dei flussi elaborativi definendo molti (anche infiniti) potenziali percorsi; in fase di elaborazione sarà intrapreso uno di questi percorsi e si darà vita ad un processo.

I potenziali flussi esecutivi di un algoritmo possono essere descritti mediante pochi e ben fatti schemi, descrivibili graficamente come riportato nella figura 5.2, dove lungo le linee ci possono essere delle istruzioni da elaborare e dei controlli.

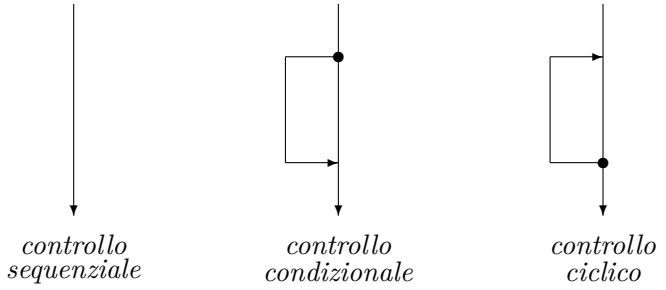


Figura 5.2: Schemi fondamentali dei controlli.

L'assunzione dell'ipotesi che, in assenza di altre indicazioni, implicitamente le istruzioni vengano elaborate sequenzialmente, come sono descritte, comporta che un salto abbia il significato fortemente diverso a seconda che si tratti di un *salto in avanti* o di un *salto all'indietro*. Nel primo caso l'effetto è di *evitare* delle istruzioni mentre il secondo caso permette di *ripetere* delle istruzioni.

5.3 Livello dei linguaggi

L'elemento che più caratterizza un linguaggio per descrivere gli algoritmi è costituito dalla forma delle direttive di controllo delle azioni, ossia dalle istruzioni che specificano all'esecutore la sequenzialità delle azioni da eseguire. Una tradizionale classificazione distingue i *linguaggi di basso livello* dai *linguaggi di alto livello*. La determinazione del livello di un linguaggio (basso/alto) avviene proprio sulla base delle direttive di controllo: i linguaggi che forniscono dei controlli a salti, ossia della forma "*continua l'esecuzione all'istruzione scritta al punto* α" vengono generalmente detti *linguaggi di basso livello*; l'attributo *basso* deriva dal fatto che questi linguaggi, rivolti a degli esecutori rappresentati dai calcolatori, sono molto aderenti alla modalità di esecuzione delle azioni da parte della macchina, e quindi, in definitiva, alla struttura fisica della macchina

esecutrice.

5.4 Linguaggi di basso livello

Il linguaggio che viene utilizzato nella comunicazione fra solutore ed esecutore dipende principalmente dalle capacità dell'esecutore, intese sia come repertorio e potenzialità azioni elementari che l'esecutore è in grado di espletare, sia dalla ricchezza ed articolazione della sintassi del discorso nel quale vengono inserite.

Le notazioni dei linguaggi nelle forme di basso livello riflettono le modalità fisiche di esecuzione degli attuali elaboratori; queste forme si basano sul seguente insieme minimale e completo di controlli:

- $goto\ \alpha$ (*salto incondizionato*)
- $if\ \mathcal{C}\ goto\ \alpha$ (*salto condizionato*)

dove α è un indicatore di posizione che individua un'istruzione (esecutiva o di controllo) di un algoritmo, e $\mathcal{C}$ è una condizione.

Gli esempi di seguito riportati illustrano i controlli delle azioni nelle forme di basso livello precedentemente descritte. In questi esempi le etichette dei salti incondizionati e condizionati sono sostituite dai numeri identificativi di ciascuna linea dell'algoritmo.

Esempio 5.4.1 - Consideriamo il seguente problema:

> *Determinare la distanza fra due numeri a e b.*

Ammettendo che il campo di appartenenza dei numeri coinvolti nell'algoritmo sia costituito dai numeri interi e supponendo che l'esecutore sia in grado di eseguire solamente le seguenti azioni elementari:

- applicare le quattro operazioni aritmetiche
- confrontare due numeri

un effettivo procedimento risolutivo può essere formulato come riportato nell'algoritmo 1.

Algoritmo 1 - Distanza fra due numeri interi

Input: numeri interi a e b
Output: distanza fra a e b

```
1: d ← a − b
2: if d ≥ 0 goto 4
3: d ← −d
4: return d
```

Vincolando ad usare solamente numeri naturali, il che equivale a supporre che l'esecutore conosca e sia in grado di operare solo sui numeri naturali, la sequenza di istruzioni precedenti non rappresenta più un algoritmo corretto in quanto, nell'ambito sei numeri naturali non risulta eseguibile l'operazione

$a - b$, ed in ogni caso non avrebbe più senso l'operazione unaria $-d$. Poiché il risultato che si cerca (d) è un numero naturale, è plausibile pensare che esista un algoritmo risolutivo per il quale sia sufficiente un ambiente di variabilità degli oggetti non più ampio dello spazio delle soluzioni. Questa supposizione è confermata dall'algoritmo 2 che determina la distanza fra due generici numeri naturali a e b, con la condizione che l'esecutore sia in grado di manipolare soltanto numeri naturali.

Algoritmo 2 - Distanza fra due numeri naturali

Input: numeri naturali a e b
Output: distanza fra a e b
1: **if** $a \geq b$ **goto** 4
2: $d \leftarrow b - a$
3: **goto** 5
4: $d \leftarrow a - b$
5: **return** d

Osservazione. La situazione presentata nel precedente esempio richiama, pur nella sua banalità, gli sforzi degli algebristi italiani del Cinquecento che cercavano dei procedimenti risolutivi per determinare le radici reali delle equazioni polinomiali di terzo e quarto grado, evitando di passare ed operare nel campo (allora minato, perchè poco conosciuto) dei numeri complessi.

Esempio 5.4.2 - Nell'algoritmo che segue viene calcolato il fattoriale di un generico numero naturale, basandosi sulla definizione stessa di fattoriale: $n! \stackrel{\text{def}}{=} 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$.

Algoritmo 3 - Algoritmo fattoriale di un numero naturale

Input: numero naturale n
Output: fattoriale di n
1: $f \leftarrow 1$
2: $k \leftarrow 1$
3: **if** $k > n$ **goto** 7
4: $f \leftarrow f * k$
5: $k \leftarrow k + 1$
6: **goto** 3
7: **return** f

5.5 I diagrammi a blocchi

La notazione dei diagrammi a blocchi si fonda sulla particolare notazione grafica utilizzata per descrivere le due forme di controllo di salto incondizionato e condizionato. In notazione grafica i due controlli di salto *goto* e *if-goto* vengono indicati con le notazioni riportate nella figura 5.3.

Figura 5.3: Controlli di basso livello.

Una notazione frequentemente utilizzata, anche se negli ultimi decenni è stata gradualmente abbandonata, è quella dei *diagrammi a blocchi*. Gli elementi di base per costruire un diagramma a blocchi sono descritti nella figura 5.4.

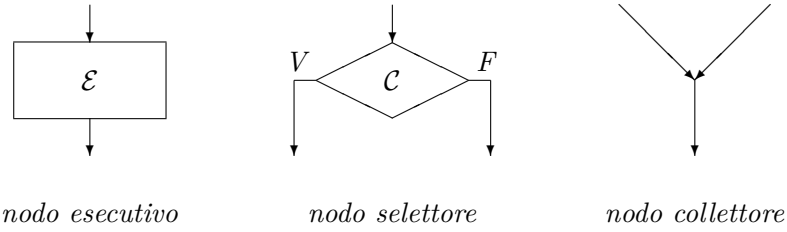


Figura 5.4: Le diverse tipologie dei nodi dei diagrammi a blocchi.

Il significato dei simboli descritti nella figura 5.4 è il seguente:

- *nodo esecutivo* : esegui l'istruzione $\mathcal{E}$
- *nodo selettore* : se la condizione $\mathcal{C}$ è vera continua sul ramo V altrimenti sul ramo F
- *nodo collettore* : continua al punto indicato

Le frasi del linguaggio dei diagrammi a blocchi vengono costruite connettendo i vari nodi sopra descritti mediante delle linee. Viene adottata la convenzione che un nodo esecutivo sul quale non confluisca alcun ramo è il primo da essere eseguito (*punto di entrata*) mentre un nodo esecutivo dal quale non si diparta alcun ramo segnala la fine delle elaborazioni (*punto di uscita*). Per rendere deterministicamente eseguibili le istruzioni descritte, si impone che ogni diagramma a blocchi abbia un solo punto d'entrata. Inoltre, poiché, come si vedrà nel seguito, risulterà necessario concatenare fra loro sequenzialmente più diagrammi a blocchi, si impone la condizione che ogni diagramma a blocchi abbia un solo punto d'uscita. Un diagramma che gode di queste due proprietà dicesi *proprio*.

La regola che segue permette di identificare univocamente le frasi corrette del linguaggio dei diagrammi a blocchi (più avanti saranno limitate, considerandone solo un sottoinsieme ben formato, costituito dai controlli della

programmazione strutturata). Viene fornita una definizione costruttiva, non dicendo quali sono le frasi corrette ma dicendo come si costruiscono le frasi corrette mediante il formalismo dei diagrammi a blocchi.

REGOLA di costruzione dei diagrammi a blocchi. Le frasi corrette si ottengono unendo fra loro, mediante delle linee, dei nodi (esecutivi, selettori, collettori), in modo da ottenere un diagramma proprio (avente un unico punto di entrata ed un unico punto di uscita) connesso (costituito di un unico pezzo).

Osserviamo che un diagramma a blocchi viene rappresentato mediante un grafo connesso in cui i nodi elaborativi hanno il significato di *istruzione* (da elaborare), mentre i nodi alternativi hanno il significato di *controllo*.

Esempio 5.5.1 - L'algoritmo 3 può essere descritto mediante il diagramma a blocchi riportato nella figura 5.5.

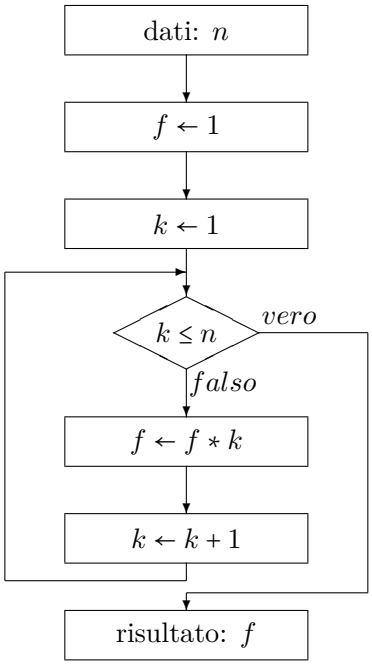


Figura 5.5: Diagramma a blocchi dell'algoritmo del fattoriale.

Un diagramma a blocchi può essere interpretato seguendo le seguenti indicazioni:

1. parti dall'inizio
2. prosegui lungo la strada indicata dalla freccia
3. esegui le istruzioni che incontri lungo il percorso
4. quando arrivi ad un bivio valuta la condizione e prosegui lungo la strada corrispondente al valore di verità ottenuto dalla valutazione
5. quando arrivi alla fine, fermati

5.6 Limiti dei linguaggi a salti

Il linguaggio di basso livello, con controlli delle azioni mediante dei salti, come descritto nel capitolo precedente, è molto semplice e molto potente. Qui con il termine *semplice* si intende che le regole che generano le frasi corrette (ossia la *grammatica*) sono poche e facilmente applicabili. Con il termine *potente* si indica il fatto che con tale linguaggio si può descrivere ogni cosa che sia fattibile mediante le potenzialità base dell'esecutore. Tale potenza rappresenta però più un difetto che un pregio del linguaggio, in quanto è possibile costruire dei diagrammi a blocchi della forma riportata nella figura 5.6 per i quali è difficilmente comprensibile la logica dei potenziali flussi di elaborazione; questo inconveniente emerge con più evidenza quando l'algoritmo è lungo e complesso.

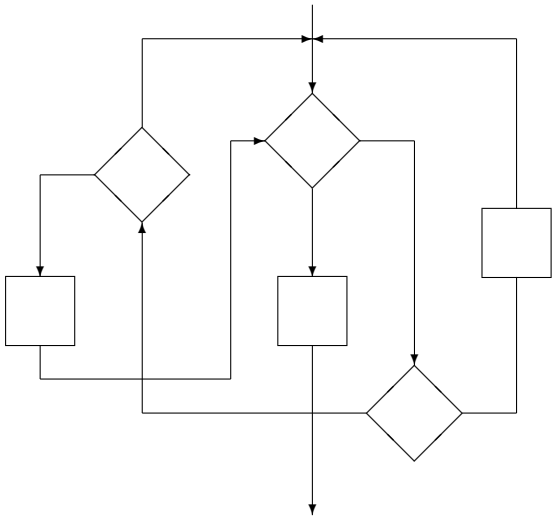


Figura 5.6: Esempio di diagramma a blocchi non strutturato.

L'aspetto negativo dei diagrammi a blocchi non strutturati consiste nel fatto che essi risultano poco comprensibili e quindi difficilmente adattabili e correggibili; in gergo tecnico si afferma che tali programmi sono difficilmente *manutenibili*. Per queste forme di programmazione primitiva è stato coniato il termine dispregiativo di *programmazione a spaghetti* che ben rende il senso del caotico sviluppo del flusso dell'elaborazione delle espressioni. Per escludere questi casi, conviene limitare l'insieme delle frasi costruibili ad un piccolo sottoinsieme di frasi ben fatte, che soddisfino a degli schemi più restrittivi. Con queste limitazioni si vedrà che, comunque, non si limiterà le possibilità di esprimere tutto ciò che è realizzabile da un generico esecutore ed esprimibile altrimenti in termini di un linguaggio di basso livello; è questo il risultato stabilito dal teorema di Böhm-Jacopini.

Una delle fasi più elementari dell'evoluzione dei linguaggi di programmazione verso forme più evolute è consistita nell'individuazione di alcuni costrutti fondamentali per il controllo delle azioni, sui quali si fonda la *programmazione strutturata*. Questi controlli fondamentali sono disponibili in tutti i più recenti linguaggi di programmazione, detti linguaggi di *alto livello*, in contrapposizione a quelli di *basso livello*, proprio per evidenziare la loro struttura più evoluta rispetto al modo di agire della macchina e più vicina al modo di pensare dell'uomo.

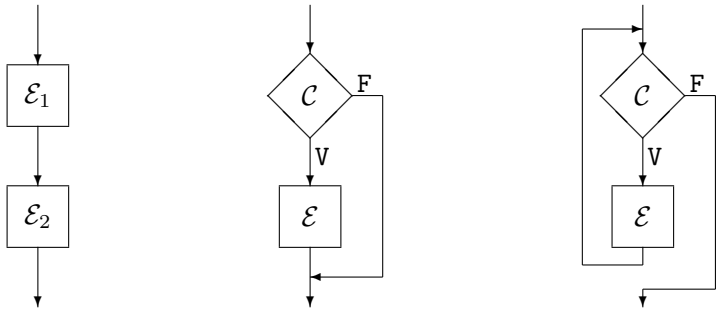
I linguaggi di programmazione attuali predispongono i controlli tipici della programmazione strutturata. Qualora fosse necessario o si volesse utilizzare comunque i controlli di basso livello risulta comodo partire dall'algoritmo espresso in notazione strutturata e poi tradurlo in forme di basso livello. L'impostazione corretta è dunque quella di esprimere l'algoritmo, in prima istanza, in una forma a noi più congeniale e più aderente al nostro modo di pensare, e di rinviare a fasi successive lo *sviluppo dei dettagli* e la *traduzione* verso forme comprensibili all'esecutore. La traduzione in una forma di basso livello risulta meccanica e non richiede alcuno sforzo supplementare che non sia quello di traduzione di una forma sintattica di alto livello in una equivalente forma sintattica di basso livello, indipendentemente dal significato delle frasi.

5.7 La programmazione strutturata

Nel 1968, in un battagliero articolo, considerato una sorta di manifesto della programmazione strutturata, Dijkstra, uno dei pionieri della programmazione, denunciava i pericoli a cui può portare l'uso indisciplinato delle istruzioni di salto, che può compromettere la comprensibilità di un programma da parte del suo stesso autore. Sulla spinta di queste critiche e sulla base del risultato positivo stabilito dal teorema di Böhm-Jacopini, fin dalla fine degli anni '60 si sono sviluppate delle metodologie di programmazione strutturata, aventi l'obiettivo di permettere solo un insieme organico di regole di programmazione basate su dei controlli di alto livello. Al di là di un'astiosa e talvolta sterile polemica sull'uso dei *goto*, la programmazione strutturata risulta particolarmente efficace come strumento di sviluppo degli algoritmi, fondato sulla metodologia di scomposizione dei problemi in sottoproblemi.

Per ottenere dei diagrammi più strutturati è necessario limitare le possibilità di costruzione degli stessi, ossia limitare le possibilità della *Regola di costruzione dei diagrammi a blocchi*. Tali limitazioni definiranno implicitamente un altro linguaggio. Sarà adottata la seguente

REGOLA di costruzione dei diagrammi a blocchi strutturati. Un diagramma a blocchi è strutturato se è composto da un'istruzione elementare (assioma). Se $\mathcal{E}$, $\mathcal{E}_1$, $\mathcal{E}_2$ sono diagrammi strutturati e $\mathcal{C}$ è una condizione, allora anche i cstrutti descritti nella figura 5.7 sono diagrammi strutturati.



controllo *sequenziale* — controllo *condizionale* — controllo *ciclico*

Figura 5.7: Controlli fondamentali della programmazione strutturata.

In altri termini, un diagramma a blocchi è strutturato se è composto da una singola istruzione oppure è composto unendo dei blocchi strutturati in una delle tre forme strutturate descritte nella nella figura 5.7. La regola di costruzione dei diagrammi a blocchi strutturati può essere espressa equivalentemente affermando che *espandendo dei blocchi strutturati mediante dei controlli strutturati si ottengono dei diagrammi a blocchi strutturati.*

Le tre forme base dei diagrammi a blocchi sopra descritte corrispondono a delle strutture di alto livello per il controllo delle azioni, caratteristiche del modo di ragionare degli uomini per l'organizzazione delle azioni. Degli esempi corrispondenti a queste forme sono i seguenti:

- *Versa il caffè* ***e poi*** *aggiungi lo zucchero.*
- ***Se*** *il caffè è doppio* ***allora*** *aggiungi un altro cucchiaino di zucchero.*
- ***Finché*** *lo zucchero non è disciolto* ***continua*** *a mescolare.*

5.8 Controlli generalizzati

I tre controlli fondamentali della programmazione strutturata descritti nel paragrafo precedente vengono spesso usati in combinazione fra loro, permettendo di generare dei formati di frase generalizzati, molto usati nella descrizione degli algoritmi. Tali controlli sono riportati nella figura 5.8. Questi controlli sono strutturati in quanto sono (facilmente) esprimibili come composizione dei controlli fondamentali. In base a questa osservazione ed alla regola di programmazione strutturata, questi controlli possono essere usati come componenti per la costruzione di algoritmi strutturati. Queste forme generalizzate sono così frequentemente utilizzate da essere considerate fondamentali, mentre

le forme analoghe definite precedentemente, dalle quali essi derivano, vengono considerate come dei casi particolari di queste forme generalizzate.

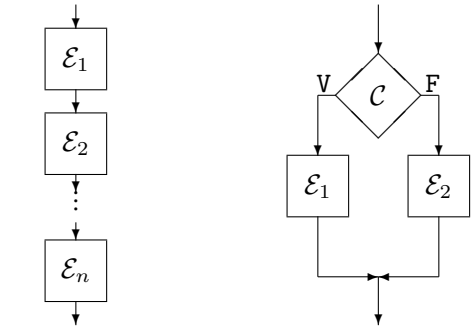


Figura 5.8: Controlli generalizzati della programmazione strutturata.

Osservazione. L'enfasi che le moderne metodologie di programmazione pongono sulla programmazione strutturata è dovuta al fatto che la scomposizione di un problema mediante la metodologia top-down e lo sviluppo mediante raffinamenti successivi (strutturati) porta automaticamente a dei programmi strutturati. Questa impostazione ha sminuito notevolmente l'importanza dei diagrammi a blocchi, i quali non vengono più usati come metodologia o come strumento per lo sviluppo degli algoritmi, ma solo per descrivere la semantica dei controlli fondamentali. Una diretta conseguenza del fatto che la scomposizione di un problema in modo strutturato porta a forme strutturate è che l'algoritmo risolutivo finale è anch'esso strutturato.

5.9 Notazioni testuali

A seguire sono indicate le notazioni testuali dei controlli della programmazione strutturata descritti nei paragrafi precedenti.

Controllo sequenziale

Il controllo per elaborare sequenzialmente delle istruzioni $\mathcal{E}_1, \ldots, \mathcal{E}_n$ viene espresso come descritto nell'algoritmo 4.

Algoritmo 4 - Controllo sequenziale

1: istruzione $\mathcal{E}_1$
2: ...
3: istruzione $\mathcal{E}_n$

Esempio 5.9.1 - Movimento di un robot su un percorso quadrangolare, a partire dalla posizione corrente del robot:

Algoritmo 5 - Movimento di un robot su un percorso quadrangolare

1: $P \leftarrow$ posizione corrente
2: avanza di 10 passi
3: ruota a sinistra di 110^o
4: avanza di 25 passi
5: ruota a sinistra di 45^o
6: avanza di 12 passi
7: vai al punto P

Controllo condizionale

Il controllo di elaborazione condizionale, che a parole si esprime nella forma *Se la condizione $\mathcal{C}$ è vera esegui le istruzioni $\mathcal{E}$.* o nella forma estesa *Se la condizione $\mathcal{C}$ è vera esegui le istruzioni $\mathcal{E}_1$ altrimenti esegui le istruzioni $\mathcal{E}_2$.* è descritto negli algoritmi 6 e 7.

Algoritmo 6 - Controllo condizionale

1: **if** condizione $\mathcal{C}$ **then**
2: istruzioni $\mathcal{E}$
3: **end if**

Algoritmo 7 - Controllo condizionale generalizzato

1: **if** condizione $\mathcal{C}$ **then**
2: istruzioni $\mathcal{E}_1$
3: **else**
4: istruzioni $\mathcal{E}_2$
5: **end if**

Esempio 5.9.2 - Esecuzione di un movimento elementare da parte di un robot:

Algoritmo 8 - Movimento elementare di un robot

1: **if** c'è spazio davanti **then**
2: avanza di 1 passo
3: **else**
4: ruota a sinistra di 90^o
5: **end if**

Controllo ciclico

L'algoritmo 9 riporta la notazione testuale del controllo ciclico *Finché la condizione $\mathcal{C}$ è vera continua ad eseguire le istruzioni $\mathcal{E}$.*

Algoritmo 9 - Controllo ciclico

```
1: while condizione C do
2:     istruzioni E_1 ... E_n
3: end while
```

Esempio 5.9.3 - Raggiungimento di una posizione obiettivo da parte di un robot:

Algoritmo 10 - Movimento ciclico di un robot

```
1: while non hai raggiunto l'obiettivo do
2:     avanza di 1 passo
3:     valuta la distanza dall'obiettivo
4:     aggiusta la direzione di avanzamento
5: end while
```

Esempio 5.9.4 - I controlli descritti sopra compaiono solitamente combinati assieme, come mostra l'algoritmo 11.

Algoritmo 11 - Percorso di un robot

```
1: inizializza il robot
2: while non hai raggiunto l'obiettivo do
3:     if c'è spazio davanti then
4:         avanza di poco
5:     else
6:         while non c'è spazio davanti do
7:             ruota a destra di poco
8:         end while
9:     end if
10:    aggiusta la direzione di avanzamento
11: end while
```

5.10 Altri controlli

Oltre ai controlli (fondamentali e derivati) precedentemente descritti, molti linguaggi di programmazione prevedono degli ulteriori controlli. Anche se questi possono essere espressi in termini dei controlli fondamentali, la ricchezza dei controlli utilizzabili, pur non aumentando la potenza espressiva di un linguaggio, permette di migliorare la compattezza e leggibilità dei programmi.

Nel caso in cui le istruzioni $\mathcal{E}$ debbano essere elaborate un prefissato numero k di volte, si può usare la forma di controllo descritta nell'algoritmo 12.

Algoritmo 12 - Controllo ciclico ripetitivo

```
1: for k times
2:     istruzioni E
3: end for
```

Un'altra forma, più generale, di ciclo predeterminato è il ciclo *for* che risulta comodo qualora serva sapere internamente al ciclo il numero dell'iterazione corrente. In molti linguaggi, il ciclo *for* viene espresso come descritto nell'algoritmo 13, essendo e_1 ed e_2 due espressioni intere ed i una variabile che assume consecutivamente i valori interi da e_1 a e_2.

Algoritmo 13 - Controllo ciclico iterativo

```
1: for i from e1 to e2
2:     istruzioni E
3: end for
```

L'algoritmo 13 può essere generalizzato con l'algoritmo 14, dove $\{e_1, \ldots, e_k\}$ è un generico insieme di valori. In questi due algoritmi, generalmente, le istruzioni $\mathcal{E}$ contengono la variabile i.

Algoritmo 14 - Controllo ciclico enumerativo

```
1: for i in {e1, ..., ek}
2:     istruzioni E
3: end for
```

5.11 Il teorema di Böhm-Jacopini

Una prima questione sulla programmazione strutturata riguarda l'eseguibilità dei tre controlli fondamentali, supponendo di disporre di un esecutore capace di eseguire gli algoritmi descritti con il linguaggio a salti di basso livello. Per dimostrare tale eseguibilità è sufficiente dimostrare che i tre controlli fondamentali sono esprimibili in termini del linguaggio di basso livello. La dimostrazione discende dalle seguenti considerazioni. Il controllo sequenziale è direttamente traducibile in quanto il flusso dell'esecuzione di un algoritmo descritto mediante il linguaggio elementare è proprio quello sequenziale. Il controllo condizionale **if** $\mathcal{C}$ **then** $\mathcal{E}_1$ **else** $\mathcal{E}_2$ **endif** può essere tradotto con

Algoritmo 15 - Traduzione a basso livello del controllo *if-then-else*

1: **if** $\neg\mathcal{C}$ **goto** 4
2: $\mathcal{E}_1$
3: **goto** 5
4: $\mathcal{E}_2$
5: ...

Il controllo ciclico **while** $\mathcal{C}$ **do** $\mathcal{E}$ **end while** può essere tradotto con

Algoritmo 16 - Traduzione a basso livello del controllo *while-do*

1: **if** $\neg\mathcal{C}$ **goto** 4
2: espressioni $\mathcal{E}$
3: **goto** 1
4: ...

La seconda questione che si pone riguarda il seguente quesito: *I controlli della programmazione strutturata sono sufficienti ad esprimere qualsiasi cosa che sia esprimibile in termini di un linguaggio "a salti") ?* La risposta (affermativa) a questo quesito venne data (e dimostrata) nel 1966 da due ricercatori italiani. Tale risultato è noto con il nome di

TEOREMA 5 (di Böhm-Jacopini)**.** Per ogni programma non strutturato (espresso mediante i controlli a salti) si può costruire un equivalente programma strutturato (espresso mediante i soli controlli della programmazione strutturata).

Il teorema di Böhm-Jacopini ha un significato molto profondo in quanto afferma che ogni algoritmo, espresso organizzando in un qualche modo delle azioni elementari di base, può essere riformulato equivalentemente mediante un algoritmo che contenga le azioni elementari di base organizzate tra loro mediante i soli tre controlli fondamentali previsti dal teorema. Questo risultato espresso dal teorema di Böhm-Jacopini si traduce nella possibilità di minimizzare un linguaggio algoritmico, senza limitarne la potenza espressiva. Il teorema costituisce, dunque, una legittimazione teorica alla programmazione strutturata ma non evidenzia perché siano preferibili i programmi strutturati rispetto a quelli non strutturati. Le motivazioni, che saranno ancora più evidenti quando sarà riconsiderata la metodologia top-down per lo sviluppo degli algoritmi, risiedono nel fatto che i tre controlli strutturati ricalcano il processo generativo dello sviluppo degli algoritmi. Notiamo che dal punto di vista teorico ci si potrebbe accontentare della forma debole del teorema, ossia della formulazione in cui al posto di *"è possibile costruire"* si sostituisca la locuzione *"esiste"*. Anche questo risultato sarebbe sufficiente a legittimare l'uso dei soli tre controlli fondamentali; ma la dimostrazione del teorema è di tipo costruttivo e fornisce anche la descrizione del metodo di traduzione.

5.12 Un confronto fra le diverse notazioni e linguaggi

In questo paragrafo descriveremo l'algoritmo del contadino russo per la moltiplicazione di due numeri naturali in diverse notazioni e linguaggi per fare una comparazione.

Notazione testuale di basso livello

Le notazioni testuali di basso livello vengono solitamente utilizzate per descrivere degli algoritmi che poi si intende tradurre in un linguaggio di programmazione di basso livello, vicino all'architettura fisica dei computer ed alle sue modalità operative.

Esempio 5.12.1 - L'algoritmo del contadino russo per il calcolo del prodotto fra due numeri naturali m ed n può essere espresso, in una forma di basso livello come descritto nell'algoritmo 17.

Algoritmo 17 - Algoritmo del contadino russo

Input: numeri naturali m e n
Output: prodotto fra m ed n
1: inizializza p a 0
2: **if** $m = 0$ **goto** 8
3: **if** m è pari **goto** 5
4: incrementa p di n
5: dimezza m
6: raddoppia n
7: **goto** 2
8: **return** p

L'algoritmo 17 può essere raffinato come riportato nell'algoritmo 18.

Algoritmo 18 - Algoritmo del contadino russo

Input: numeri naturali m e n
Output: prodotto fra m ed n
1: $p \leftarrow 0$
2: **if** $m = 0$ **goto** 11
3: $q \leftarrow m \div 2$
4: $p \leftarrow m * 2$
5: $r \leftarrow m - p$
6: **if** $r = 0$ **goto** 8
7: $p \leftarrow p + n$
8: $m \leftarrow m \div 2$
9: $n \leftarrow n * 2$
10: **goto** 2
11: **return** p

Notazione dei diagrammi a blocchi

A partire da una notazione a basso livello si può derivare in modo diretto il diagramma a blocchi corrispondente.

Esempio 5.12.2 - L'algoritmo del contadino russo (algoritmo 17) può essere descritto mediante il diagramma a blocchi riportato nella figura 5.9.

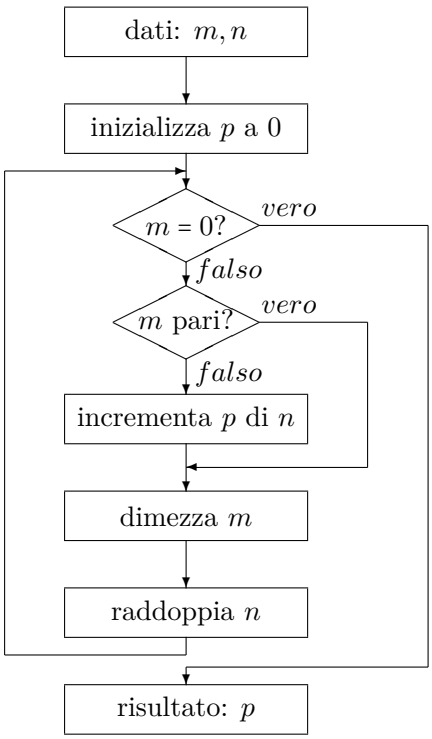


Figura 5.9: Diagramma a blocchi dell'algoritmo del contadino russo.

Notazione testuale di alto livello

Gli algoritmi vengono quasi sempre espressi in forma testuale ad alto livello in quanto offre il vantaggio di esprimere in modo chiaro l'idea della logica del procedimento ed inoltre risulta facilmente traducibile nella maggior parte dei linguaggi di programmazione moderni.

Esempio 5.12.3 - A seguire è riportato l'algoritmo del contadino russo, espresso in una forma di alto livello.

Algoritmo 19 - Moltiplicazione di due numeri naturali (algoritmo babilonese)

Input: numeri naturali m e n
Output: prodotto fra m e n

```
1: inizializza p a 0
2: while m ≠ 0 do
3:    if m è dispari then
4:       incrementa p di n
5:    end if
6:    dimezza m
7:    raddoppia n
8: end while
9: return p
```

Notazione grafica di un linguaggio visuale

Negli ultimi anni stanno sempre più prendendo piede degli ambienti di programmazione visuale (fra questi, *Scratch* e *Snap!*). Questi linguaggi hanno il pregio, specialmente per chi inizia a programmare, di suggerire la scrittura dei programmi, favorendo un approccio intuitivo e semplificato alla programmazione.

Esempio 5.12.4 - La figura che segue descrive un blocco in linguaggio Snap! che calcola il prodotto di due numeri naturali m ed n mediante l'algoritmo del contadino russo.

Notazione testuale di un linguaggio di programmazione

Per poter essere eseguito da uno specifico esecutore un algoritmo deve essere tradotto in un programma mediante un linguaggio di programmazione che non è però quasi mai il linguaggio della macchina esecutrice ma un linguaggio ausiliario aderente alla struttura ed al livello con cui è scritto l'algoritmo. Ciò per facilitare la traduzione (manuale) dell'algoritmo in un equivalente programma. Una volta ottenuto il programma in un qualche linguaggio di programmazione

vengono utilizzati appositi programmi di traduzione (compilatori ed interpreti) che traducono il programma in un formato comprensibile dalla macchina.

Dall'inizio dell'era dei computer, a partire dalla metà del secolo scorso, sono stati ideati migliaia di linguaggi di programmazione. La maggior parte sono di tipo testuale. A seguire un esempio.

Esempio 5.12.5 - La porzione di programma che segue descrive una funzione in linguaggio Python che calcola il prodotto di due numeri naturali m ed n mediante l'algoritmo del contadino russo.

```
def russo(m:int,n:int) -> int :
    p = 0
    while m != 0:
        if m%2 == 1:
            p += n
        m //= 2
        n *= 2
    return p
```

Osservazione. Il concetto di *livello di un linguaggio* è relativo e non ha senso parlare in termini assoluti di *alto livello* e *basso livello* quanto piuttosto in termini relativi, quali*livello x più alto/basso di ...*, *livello x adeguato al problema y*. Più che di una classificazione dicotomica, si tratta di una gradazione fine fra due poli estremi: da un lato le modalità operative della macchina e dall'altro le modalità di ragionamento dell'uomo. In altri termini il livello di un linguaggio specifica il grado di distanza del linguaggio dal modo di operare dell'esecutore. Per inciso osserviamo che il *livello più basso di linguaggio* è rappresentato dal linguaggio dell'esecutore che si dispone e che il *livello più alto* nella soluzione di un dato problema P può essere considerato quello costituito da frasi della forma *Risolvi il problema P*.

Osservazione. Le forme algoritmiche di basso livello, caratterizzate da istruzioni di salto, possono essere espresse, equivalentemente, in forme di alto livello con strutture di controllo più vicine al nostro modo di pensare. I due algoritmi 17 e 19, pur espressi con notazioni di diverso livello, descrivono le stesse operazioni. Dal loro confronto si evidenzia che la notazione strutturata di alto livello risulta molto più comprensibile, nella sua logica, rispetto alla corrispondente notazione a basso livello. Si evidenzia inoltre che, avendo a disposizione controlli delle azioni descritti mediante dei salti, lo sviluppo di algoritmi si presenta abbastanza difficoltoso in quanto ci obbliga, in veste di solutori, ad uno sforzo di traduzione dell'idea originale di soluzione per esprimerla nel linguaggio imposto, il quale risulta alquanto innaturale per l'uomo. La descrizione degli algoritmi mediante questi linguaggi è adeguata solo per piccoli algoritmi e mal si presta allo sviluppo di algoritmi complessi. Finché i problemi sono piccoli tale metodo può essere applicato con successo, mentre, quando i problemi sono più complessi e richiedono, quindi, degli algoritmi più complessi, si sente maggiormente la necessità di poter usare un linguaggio più evoluto e più aderente al modo di pensare dell'uomo.

5.13 Macchine e linguaggi

È consuetudine usare interscambievolmente le due locuzioni *Disponendo della macchina* $\mathcal{M}$ e *Usando il linguaggio* $\mathcal{L}$. Questa abitudine è coerentemente fondata sul fatto che un linguaggio definisce univocamente l'esecutore che comprende il dato linguaggio ed opera in base a quanto è descrivibile con il linguaggio da esso compreso. Viceversa, se una macchina $\mathcal{M}$ è in grado di eseguire dei programmi, questi dovranno necessariamente essere descritti nello specifico linguaggio $\mathcal{L}$ compreso dalla macchina. Si stabilisce così una corrispondenza biunivoca fra macchina e linguaggio. In base a questa corrispondenza, dato un linguaggio $\mathcal{L}$, si parla anche di *macchina* $\mathcal{L}$ e, data una macchina $\mathcal{M}$, si parla di *linguaggio* $\mathcal{M}$. Per denotare che una macchina $\mathcal{M}$ è in grado di comprendere il linguaggio $\mathcal{L}$ si scriverà $\mathcal{M}_{\mathcal{L}}$.

Definire il linguaggio di una macchina (o di un generico esecutore) significa descrivere gli oggetti che la macchina è in grado di gestire, stabilire le azioni che essa è in grado di eseguire su di essi ed il formato per richiamare le azioni.

Esempio 5.13.1 - Nelle costruzioni geometriche con *riga e compasso* si hanno a disposizione degli enti primitivi costituiti dai *punti* del piano e degli strumenti di disegno (matita, riga e compasso). A partire dagli enti primitivi si possono costruire altre entità geometriche quali segmenti, rette e circonferenze, con delle azioni della forma:

- tracciare il segmento congiungente due dati punti
- tracciare la retta passante per due dati punti
- tracciare la circonferenza di dato centro e punto di passaggio

Eseguendo delle *operazioni* sugli elementi geometrici (primitivi e non) si possono generare altri elementi con delle operazioni come le seguenti:

- selezionare un punto su un'entità
- determinare il punto di intersezione fra due rette
- determinare i punti di intersezione fra due circonferenze
- determinare i punti di intersezione fra una retta ed una circonferenza

Per descrivere dei procedimenti nel contesto delle costruzioni geometriche è necessario formalizzare mediante un linguaggio preciso tutte le precedenti operazioni.

5.14 Linguaggi e traduttori

In generale una macchina non comprende direttamente il linguaggio del solutore ma fra il solutore e la macchina vengono interposti degli opportuni interpreti che traducono alla macchina le direttive impartite dal solutore. Questo è proprio quello che avviene in realtà usando i calcolatori. Per una stessa macchina si hanno generalmente a disposizione più interpreti per colloquiare con la macchina, mediante diversi linguaggi di programmazione. Altrimenti si può pensare che l'interprete faccia parte della macchina la quale può essere pensata come una macchina che comprende il linguaggio del solutore tramite la mediazione dell'interprete. È facilmente riconoscibile che queste diverse impostazioni sono fra loro equivalenti; pertanto non dovremmo preoccuparci di quale stiamo adottando.

Per far comprendere i linguaggi alle macchine vengono utilizzati degli appositi programmi di traduzione (compilatori, interpreti, macroespansori, ...). La possibilità di *traducibilità automatica* da forme di alto livello ad equivalenti forme di basso livello, mediante degli schemi di traduzione o mediante appositi programmi di traduzione, consente ad un solutore di problemi di esprimere gli algoritmi in forme adeguate al problema in esame e di disinteressarsi della traduzione in forme più elementari effettivamente eseguibili dall'esecutore. Come strategia generale, un solutore di problemi dovrebbe pensare al più alto livello possibile, compatibilmente con il problema in questione, e di tradurre successivamente nel linguaggio dell'esecutore (virtuale) che si dispone. Questo approccio trova concretezza nella metodologia di sviluppo della soluzione di un problema nota come *metodologia top-down* o *metodologia dei raffinamenti successivi*.

Nei casi pratici l'esecutore è costituito da un elaboratore elettronico che è in grado di eseguire istruzioni in un *linguaggio macchina* $\mathcal{L}$ (ossia il linguaggio direttamente eseguibile dall'hardware della macchina); poiché tale linguaggio mal si presta (per l'uomo) a descrivere i programmi, risulta più comodo descrivere il programma $\mathcal{A}$ in un opportuno linguaggio di programmazione $\mathcal{L}'$ e si dota l'esecutore di un traduttore $T_{\mathcal{L}'\to\mathcal{L}}$ che traduce dal linguaggio $\mathcal{L}'$ al linguaggio $\mathcal{L}$ (figura 5.10).

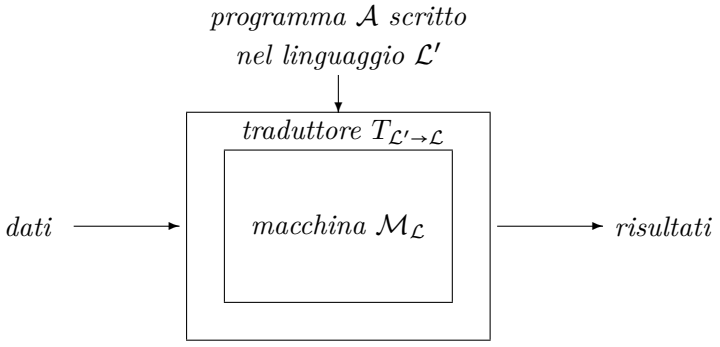


Figura 5.10: Schema di una macchina programmabile dotata di un traduttore.

Il blocco costituito dalla macchina $\mathcal{M}_{\mathcal{L}}$ e dal traduttore $T_{\mathcal{L}'\to\mathcal{L}}$ (parte costituita dal riquadro più esterno nella figura 5.10) diventa così una macchina $\mathcal{M}'_{\mathcal{L}'}$ capace di eseguire programmi scritti nel linguaggio $\mathcal{L}'$. In base all'equivalenza fra esecutori e linguaggi, le due locuzioni *Scrivere un traduttore del linguaggio* $\mathcal{L}'$ *al linguaggio* $\mathcal{L}$ e *Implementare la macchina virtuale* $\mathcal{M}_{\mathcal{L}'}$ sono equivalenti.

L'idea di dotare una macchina di un traduttore può essere generalizzata; ad esempio, disponendo di un traduttore $T_{\mathcal{L}"\to\mathcal{L}'}$ si può costruire una macchina $\mathcal{M}"_{\mathcal{L}"}$ secondo lo schema illustrato nella figura 5.11.

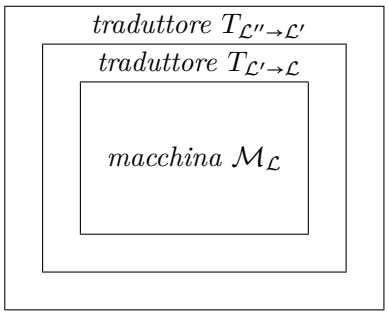


Figura 5.11: Struttura a strati di una macchina programmabile dotata di più traduttori.

Poichè ogni traduttore è anch'esso un programma, risulta legittima ed interessante la domanda "In quale linguaggio viene scritto il traduttore?". Ovviamente il traduttore $T_{\mathcal{L}'\to\mathcal{L}}$ del linguaggio $\mathcal{L}'$ non può essere scritto (completamente) in linguaggio $\mathcal{L}'$ poiché al momento dell'implementazione della macchina $\mathcal{M}_{\mathcal{L}}$ non si dispone della macchina $\mathcal{M}_{\mathcal{L}'}$. La soluzione adottata è quella di implementare la macchina virtuale $\mathcal{L}'$ su una macchina virtuale di livello più basso. L'implementazione di una macchina virtuale $\mathcal{M}_1$ su una macchina virtuale $\mathcal{M}_0$ di livello inferiore avviene, in generale, attraverso un processo di traduzione del linguaggio $\mathcal{L}_1$ della macchina $\mathcal{M}_1$ nel linguaggio $\mathcal{L}_0$ della macchina $\mathcal{M}_0$. In questo modo però il problema risulta solo rinviato. In generale, si ha una gerarchica lineare di macchine virtuali di livello più basso, ciascuna implementata sulla macchina virtuale direttamente sottostante. Questa gerarchia di macchine virtuali viene chiusa, al livello più basso, da una macchina reale fisica (composta di circuiti integrati e microprocessori vari) la quale interpreta ed esegue direttamente gli ordini impartiti dalla macchina di livello direttamente superiore.

Osservazione. Se le fasi di scrittura (utilizzando un linguaggio di programmazione), di traduzione ed uso di un programma vengono assommate e viste globalmente, la situazione che si evidenzia è quella di un utente che usa una macchina virtuale che comprende il dato linguaggio di programmazione utilizzato. Mediante una tale macchina virtuale, un solutore può descrivere un programma nel linguaggio della macchina virtuale e tale programma può essere eseguito dalla macchina virtuale. In questo caso il passo di traduzione dello specifico linguaggio del solutore al linguaggio macchina può essere ignorato dall'utente della macchina virtuale. Ad un opportuno livello di astrazione, per un utente dell'esecutore virtuale la distinzione fra macchina fisica e programma traduttore è del tutto trasparente: le risorse hardware (macchina fisica) e software (programmi) si configurano come un esecutore virtuale capace di comprendere il linguaggio ad alto livello e capace di eseguire le azioni elementari sugli oggetti elementari predisposti dal particolare ambiente di programmazione in cui ci si pone, ossia dal linguaggio utilizzato. Qui il termine *virtuale* è da intendere nel senso che è come se si disponesse di una macchina capace di capire il linguaggio $\mathcal{L}$ ad alto livello e capace di agire nell'ambiente fornito dal linguaggio di programmazione. A seconda del linguaggio $\mathcal{L}$ utilizzato si parla di macchina virtuale $\mathcal{L}$. Al posto del termine macchina virtuale si parla anche di *macchina astratta* volendo evidenziare il fatto che un utente della macchina può astrarre dalla particolare implementazione della macchina stessa.

ESERCIZI

5.1 Usando le forme di controllo di basso livello (salti condizionati ed incondizionati), risolvere i seguenti esercizi:

1. Dati due numeri, determinarne il massimo.
2. Dati tre numeri, determinare il massimo. Suggerimento: utilizzare la soluzione dell'esercizio precedente.
3. Dati tre numeri, determinare la somma dei due numeri più piccoli. Suggerimento: utilizzare la soluzione dell'esercizio precedente.
4. Dati tre numeri, determinare se essi possono costituire le lunghezze dei lati di un triangolo. Suggerimento: utilizzare le soluzioni degli esercizi precedenti.
5. Ordinare due numeri.
6. Ordinare tre numeri.
7. Date le misure dei lati di due rettangoli, stabilire se essi sono simili.
8. Dati quattro numeri, determinare se essi possono costituire i membri di una proporzione.
9. Date le misure dei lati di due rettangoli, stabilire se uno dei due può ricoprire completamente l'altro.

5.2 Risolvere i precedenti esercizi (esercizio 1.1) usando i controlli della programmazione strutturata.

5.3 Fra i seguenti problemi uno non è risolvibile con i mezzi finora visti. Individuare qual è. Risolvere gli altri. Acquisire in input ciclicamente dei numeri, terminando la lettura quando viene letto il numero 0. Terminata la fase di lettura, dare in output le seguenti quantità:

1. il massimo
2. il minimo ed il massimo
3. il numero e la somma dei numeri letti
4. la somma dei numeri pari
5. la media dei valori
6. i due valori più grandi
7. il valore più vicino alla media
8. decidere se la sequenza è ordinata decrescentemente

5.4 Descrivere in notazione di basso ed alto livello un algoritmo per calcolare la somma dei numeri naturali compresi fra due dati numeri naturali.

5.5 Esprimere in notazione algoritmica, utilizzando i controlli della programmazione strutturata, la seguente porzione di ricetta: *Dopo aver preparato l'impasto ed averlo messo in una casseruola, cuocere per* 30 *minuti mescolando per un minuto ogni* 5 *minuti, aggiungendo un cucchiaio d'acqua all'occorrenza.*

5.6 Tradurre le seguenti porzioni di algoritmo mediante i controlli della programmazione strutturata:

1. 1: **if** $\mathcal{C}$ **goto** 4
   2: $\mathcal{E}$
   3: **goto** 1
   4: ...

2. 1: **if** $\mathcal{C}_1$ **goto** 4
   2: $\mathcal{E}$
   3: **if** $\mathcal{C}_2$ **goto** 2
   4: ...

3. 1: **if** $\mathcal{C}_1$ **goto** 4
   2: $\mathcal{E}$
   3: **if** $\mathcal{C}_2$ **goto** 1
   4: ...

4. 1: **if** $\mathcal{C}_1$ **goto** 3
   2: $\mathcal{E}$
   3: **if** $\mathcal{C}_2$ **goto** 2

5. 1: $\mathcal{E}_1$
   2: **if** $\mathcal{C}_1$ **goto** 1
   3: **if** $\mathcal{C}_2$ **goto** 6
   4: $\mathcal{E}_2$
   5: **goto** 1
   6: ...

6. 1: $\mathcal{E}_1$
   2: **if** $\mathcal{C}_1$ **goto** 5
   3: **if** $\mathcal{C}_2$ **goto** 6
   4: $\mathcal{E}_2$
   5: **goto** 1
   6: ...

5.7 Tradurre il seguente schema di controllo in notazione di basso livello:

1: **while** $\mathcal{C}_1$ **do**
2: **if** $\mathcal{C}_2$ **then**
3: $\mathcal{E}_1$
4: **endif**
5: $\mathcal{E}_2$
6: **endwhile**

5.8 Tradurre i seguenti schemi di controllo condizionali, usando i controlli di basso livello ed i controlli fondamentali della programmazione strutturata, senza ricorrere all'uso di operatori logici:

1. 1: **if** $\mathcal{C}_1 \vee \mathcal{C}_2$ **then**
   2: $\mathcal{E}_1$
   3: **else**
   4: $\mathcal{E}_2$
   5: **end if**

2.
1: **if** $\mathcal{C}_1 \wedge \mathcal{C}_2$ **then**
2: $\quad \mathcal{E}_1$
3: **else**
4: $\quad \mathcal{E}_2$
5: **end if**

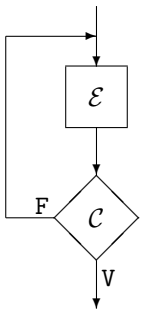
3.
1: **if** $\mathcal{C}_1 \wedge (\mathcal{C}_2 \vee \mathcal{C}_3)$ **then**
2: $\quad \mathcal{E}_1$
3: **else**
4: $\quad \mathcal{E}_2$
5: **end if**

5.9 Tradurre i seguenti schemi di controllo ciclici in notazione di basso livello ed usando solo i controlli fondamentali della programmazione strutturata (k, k_1 e k_2 sono delle generiche espressioni intere ed $\mathcal{E}$ un generico blocco di istruzioni):

1.
1: **for** k **times**
2: $\quad \mathcal{E}$
3: **end for**

2.
1: **for** i **from** k_1 **to** k_2
2: $\quad \mathcal{E}$
3: **end for**

5.10 Uno schema di controllo spesso ricorrente è costituito da un controllo ciclico postcondizionato descritto dal seguente diagramma a blocchi e dalla notazione testuale a fianco:



repeat
$\mathcal{E}$
until $\mathcal{C}$

Descrivere questo controllo in notazione a basso livello e come combinazione dei controlli della programmazione strutturata.

5.11 In alcune situazioni si ricorre ad uno schema di controllo ciclico della forma **loop** ... **end loop** che esegue indefinitamente un blocco di azioni. Nella sua forma base questo controllo risulta potenziamlmente infinito. Per terminare il ciclo infinito innescato dal controllo **loop** è possibile usare, all'interno del ciclo, il comando **exit**, inserito in un controllo condizionale della forma **if** $\mathcal{C}$ **then exit endif**, dove $\mathcal{C}$ è una generica condizione. A seguire sono riportate alcune tipiche situazioni:

1.
```
1: loop
2:     E
3: end loop
```

2.
```
1: loop
2:     E1
3:     if C then
4:         exit
5:     end if
6:     E2
7: end loop
```

3.
```
1: loop
2:     E1
3:     if C1 then
4:         exit
5:     end if
6:     E2
7:     if C2 then
8:         exit
9:     end if
10: end loop
```

Tradurre i precedenti schemi di controllo ciclici in notazione di basso livello, mediante la notazione dei diagrammi a blocchi ed usando i controlli fondamentali della programmazione strutturata.

5.12 Usando solo i controlli fondamentali della programmazione strutturata, scrivere una porzione di algoritmo in grado di generare i seguenti processi:

$P_1 \equiv [\,]$ (processo nullo)

$P_2 \equiv [\mathcal{E}_1]$

$P_3 \equiv [\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_1, \mathcal{E}_1, \mathcal{E}_2]$

$P_4 \equiv [\mathcal{E}_1, \mathcal{E}_1, \mathcal{E}_1, \mathcal{E}_1]$

$P_5 \equiv [\mathcal{E}_1, \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_2, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_2, \mathcal{E}_1]$

6

DENOTARE

Irritato da questa confusione, cercai di orientarmi ricorrendo a uno degli stratagemmi prediletti dai filosofi: cominciai ad assegnare un nome alle cose.

D. C. Dennet, *Brainstorm*

La nascita della cultura può essere fatta partire nel momento in cui l'uomo ha iniziato a descrivere i fatti, le conoscenze e le emozioni, sulle pareti delle caverne, mediante disegni, grafiti ed incisioni. Similmente, nello sviluppo cognitivo di un bambino un passo significativo è rappresentato dal momento in cui il bambino inizia a chiamare per nome le persone e le cose che gli stanno intorno. Questo stratagemma di dare un nome alle cose, oltre che essere a fondamento di molti concetti filosofici, è uno strumento pragmatico caratteristico dei linguaggi di programmazione e permette di chiamare per nome quei dati che, sotto-sotto, nella memoria del computer, sono sequenze di 0 e 1.

6.1 Segno, significato e verità

Il rapporto fra *segno*, *significato* e *verità* costituisce una delle problematiche cruciali dell'epistemologia e della filosofia ed ha avuto una particolare riconsiderazione ed analisi in connessione all'elaborazione mediante computer. Distinguiamo, mediante un esempio, tre distinti livelli di analisi, corrispondenti ciascuno ai tre diversi concetti sopra elencati; consideriamo la *scrittura*

```
2+2=5
```

Al livello dei *segni*, quanto scritto nella riga precedente può essere analizzato e portare a delle affermazioni come le seguenti:

1. La stringa è composta da 5 caratteri (si noti che il 5 che qui viene utilizzato rappresenta un *numero* e costituisce qualcosa di completamente diverso dal *segno* `5` che compare nella stringa).
2. Nella stringa compare il carattere `+` (sarebbe sbagliato, in questo contesto, chiamare tale segno con il termine (*operatore di*) *addizione*).
3. Il segno `2` compare 2 volte (si noti qui l'artificio tipografico di utilizzare font diversi per differenziare i diversi piani descrittivi).

Indagando il *significato*, ossia interpretando i segni, la stringa può portare a delle affermazioni della forma:

1. La frase esprime il confronto fra due espressioni numeriche.
2. Nella frase compare un'operazione di *addizione* fra *due* numeri.
3. Il numero 2 compare 2 volte.
4. I due 2 che compaiono nell'affermazione precedente hanno significati diversi.
5. Le stringhe `due` e `2` riportate nella frase precedente hanno *due* diversi significati.

Un'analisi più approfondita, che ci richiede uno sforzo di *elaborazione*, ci fa concludere che *La frase è falsa.*

In questo particolare esempio il livello di indagine relativo al significato ed il livello relativo all'indagine di verità, essendo entrambi immediati, si confondono; in realtà si tratta di due livelli ben distinti, come si può intuire dalla seguente frase (scritta nell'usuale linguaggio matematico):

$$\exists\, a, b, c, n \in \mathbb{N}, n > 2 : a^n + b^n = c^n$$

Si tratta del famoso *Ultimo teorema di Fermat.* In questo caso un calcolo diretto non porterebbe ad alcuna conclusione; fidandoci di quello che ha scritto Wiles nel 1994, possiamo dire che la frase è *falsa.*

La distinzione fra *segno* e *significato* richiede, sul piano descrittivo, una corrispondente distinzione fra il *piano degli oggetti descritti* ed il *piano del testo che descrive gli oggetti ed il loro significato.* A livello tipografico la distinzione

fra questi due piani può essere fatta racchiudendo fra apici la stringa denotata, come ad esempio nella seguente scrittura:

"tre" è una stringa composta da tre caratteri.

Tipograficamente si può ricorrere all'uso di font diversi, come nella scrittura che segue:

`110100` è la codifica del numero 100 in base 2.

Adottando questa convenzione, con `x+3` denoteremo una stringa di 3 caratteri, mentre con $x + 3$ si denoteremo un binomio.

Osservazione. Certe volte si è indotti a lavorare sui segni, guidati solamente dalle regole formali, disinteressandosi del *significato* che sta sotto ai *segni*, confidando che le elaborazioni formali portino a conclusioni sensate. Fu proprio con questa fiducia in animo che gli algebristi del XVI secolo iniziarono a lavorare su degli strani segni della forma $\sqrt{-a}$, con a numero reale positivo, sostituendo scritture della forma $(\sqrt{-a})^2$ con $i\,a$. I loro tentativi sono stati coronati da successo dando vita ai numeri complessi, un sistema funzionale e produttivo che ha trovato molteplici applicazioni in settori all'apparenza lontani, dalla matematica alla fisica ed all'elettrotecnica.

6.2 La codifica dei numeri

Una volta stabilito il meccanismo di conteggio mediante una corrispondenza biunivoca del tipo "tante tacche - altrettante pecore", subentra un problema di ordine pratico: diventa difficile rappresentare una quantità elevata di pecore. La soluzione che è stata adottata consiste nel assegnare *significati* diversi ai *simboli* che vengono utilizzati; ad esempio, come fecero i romani oltre duemila anni fa, il simbolo `I` per denotare le *unità*, il simbolo `X` per denotare le *decine*, il simbolo `C` per denotare le centinaia*cento* pecore e similmente per le altre cifre romane `V`, `L`, `D` ed `M`. Inoltre, sempre nella notazione dei romani, il significato dei simboli dipende anche dal contesto; ad esempio, il simbolo `I` ha significati diversi nelle due stringhe `IX` e `XI`: nel primo caso denota un'unità che viene sottratta alla decina mentre nel secondo caso l'unità viene aggiunta alla decina.

Agli inizi del secondo millenio iniziarono a diffondersi in Europa, attraverso delle traduzioni in latino, molte opere arabe; fra queste ebbe particolare successo l'opera del matematico ed astronomo arabo al-Khuwarizmi. L'italiano Leonardo Pisano (Pisa, 1175 - Pisa, 1235), più noto con il nome di Fibonacci, ebbe modo di accedere a una traduzione in latino dell'opera di al-Khuwarizmi e, nel suo *Liber abaci*, presentò e discusse problemi e metodi algebrici della cultura araba, e, in particolare, si fece sostenitore dell'uso delle cifre indo-arabiche. L'abbandono del vecchio sistema numerico romano avvenne però molto lentamente. Nel Medioevo il sistema di numerazione posizionale in base 10 conobbe forti ostilità tra i tradizionalisti, estimatori dell'abaco, i cosiddetti *abacisti* che calcolavano con l'abaco; a questi si contrapposero gli *algoristi* che calcolavano usando le nuove cifre arabe. Per molti secoli vi fu una forte competizione tra abacisti e algoristi rappresentata nella figura 6.1. Il confronto si concluse solo nel Cinquecento con la vittoria degli algoristi: superate le diffidenze nei

confronti dei nuovi numeri, lo zero e il sistema di numerazione indo-arabico diventarono popolari anche al di fuori degli ambienti matematici.

Figura 6.1: Nella illustrazione, tratta dall'opera enciclopedica *Margarita Philosophica*, un libro tedesco di introduzione all'aritmetica scritto da Gregor Reisch nel 1503, è rappresentata la contrapposizione fra i due modi di calcolare entrambi ancora usati in quel periodo. L'Aritmetica, una donna in piedi, ha tra le mani due libri per i due diversi sistemi di calcolo; a destra Pitagora che utilizza il tavolo con i gettoni mentre a sinistra Boezio utilizza le cifre arabe per i suoi calcoli scritti.

6.3 Sistemi di numerazione

I metodi usati per la rappresentazione dei numeri sono passati attraverso varie fasi durante le quali i simboli usati ed il metodo di rappresentazione simbolica sono cambiati.

Le forme più primitive sono iconiche in quanto rimandano ad un'immagine o ad un oggetto. Alcune fra queste sono:

- costituire un insieme di cardinalità pari al numero che si vuole rappresentare, utilizzando piccoli oggetti concreti (sassolini, conchiglie, semi, ...)
- intagliare in un oggetto (osso, legno) un numero di segni pari alla quantità che si vuole rappresentare
- utilizzare parti del proprio corpo (ad esempio le dita delle mani)

Contrapposte alle rappresentazioni iconiche ci sono quelle simboliche basate su un sistema di segni più o meno astratti. Il sistema di questo tipo più semplice ed intuitivo è il *sistema unario* che si basa sull'uso di un unico simbolo, per denotare l'unità e, con la ripetizione di questo segno, ogni altro numero.

A partire dal sistema unario si sono sviluppati numerosi altri tipi di rappresentazione simbolica. Fra questi si distinguono due grandi famiglie: i sistemi additivi e i sistemi posizionali.

- nei *sistemi additivi*, ogni numero viene indicato accostando una serie di cifre fino a quando la somma dei numeri corrispondenti alle diverse cifre pari al numero che si vuole rappresentare; ad esempio il numero *ventitre* viene denotato con la sequenza di simboli `XXIII`, corrispondentemente all'identità

$$10 + 10 + 1 + 1 + 1 = 23$$

- nei *sistemi posizionali*, la posizione delle diverse cifre nel numero è fondamentale. Viene scelta una base e vengono definite una serie di cifre che indicano tutti i numeri naturali più piccoli della base, compreso lo zero. Tutti gli altri numeri vengono espressi in funzione di potenze della base. Ad esempio, il numero *ventitre* viene denotato con la sequenza di simboli `23`, corrispondentemente all'identità

$$2 * 10 + 3 * 1 = 23$$

  Nella pratica sono frequentemente utilizzati il *sistema binario* ed il *sistema decimale*.

6.4 Tipologie dei valori

Nella corso della sua evoluzione l'uomo ha utilizzato dapprima i numeri naturali e successivamente, sollecitato dalla necessità di fare operazioni più avanzate, ha ideato, nell'arco di molti secoli, i numeri frazionari, i numeri reali, i numeri interi, i numeri complessi ed altri ancora. Nel XIX secolo, accanto ai numeri, fecero capolino ad opera di George Boole, nel contesto di un'algebra ben definita, i valori logici *falso* e *vero*, tradizionalmente denotati con `FALSE` e `TRUE`. Successivamente, specialmente in ambito informatico, si iniziò ad utilizzare caratteri, stringhe, testi con la disinvoltura dei valori numerici e per queste nuove tipologie di valori si definirono delle specifiche operazioni.

Un insieme di valori è strettamente collegato alle operazioni che si possono applicare su di essi. Questo connubio di valori ed operazioni assume il nome di *tipo di dato*. I linguaggi di programmazione predispongono generalmente i seguenti tipi di dato: *numeri* (classificati in *naturali*, *interi*, *razionali*), *valori logici*, *caratteri*, *stringhe*. Su questi tipi di base sono predisposte delle operazioni che producono, come risultato, altri valori. Nella tabella 6.1 sono riportati i tipi di dato elementari che si ritrovano negli usuali linguaggi di programmazione, con l'indicazione di alcune operazioni. Oltre a queste, sui tipi di dato riportati nella tabella 6.1, si possono applicare le operazioni di confronto: $=$, $\neq$, $<$, $\leq$, $>$, $\geq$.

Simbolo	Descrizione	Valori	Operazioni
$\mathbb{N}$	numeri naturali	$0, 1, 2, \ldots$	$+$, $-$, $*$, $\div$, mod
$\mathbb{Z}$	numeri interi	$\ldots, {}^{-}2, {}^{-}1, 0, 1, 2, \ldots$	$+$, $-$, $*$, $\div$, mod
$\mathbb{D}$	numeri decimali	$7.4, 419., 3.1415, \ldots$	$+$, $-$, $*$, $/$
$\mathbb{B}$	valori logici	`FALSE`, `TRUE`	$\wedge$, $\vee$, $\neg$
$\mathbb{A}$	caratteri alfabetici	$'a', 'b', '0', '*', \ldots$	
$\mathbb{S}$	stringhe di caratteri	$''lato'', ''due'', \ldots$	

Tabella 6.1: Tabella dei tipi di dato elementari.

Nei linguaggi di programmazione di tipo visuale il tipo dei valori viene evidenziato in modo grafico con una particolare tipologia di forma (eventualmente colorata) contenente il dato valore.

Esempio 6.4.1 - Nei linguaggi Scratch e Snap! i valori numerici vengono rappresentati in un ovale, i valori booleani *vero* e *falso* in un esagono e le stringhe in rettangoli, come esemplificato nella figura 6.2.

Figura 6.2: Rappresentazione dei valori numerici, logici e stringhe nei linguaggi Scratch e Snap!.

6.5 Valori ed indirizzi

Nella memoria del computer sono localizzati i *dati* e le *istruzioni*. I dati possono essere classificati come segue:

- *valori*: dati che rappresentano informazioni rilevanti per l'applicazione dell'utente (dati del problema, dati intermedi ottenuti nel processo di elaborazione, risultati, ...)
- *indirizzi*: dati che individuano una posizione in memoria dove sono memorizzati dati o istruzioni. Gli indirizzi sono generalmente accessibili solo nei linguaggi di programmazione di basso livello; nei linguaggi di alto livello vengono mascherati mediante il concetto di *riferimento*.

6.6 Denotazione degli oggetti

Nello sviluppo del pensiero scientifico un decisivo passo verso un più alto livello di astrazione è consistito nel denotare gli oggetti reali o astratti (numeri, entità geometriche, ...) mediante dei nomi simbolici. Questa idea è stata la base sulla quale si è sviluppata l'algebra. Anche nei linguaggi di programmazione un significativo passo evolutivo è rappresentato dalla possibilità di denotare i dati e le istruzioni mediante dei nomi identificativi costituiti da sequenze di caratteri, detti *identificatori*. Questo artificio ha portato ad una notevole semplificazione nella scrittura dei programmi.

Per denotare un oggetto con un identificatore si utilizzerà la notazione (detta *assegnazione*)

$$identificatore \leftarrow oggetto$$

Per indicare il fatto che un identificatore è associato con un determinato oggetto, si usa scrivere

$$(identificatore, oggetto)$$

mentre per indicare che un identificatore è indefinito, ossia che non è associato ad alcun oggetto, si usa scrivere

$$(identificatore, \perp)$$

A seconda che l'associazione fra identificatore ed oggetto sia statica (non ridefinibile) o dinamica (ridefinibile) il nome che denota l'oggetto è detto rispettivamente *costante* o *variabile*.

Esempio 6.6.1 - Negli usuali linguaggi di programmazione gli identificatori vengono costruiti rispettando le seguenti regole sintattiche: il primo carattere deve essere alfabetico (`a`,...,`z`,`A`,...,`Z`), i caratteri successivi al primo possono essere alfabetici, numerici (`0`,...,`9`) oppure alcuni caratteri speciali (`_`). Ad esempio sono corretti i seguenti identificatori: `altezza_triangolo`, `numeroPrimo`, `x1`, `prodotto` mentre non lo sono i seguenti: `questo_nome_non_e'_corretto`, `neanche questo`, `2a`, `1720bis`, `a7/b`. Viene inoltre adottata la convenzione che gli identificatori delle costanti siano composti da caratteri maiuscoli mentre gli identificatori delle variabili inizino con una lettera minuscola. Ad esempio, sono identificatori di costante i seguenti: `PIGRECO`, `FALSE` e nomi di variabili i seguenti: `somma` e `massimoFraXedY`.

6.7 Variabili e riferimenti

Le variabili costituiscono uno dei concetti fondamentali dei linguaggi di programmazione a paradigma imperativo. Una variabile identifica una zona di memoria dove è memorizzato un valore. Con la locuzione *variabile* x oppure *valore della variabile* x si denota il valore contenuto nella variabile x, mentre con la locuzione *indirizzo della variabile* x si denota l'indirizzo della posizione di memoria dove è localizzato il valore individuato dalla variabile x. Una variabile contenente un indirizzo di memoria (dove è memorizzato un valore, un indirizzo, un oggetto, un'istruzione) viene detta *puntatore*.

Nel contesto della programmazione a paradigma imperativo risulta coerente e comodo considerare una variabile come un contenitore, localizzato in una zona di memoria interna al calcolatore, contenente un valore che può essere modificato durante le varie fasi di elaborazione. Per poter essere manipolate le variabili vengono denotate mediante un identificatore. Per questo è invalsa la significativa abitudine di descrivere l'associazione fra un identificatore ed un valore mediante un contenitore avente il nome dell'identificatore e contenente il valore ad esso associato, come illustrato nella figura 6.3.

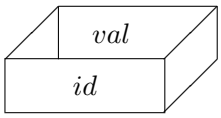


Figura 6.3: Variabile *id* contenente il valore *val*.

La situazione rappresentata graficamente nella figura 6.3 viene più semplicemente descritta come riportato nella figura 6.4.

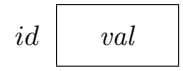


Figura 6.4: Variabile *id* contenente il valore *val*.

In alcuni linguaggi visuali (Scratch e Snap!) per descrivere una variabile, ossia l'associazione fra un identificatore *id* ed un valore *val*, si usa la notazione riportata nella figura 6.5.

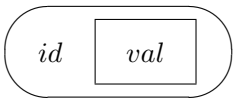


Figura 6.5: Variabile *id* contenente il valore *val*.

Gli *oggetti* sono le entità tipiche dei linguaggi orientati agli oggetti. Vengono creati dinamicamente durante l'esecuzione del programma e vengono denotati mediante degli identificatori detti *riferimenti* ossia mediante nomi di variabili contenenti l'indirizzo di memoria di dove è localizzato l'oggetto; la situazione viene descritta graficamente come riportato in figura 6.6.

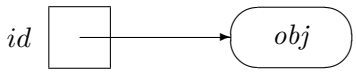


Figura 6.6: Riferimento *id* che referenzia l'oggetto *obj*.

Dal punto di vista operativo, nei linguaggi di programmazione possiamo assumere la seguente caratterizzazione:

- una *variabile* contiene un valore
- un *riferimento* denota un oggetto

6.8 Notazioni delle operazioni

Le operazioni vengono denotate con vari formalismi, di tipo testuale e di tipo grafico. Per talune notazioni è necessario ricorrere a delle parentesi tonde per precisare gli argomenti sui quali opera ciascun operatore e per imporre la corretta priorità di valutazione. A seguire sono descritte alcune fra le notazioni più frequentemente utilizzate.

Notazioni testuali

Nelle notazioni testuali le operazioni vengono denotate mediante un simbolo oppure mediante un nome identificativo.

Se l'operazione coinvolge un solo operando, il segno di operazione viene messo prima dell'operando (***notazione prefissa***; ad esempio: operazione di *cambia segno*: $-x$) oppure dopo dell'operando (***notazione postfissa***; esempio: operazione di *fattoriale*: $n!$).

Nel caso in cui l'operazione coinvolga due operandi, viene frequentemente utilizzata la ***notazione infissa*** in cui il segno di operazione viene scritto fra i due operandi su cui si esercita l'operazione. Indicando con $\circ$ un generico operatore e con a e b due operandi l'operazione si esprime nella forma

$$a \circ b$$

La ***notazione polacca inversa*** (*RPN*, *R*everse *P*olish *N*otation) deve il suo nome al matematico e filosofo Jan Lukasiewicz che la utilizzò per la prima volta intorno all'anno 1920 per descrivere il calcolo proposizionale. Sebbene non sia più utilizzata nell'ambito della logica, la notazione ha con il tempo acquisito una importanza in informatica, per denotare le espressioni matematiche. In questa notazione gli operatori vengono scritti dopo gli operandi sui quali agiscono:

$$a\ b\ \circ$$

Nella ***notazione Lisp***, caratteristica del linguaggio omonimo, il segno di operazione $\circ$ e gli operandi $a_1 \ldots a_n$) vengono inseriti in una lista:

$$(\circ\ a_1 \ldots a_n)$$

La ***notazione funzionale***, che deriva dalla classica notazione matematica usata per denotare una funzione, costituisce una delle forme più usate per descrivere un'operazione: indicando con *op* il nome identificativo dell'operazione e con $a_1, \ldots, a_n$ gli argomenti dell'operazione, l'operazione viene indicata con la notazione testuale

$$op\,(a_1, \ldots, a_n)$$

La ***notazione puntata***, tipica della *programmazione orientata agli oggetti*, è caratterizzata dalla sintassi:

$$a.op\,(a_1, \ldots, a_n)$$

dove a è l'oggetto sul quale viene riferita l'operazione *op* ed $a_1, \ldots, a_n$ sono gli argomenti dell'operazione.

Esempio 6.8.1 - A seguire è riportata l'addizione fra due generici numeri a ed b nelle diverse notazioni sopra descritte.

$a + b$	notazione algebrica
$a\ b\ +$	notazione polacca inversa
$(+\ a\ b)$	notazione Lisp
$add(a, b)$	notazione funzionale
$a.add(b)$	notazione puntata

Esempio 6.8.2 - Nella seguente tabella sono indicate alcune notazioni corrispondenti all'espressione algebrica $a + b * c + d$ composta con operatori binari infissi.

$a\ b\ c * +\ d\ +$	notazione polacca inversa
$(+\ a\ (*\ b\ c)\ d)$	notazione Lisp
$add(add(a, mul(b, c)), d)$	notazione funzionale
$a.add(b.mul(c)\,).add(d)$	notazione puntata

Esempio 6.8.3 - La notazione RPN ha il pregio di evitare l'uso di parentesi e i problemi e convenzioni legate alla precedenza degli operatori. Ad esempio l'espressione algebrica $7 - (9 - 3)/(8 - 3 * 2) + 5$ in notazione RPN si scrive

$$7 \ 9 \ 3 \ - \ 8 \ 3 \ 2 \ * \ - \ / \ - \ 5 \ +$$

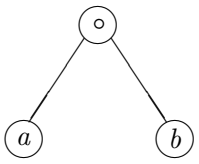
Per migliorare la leggibilità e l'espressività delle scritture, viene talvolta usata una notazione in cui il simbolo di operatore viene suddiviso in più parti e distribuito fra gli argomenti (***notazione distribuita***), come illustrato nei casi riportati nell'esempio che segue.

Esempio 6.8.4 - A seguire sono riportate alcune operazioni in notazione distribuita:

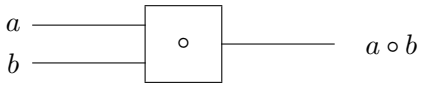
$\|x\|$	operazione unaria di *valore assoluto*
$a[3]$	operazione binaria di *accesso agli elementi di un array*
resto di x diviso y	operazione binaria di *resto della divisione intera*
$x \equiv y \ (\mod n)$	operazione ternaria di *equivalenza modulo n*

Notazioni grafiche

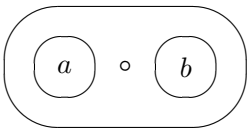
Per descrivere la struttura sintattica di un'espressione viene spesso utilizzata la ***notazione ad albero***, dove nella radice (in alto) viene posto il simbolo dell'operazione e nei nodi figli (in basso) vengono indicati gli operandi. La figura che segue denota una generica operazione $\circ$ fra due operandi a e b.



Una notazione alternativa consiste nel rappresentare un'operazione mediante un blocco avente come ingressi i dati e come uscite i risultati; poiché i dati fluiscono attraverso dei fili connessi al blocco, tale notazione viene detta ***notazione data-flow***. Con questa notazione l'operazione $\circ$ fra a e b viene denotata mediante il seguente schema:



In alcuni linguaggi di programmazione visuali (Scratch, Snap!) un'operazione viene descritta in modo grafico mediante una ***notazione a blocchi***. Con questa notazione l'operazione $\circ$ fra a e b viene denotata mediante il seguente schema:



Esempio 6.8.5 - Adottando le notazioni descritte sopra, l'espressione aritmetica $(a + b)\, c$, può essere denotata come descritto nella figura 6.7.

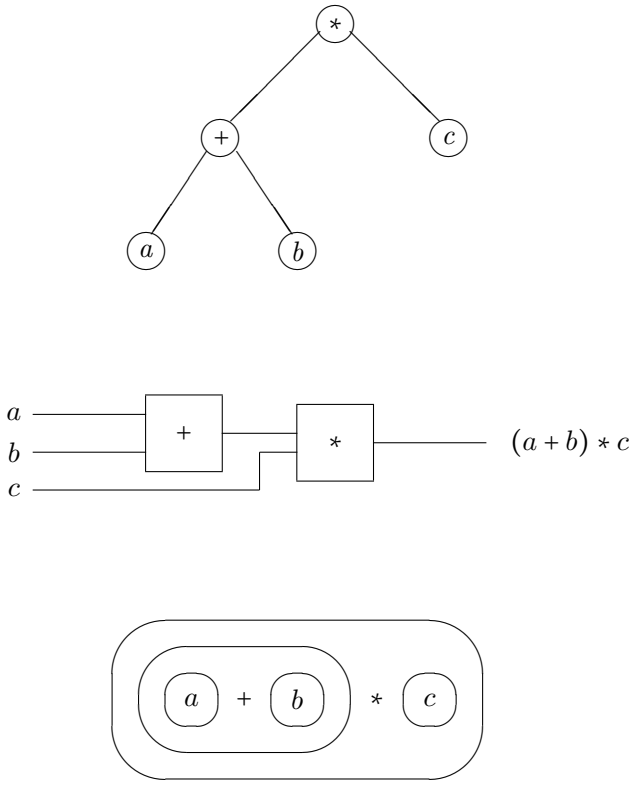


Figura 6.7: Diverse notazioni dell'espressione aritmetica $(a + b)\, c$: *notazione notazione ad albero, notazione data-flow, notazione a blocchi.*

Notazioni tabellari

Nel caso in cui l'insieme di elementi sui quali si svolge l'operazione abbia una cardinalità finita ed in particolare quando sia costituito da pochi elementi, un formalismo spesso utilizzato per descrivere un'operazione binaria è consiste in una tabella bidimensionale in cui gli elementi rappresentano il risultato fra l'elemento di testa delle righe e delle colonne.

Esempio 6.8.6 - La seguente tabella descrive l'operazione di addizione fra i primi 5 numeri naturali.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	5
2	2	3	4	5	6
3	3	4	5	6	7
4	4	5	6	7	8

Esempio 6.8.7 - Un'operazione non binaria viene solitamente descritta mediante una tabella bidimensionale composta da tante colonne quanti sono complessivamente gli insiemi costituenti il dominio ed il codominio. A seguire è riportato l'esempio relativo all'operazione di minimo e massimo fra alcune terne di numeri.

a	b	c	$min(a,b,c)$	$max(a,b,c)$
4	1	2	1	4
3	4	6	3	6
1	3	1	1	3
2	4	3	2	4
7	3	5	3	7

6.9 Dare un nome agli algoritmi

Come ogni altra cosa, anche i procedimenti possono essere denotati e richiamati mediante un nome identificativo. Poiché gli algoritmi sono caratterizzati dai dati di input, nella denotazione si conviene di indicare anche la lista dei nomi dei dati, detta lista degli *argomenti* (o *parametri*) *formali*. Al momento dell'uso dell'algoritmo su una specifica istanza di problema, vengono indicati, in corrispondenza della lista degli argomenti formali, la lista degli *argomenti attuali* che specificano la particolare istanza di problema da risolvere. Dare un nome agli algoritmi risulta fondamentale nell'adozione della metodologia top-down e bottom-up, specialmente, ma non solo, quando l'algoritmo debba essere utilizzato in più punti. Risulta fondamentale anche nella tecnica ricorsiva. Come vantaggio collaterale, dare un nome ad un algoritmo fornisce un meccanismo sintattico che permette di astrarre come l'algoritmo opera al suo interno.

Esempio 6.9.1 - L'algoritmo per il calcolo del massimo comune divisore fra due numeri naturali può essere denotato con $mcd(m,n)$. Quando questo algoritmo viene utilizzato verranno specificati i valori degli argomenti attuali, ad esempio $mcd(12,54)$.

6.10 Alberi di operazioni binarie

Consideriamo un'operazione binaria associativa $\circ : (x_1, x_2) \mapsto y$ che denoteremo in modo grafico come descritto nella figura 6.8.

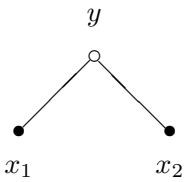


Figura 6.8: Schema di un generico operatore binario che agli ingressi x_1, x_2 associa l'uscita y.

Consideriamo un'espressione composta da più operazioni della forma

$$x_1 \circ x_2 \circ \cdots \circ x_n$$

La proprietà di associatività permette di costruire alberi equivalenti ma di diversa topologia; queste diverse topologie di alberi presentano efficienze significativamente diverse nel caso di esecuzione parallela.

In queste ipotesi l'operatore su più operandi $x_1, x_2, \ldots, x_n$ può essere denotato mediante un albero come descritto nella figura 6.9, corrispondente alla seguente scrittura associativa a sinistra:

$$((((((x_1 \circ x_2) \circ x_3) \circ x_4) \circ x_5) \circ x_6) \circ x_7) \circ x_8$$

Un albero equivalente può essere ottenuto organizzando gli operandi con una parentizzazione bilanciata, come descritto nella nella figura 6.10, corrispondente alle scrittura bilanciata:

$$((x_1 \circ x_2) \circ (x_3 \circ x_4)) \circ ((x_5 \circ x_6) \circ (x_7 \circ x_8))$$

Gli alberi descritti nelle figure 6.9 e 6.10 sono topologicamente diversi ma equivalenti dal punto di vista computazionale. Questi due alberi hanno la stessa efficienza nel caso di esecuzione sequenziale ma manifestano, invece, un'efficienza sostanzialmente diversa nel caso di esecuzione parallela: nell'ipotesi che ciascun operatore impieghi un tempo di esecuzione unitario, il primo albero impiega un tempo pari a n mentre il secondo impiega un tempo pari a $\log_2 n$.

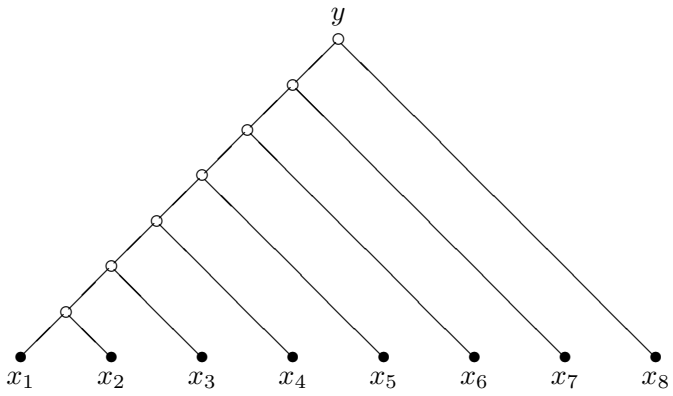


Figura 6.9: Albero di operazioni binarie che agli ingressi $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ associa l'uscita y, mediante una valutazione basata sull'associatività a sinistra.

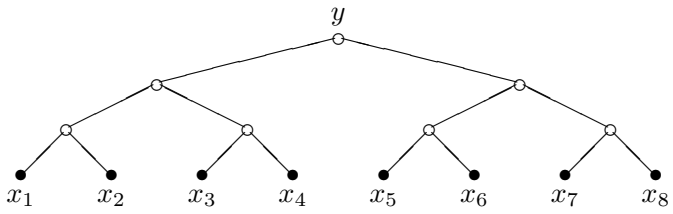


Figura 6.10: Albero di operazioni binarie che agli ingressi $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ associa l'uscita y, mediante una valutazione basata su una parentizzazione bilanciata.

ESERCIZI

6.1 Analizzare, sul piano dei *segni*, del *significato* e della *verità*, le seguenti affermazioni:

1. I 2 2 che compaiono in questa affermazione hanno significati diversi.
2. I due 2 che compaiono in questa affermazione hanno significati diversi.
3. I due due che compaiono in questa affermazione hanno significati diversi.
4. I 3 2 che compaiono in questa affermazione hanno significati diversi.

6.2 Completare e correggere la frase riportata nel seguente riquadro in modo che risulti sintatticamente e semanticamente corretta e che risulti vera.

Questa fra è composta da 6 parole.

6.3 Discutere la seguente scrittura:

$$1010_2 = 10$$

In particolare si analizzi il livello dei segni, dei concetti, dell'interpretazione, dei significati, dei valori di verità, delle ambiguità, delle convenzioni sulle parti sottintese.

6.4 Analizzare il significato del trattino orizzontale - che compare nelle seguenti scritture:

$$4 \qquad {}^{-}4 \qquad -4 \qquad -x \qquad 7-4 \qquad \frac{7}{4}$$

6.5 Individuare le operazioni coinvolte nella seguente espressione:

$$-\sqrt[3]{{}^{-}4 - 2^2}$$

6.6 Discutere la differenza fra le seguenti due espressioni RPN:

1. $a\ b\ c\ -\ -$
2. $a\ b\ -\ c\ -$

6.7 Descrivere in notazione infissa, RPN e funzionale mediante le operazioni *add*, *sub*, *mul*, *div* le seguenti espressioni:

1. $(a+b)(a-b)$
2. $a^2 - b^2$
3. $ax + by + c$
4. $ax^2 + bx + c$
5. $x^2 - 2xy - y^2$
6. $\frac{a-b}{ab}$

6.8 Usando le seguenti funzioni:

$$\begin{array}{lcl} add(x,y) & : & \text{addizione } x+y \\ mul(x,y) & : & \text{moltiplicazione } x*y \\ chs(x) & : & x \text{ cambiato di segno} \\ pow(x,y) & : & \text{potenza } x^y \end{array}$$

scrivere in notazione funzionale la seguente espressione:

$$\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$$

6.9 Le seguenti espressioni in notazione funzionale determinano il massimo fra tre dati elementi a, b, c. Analizzarle e discuterne le differenze.

$$max(a,b,c)$$

$$max(max(a,b),c)$$

$$max(a,max(b,c))$$

7

ESPRIMERE

[...] in LISP i programmi non si chiamano programmi ma espressioni. E non se le esegue, le si valuta. Il risultato della valutazione di un'espressione è semplicemente un valore; non vi sono effetti collaterali. Lo stato dell'universo rimane invariato.

Gregory Chaitin, *Alla ricerca di Omega*

Richiamando la citazione sopra riportata, esistono linguaggi di programmazione fortemente basati sulle espressioni; in quesi ambienti l'operazione fondamentale per il solutore consiste nello *scrivere* espressioni mentre il compito fondamentale per l'esecutore consiste nel *valutare* espressioni.

Di conseguenza, nei casi in cui il problema ammetta come soluzione un'espressione, il compito del solutore si riduce a scrivere espressioni che costituiscono il procedimento risolutivo del problema.

Nel contesto di un generico linguaggio, *esprimere* significa scrivere *espressioni*, in modo testuale o grafico od iconico. In un linguaggio naturale, quale ad esempio l'italiano o l'inglese, le espressioni sono costituite da componenti elementari quali nomi, articoli, aggettivi, predicati verbali; questi elementi vengono organizzati secondo specifiche regole grammaticali; similmente nel linguaggio matematico ed informatico le espressioni sono costituite da valori, da operatori e da altri elementi quali le parentesi.

Le espressioni costituiscono uno degli oggetti matematici più tipici ed importanti ed intervengono in molte questioni tradizionalmente di dominio della matematica.

7.1 Espressioni

Combinando, secondo delle specifiche regole [1], *operandi* (valori o oggetti denotati mediante un identificatore) ed *operatori* (in una qualche notazione (infissa, funzionale, ...)) si ottengono delle *espressioni*. Per talune di queste notazioni, la struttura delle espressioni può avvalersi dell'uso di parentesi che descrivono la priorità di esecuzione delle operazioni. Nel caso estremo più semplice un'espressione può essere costituita da un solo valore o da un identificatore che denota un valore.

Esempio 7.1.1 - A seguire sono riportate delle espressioni; gli identificatori che compaiono in queste espressioni denotano numeri decimali.

a) 327

b) $5 + 12$

c) $((45 - 13) * 3 - 8)/2 + 4 * (9 - 3)$

d) $x + y - 3$

e) $(a + b) < (a - b)$

Esempio 7.1.2 - Le espressioni, come visto nel capitolo 6, possono essere descritte in varie notazioni (algebrica, *RPN*, funzionale, ...) come riportato nelle seguenti scritture, tutte corrispondenti alla lunghezza della diagonale di un quadrato di lati di lunghezza a e b.

a) $\sqrt{x^2 + y^2}$ (notazione algebrica)

b) $x\ x\ *\ y\ y\ *\ +\ \sqrt{}$ (notazione RPN)

c) $sqrt(add(sqr(x), sqr(y)))$ (notazione funzionale)

7.2 Operazioni sulle espressioni

Un'espressione può essere elaborata con varie modalità: può essere *valutata* eseguendo un calcolo basato sulle regole interne dell'insieme dei valori componenti le espressioni, ottenendo un valore come risultato; si può eseguire un *controllo di equivalenza* fra due espressioni, ossia indagare se forniscono gli stessi valori; un'espressione può essere *ridotta* in modo simbolico, secondo delle regole algebriche, giungendo ad un'espressione equivalente ma di forma più semplice; un'espressione può essere *trasformata* sostituendo gli identificatori che vi compaiono con i valori o le espressioni ad essi associati. Tutte queste forme di elaborazione saranno trattate nei prossimi capitoli.

7.3 Espressioni e problemi

Le espressioni costituiscono un importante meccanismo per descrivere la soluzione di una ampia classe di problemi. Per questi problemi i dati costitui-

[1] Per mantenere il livello della presentazione molto elementare non vengono qui descritte in modo formale le regole sintattiche per la costruzione delle espressioni ma ci si limita a fornire degli esempi significativi.

scono gli operandi dell'espressione, le capacità dell'esecutore costituiscono gli operatori e l'espressione stessa costituisce la soluzione del problema.

I problemi risolti mediante un'espressione possono essere classificati in base alla forma ed al tipo dell'espressione che costituisce la soluzione.

Problemi aritmetici

Le espressioni che, valutate, forniscono dei numeri permettono di esprimere la soluzione di molti problemi di tipo computazionale ed algebrico, come descritto negli esempi che seguono.

Esempio 7.3.1 - La soluzione del problema:

> *Determinare l'n-esimo numero dispari.*

è costituita dall'espressione

$$2 * n + 1$$

Esempio 7.3.2 - La soluzione del problema:

> *Determinare l'area di un trapezio avente le basi di lunghezza B e b e l'altezza h.*

è data dalla nota espressione

$$\frac{(B + b) * h}{2}$$

Problemi decisionali

Le espressioni che, valutate, forniscono dei valori logici si prestano a descrivere la soluzione di molti problemi di tipo decisionale. Gli esempi che seguono illustrano alcuni casi interessanti.

Esempio 7.3.3 - Consideriamo il seguente problema:

> *Decidere se un dato un numero naturale di 3 cifre, considerato in base 10, è palindromo, ossia se si legge indifferentemente da sinistra a destra e da destra a sinistra. Ad esempio il numero 373 è palindromo.*

Un numero di tre cifre è palindromo se la cifra delle centinaia è uguale alla cifra delle unità. Pertanto, indicando con n il numero dato, esso risulta palindromo se e solo se

$$(n \,\mathsf{div}\, 100) = (n \,\mathsf{mod}\, 10)$$

Questa espressione costituisce la soluzione del problema.

Esempio 7.3.4 - La soluzione del seguente problema:

> *Stabilire se 3 dati numeri x, y, z sono in ordine crescente.*

è data dall'espressione

$$(x \leq y) \wedge (y \leq z)$$

Esempio 7.3.5 - Consideriamo il seguente problema:

> *Decidere se un dato anno è bisestile (un anno è bisestile se è divisibile per* 4 *ma non per* 100*, oppure è divisibile per* 400*; ad esempio sono bisestili gli anni* 1996*,* 2000*,* 2004*, mentre non lo sono gli anni* 1900*,* 2010*,* 2100*).*

Indicando con a l'anno che costituisce il dato del problema, l'espressione che esprime la condizione di essere bisestile si scrive:

$$(((a \bmod 4) = 0) \wedge ((a \bmod 100) \neq 0)) \vee ((a \bmod 400) = 0)$$

Tale espressione può essere denotata significativamente con $bisestile(a)$.

Problemi condizionali

L'*operatore condizionale if* è un'operazione ternaria che serve per selezionare una fra due alternative x ed y in base al valore di verità di una condizione c. Viene scritto con la notazione funzionale

$$if(c, x, y)$$

ed ha il seguente significato: se la condizione c è vera allora il risultato è x, altrimenti il risultato è y.

L'operatore condizionale *if* consente di costruire particolari espressioni, dette *espressioni condizionali*, che costituiscono la soluzione di molteplici problemi, come confermato dagli esempi che seguono.

Esempio 7.3.6 - Il risultato dell'operazione $if(1 = 2, 3, 4)$ è 4 in quanto la condizione $1 = 2$ è falsa.

Esempio 7.3.7 - Il valore assoluto di un numero reale x è esprimibile mediante l'espressione

$$if(x \geq 0, x, -x)$$

Esempio 7.3.8 - La distanza $|x-y|$ fra due numeri x ed y è esprimibile mediante l'espressione

$$if(x < y, y - x, x - y)$$

Esempio 7.3.9 - Il minimo fra due valori x ed y è esprimibile mediante l'espressione

$$if(x < y, x, y)$$

Esempio 7.3.10 - Il minimo fra tre valori x, y e z è esprimibile mediante l'espressione

$$if(x < y, if(x < z, x, z), if(y < z, y, z))$$

Esempio 7.3.11 - Consideriamo il problema:

> *Dati tre numeri reali, determinare la somma dei due numeri più grandi.*

Indicando con a, b, c i tre numeri dati, la somma dei due numeri più grandi è data da

$$(a+b+c) - min(a,b,c)$$

L'espressione funzionale $min(a,b,c)$ può essere esplicitata mediante l'operatore if come segue:

$$if(a<b, min(a,c), min(b,c))$$

Le qui riportate espressioni funzionali della forma $min(x,y)$ possono essere esplicitate come segue:

$$if(x<y, x, y)$$

In definitiva, un'espressione complessiva risolvente il problema proposto è:

$$(a+b+c) - if(a<b, if(a<c, a, c), if(b<c, b, c))$$

Esempio 7.3.12 - Il seguente problema costituisce un frequente test di controllo sui dati di problemi di tipo geometrico:

> *Decidere se tre assegnati valori numerici a, b, c possono costituire le misure dei lati di un triangolo.*

Dalla geometria la condizione di costruibilità di un triangolo si esprime:

> *Ogni lato deve essere minore della somma degli altri due.*

che è equivalente a

> *Il lato maggiore deve essere minore della somma degli altri due.*

ossia

$$max(a,b,c) < (a+b+c-max(a,b,c))$$

da cui

$$max(a,b,c) < (a+b+c)/2$$

e quest'ultima può essere espansa mediante l'operatore *if* come segue:

$$if(a>b, if(a>c, a, c), if(b>c, b, c)) < (a+b+c)/2$$

7.4 Espressioni come funzioni

Per indicare che nell'espressione E compaiono gli identificatori $x_1, x_2, \ldots, x_n$ useremo la notazione

$$E(x_1, x_2, \ldots, x_n)$$

Le espressioni assumono valori diversi a seconda dei valori che vengono associati agli identificatori coinvolti nelle espressioni stesse. In questo modo un'espressione può essere considerata come una funzione che dipende dagli identificatori coinvolti nell'espressione, facendo corrispondere loro il valore ottenuto nel processo di valutazione.

Esempio 7.4.1 - L'espressione algebrica $x^2 + 2xy + y^2$ identifica la funzione anonima

$$(x, y) \mapsto (x + y)^2$$

Per questioni di leggibilità e di usabilità un'espressione viene talvolta denotata mediante un identificatore.

Esempio 7.4.2 - La seguente espressione *divisibile* da come risultato un valore logico che esprime la condizione di divisibilità del numero m per il numero n:

$$divisibile(m, n) \stackrel{\text{def}}{=} (m \bmod n) = 0$$

7.5 Costruzione di espressioni

In molti casi la costruzione di un'espressione che costituisce la soluzione di un problema può essere sviluppata adottando le metodologie top-down e bottom-up. Gli esempi che seguono illustrano questa situazione.

Problema 7.5.1 Dati tre numeri naturali g, m, a, da interpretarsi come *giorno*, *mese*, *anno*, decidere se essi costituiscono una data corretta.

La soluzione è suggerita dalla celebre filastrocca si individua così facilmente l'espressione che caratterizza una data corretta (si osservi che qui le virgole che separano le condizioni fra parentesi tonde vanno intese come operatore logico *and*):

$$(m \in \{4, 6, 9, 11\}, g \leq 30)$$
$$\text{oppure}$$
$$(m = 2, (a \text{ è bisestile}, g \leq 29) \text{ oppure } (g \leq 28))$$
$$\text{oppure}$$
$$(m \in \{1, 3, 5, 7, 8, 10, 12\}, g \leq 31)$$

Ad un livello di raffinamento tipico dei linguaggi di programmazione, la condizione sopra può essere espressa come segue:

$$(((m = 4) \vee (m = 6) \vee (m = 9) \vee (m = 11)) \wedge (g \leq 30))$$
$$\vee$$
$$(m = 2) \wedge (bisestile(a) \wedge g \leq 29) \vee (g \leq 28))$$
$$\vee$$
$$((dispari(m) \wedge in(m, 1, 7)) \vee (pari(m) \wedge in(m, 8, 12))) \wedge (g \leq 31))$$

Le espressioni corrispondenti a *pari*, *dispari* e *in* possono essere scritte rispet-

tivamente come segue:

$$\begin{aligned} pari(m) &\stackrel{\text{def}}{=} (m \bmod 2) = 0 \\ dispari(m) &\stackrel{\text{def}}{=} (m \bmod 2) = 1 \\ in(m,a,b) &\stackrel{\text{def}}{=} (a \leq m) \wedge (m \leq b)) \end{aligned}$$

Si noti come nello sviluppo della soluzione del problema di questo esempio si sia usata in alcuni passaggi la metodologia top-down ed in altri la metodologia bottom-up. □

7.6 Espressioni ed insiemi di entità

Le espressioni costituiscono uno strumento per definire e descrivere sottoinsiemi delle entità di una classe C avente parametri caratteristici $(x_1, \ldots, x_n)$. Un'espressione logica E coinvolgente i parametri $x_1, \ldots, x_n$ individua il sottoinsieme delle entità di C soddisfacenti alla condizione $E(x_1, \ldots, x_n)$.

Esempio 7.6.1 - I punti del piano cartesiano sono individuati da una coppia (x, y) di valori numerici reali che denotano le coordinate nel riferimento cartesiano considerato. In questo contesto una equazione della forma $E(x, y) = 0$ rappresenta i punti (x, y) del piano che la soddisfano. Ad esempio l'equazione $3x - 2y + 4 = 0$ individua i punti di una *retta*; l'equazione $x^2 + y^2 - 4x - 6y + 5 = 0$ individua una *circonferenza*.

Esempio 7.6.2 - Nel contesto del piano cartesiano descritto nel precedente esempio 7.6.1, l'insieme dei punti appartenenti alle bisettrici dei quadranti e diversi dall'origine risulta individuato dall'espressione

$$((x = y) \vee (x = -y)) \wedge (x \neq 0)$$

7.7 Sistemi di condizioni

In molte situazioni vengono considerate più condizioni connesse fra loro mediante degli operatori logici *and* ($\vee$) e *or* ($\wedge$). In questi casi si parla di *sistemi di condizioni*. A seconda del particolare operatore logico utilizzato si hanno due tipologie di sistemi.

Un *sistema di congiunzione* (o *sistema di intersezione*) di k condizioni

$$\begin{cases} E_1(x_1, \ldots, x_n) \\ \vdots \\ E_k(x_1, \ldots, x_n) \end{cases}$$

individua le entità soddisfacenti a tutte le condizioni indicate. Un tale sistema risulta equivalente alla singola condizione

$$E_1(x_1, \ldots, x_n) \wedge \cdots \wedge E_k(x_1, \ldots, x_n)$$

Analogamente, un *sistema di disgiunzione* (o *sistema di unione*) di k condizioni

$$\left[\begin{array}{l} E_1(x_1,\ldots,x_n) \\ \vdots \\ E_k(x_1,\ldots,x_n) \end{array}\right.$$

individua le entità soddisfacenti ad almeno una delle condizioni indicate. Un tale sistema risulta equivalente alla singola condizione

$$E_1(x_1,\ldots,x_n) \vee \cdots \vee E_k(x_1,\ldots,x_n)$$

Esempio 7.7.1 - I numeri reali compresi nell'intervallo $[0,1]$ possono essere espressi mediante il sistema di congiunzione

$$\left\{\begin{array}{l} x \geq 0 \\ x \leq 1 \end{array}\right.$$

mentre i numeri esterni allo stesso intervallo possono essere descritti mediante il sistema di unione

$$\left[\begin{array}{l} x < 0 \\ x > 1 \end{array}\right.$$

Esempio 7.7.2 - I sistemi di congiunzione e disgiunzione risultano utili per descrivere delle figure nel piano cartesiano, dove un punto risulta individuato mediante le sue coordinate ed una figura (lineare o bidimensionale) può essere individuata mediante un'espressione che coinvolge le coordinate (x,y) di un generico punto della figura. La circonferenza di raggio 1 e centro l'origine (o, equivalentemente, l'insiemi dei punti (x,y) che hanno distanza 1 dall'origine) è individuata dall'espressione

$$\sqrt{x^2+y^2} = 1$$

Il semicerchio delimitato dalla precedente circonferenza ed appartenente al semipiano delle ordinate positive è descritto dal sistema di congiunzione

$$\left\{\begin{array}{l} \sqrt{x^2+y^2} \leq 1 \\ y > 0 \end{array}\right.$$

Questo sistema puè essere espresso in modo equivalente mediante la condizione

$$(x^2+y^2 \leq 1) \wedge (y > 0)$$

La porzione di piano delimitata dal quadrato di vertici $(1,1)$, $(^-1,1)$, $(^-1,^-1)$, $(1,^-1)$ è descritto dal sistema di congiunzioni

$$\left\{\begin{array}{l} |x| \leq 1 \\ |y| \leq 1 \end{array}\right.$$

Questo sistema è equivalente alla condizione

$$(|x| \leq 1) \wedge (|y| \leq 1)$$

che può essere esplicitata nella forma

$$(^{-}1 \leq x) \wedge (x \leq 1) \wedge (^{-}1 \leq y) \wedge (y \leq 1)$$

Esempio 7.7.3 - La croce posta sugli assi cartesiani, avente centro nell'origine ed avente i 4 lati di lunghezza 1 può essere descritta come unione di due segmenti, mediante il seguente sistema misto:

$$\left[\begin{array}{l} \left\{\begin{array}{l} ^{-}1 \leq x \\ x \leq 1 \\ y = 0 \end{array}\right. \\ \left\{\begin{array}{l} x = 0 \\ ^{-}1 \leq y \\ y \leq 1 \end{array}\right. \end{array}\right.$$

oppure, in modo equivalente, come intersezione della circonferenza unitaria di centro l'origine e gli assi coordinati, mediante il seguente sistema misto:

$$\left\{\begin{array}{l} x^2 + y^2 \leq 1 \\ \left[\begin{array}{l} x = 0 \\ y = 0 \end{array}\right. \end{array}\right.$$

Questo sistema è equivalente alla condizione

$$(x^2 + y^2 \leq 1) \wedge ((x = 0) \vee (y = 0))$$

7.8 Valutazione delle espressioni

Una delle più frequenti forme di elaborazione di un'espressione composta solo da valori consiste nell'eseguire i calcoli basandosi sulle regole interne all'insieme dei valori cha la compongono, ottendo come risultato un valore. Tale forma di elaborazione viene detta *valutazione* e produce come risultato un valore, secondo il seguente schema di trasformazione:

$$Espressione \rightarrow Valore$$

Negli usuali linguaggi di programmazione la valutazione di un'espressione viene considerata come una capacità di base dell'esecutore. Nonostante ciò, risulta interessante esaminare i meccanismi sottostanti i processi di valutazione di un'espressione. Questi meccanismi dipendono dalla notazione usata per descrivere l'espressione.

Disponendo di un esecutore in grado di eseguire le sole operazioni elementari fra numeri e valori logici, il processo di valutazione di un'espressione viene svolto attraverso una sequenza di trasformazioni (dell'espressione) che dipendono dalla struttura sintattica (forma) dell'espressione, ossia dalla notazione mediante la quale è descritta, eseguendo ad ogni passo il calcolo di un'operazione.

Il processo di valutazione di un'espressione si fonda sulle capacità dell'esecutore di eseguire una singola operazione; il caso più frequente è costituito dalle operazioni elementari fra due numeri. Questo processo di valutazione può essere studiato e descritto a partire dall'analisi del procedimento che una persona applica nella valutazione di un'espressione articolata; è il procedimento che abbiamo applicato nei primi anni della scuola media inferiore per valutare un'espressione aritmetica articolata. In sintesi, il procedimento si fondava sulle seguenti indicazioni:

- svolgere dapprima le operazioni elementari all'interno delle parentesi più interne
- eseguire le operazioni a più alta priorità (prima le addizioni e sottrazioni e poi moltiplicazioni e divisioni)

Questo processo di valutazione si esplica eseguendo ad ogni passo una o più operazioni e trascrivendo l'espressione che man mano si riduce fino a ridursi ad un singolo valore che costituisce il risultato della valutazione.

Esempio 7.8.1 - A seguire sono riportati i passi per la valutazione di un'espressione. Il simbolo $\rightarrow$ denota un passo del processo di valutazione dell'espressione.

$$(7-5)*4-6/2 \rightarrow 2*4-6/2 \rightarrow 8-3 \rightarrow 5$$

Esempio 7.8.2 - Oltre a numeri ed operatori aritmetici, in un'espressione possono interverire operatori relazionali, logici e valori booleani, come descritto nei due casi che seguono:

a) $(7-2) \leq 5 \rightarrow 5 \leq 5 \rightarrow$ `TRUE`

b) $(1<2)<(3<4) \rightarrow$ `TRUE` $<(3<4) \rightarrow$ `TRUE` $<$ `TRUE` $\rightarrow$ `FALSE`

Il procedimento di valutazione di un'espressione può essere descritto mediante un ***albero di valutazione*** nel quale vengono riportati i risultati intermedi ottenuti dalla valutazione delle sottoespressioni.

Esempio 7.8.3 - Il procedimento di valutazione dell'espressione $(7-5)*4-6/2$ può essere descritto mediante il seguente albero di valutazione:

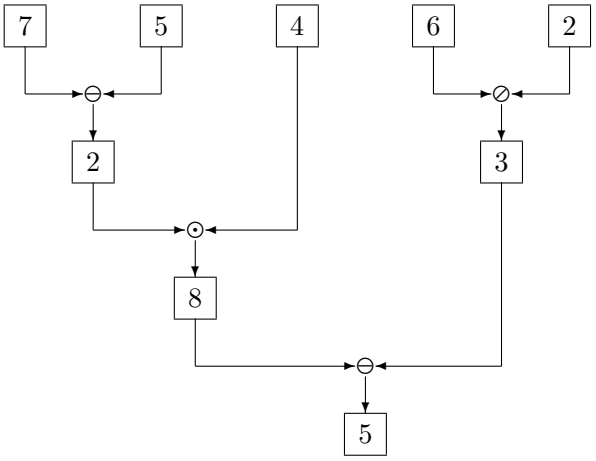


Figura 7.1: *Albero di valutazione* dell'espressione $(7-5)*4-6/2$.

7.9 Tipo delle espressioni

Un'espressione può essere classificata in base al tipo del valore che si ottiene dalla sua valutazione. Se il valore che si ottiene dalla valutazione di un'espressione appartiene ad un insieme T, si dice che l'espressione è *di tipo* T; in particolare, se T è un insieme numerico, l'espressione viene detta *espressione aritmetica*, se T è costituito dall'insieme dei valori logici $\{\texttt{FALSE}, \texttt{TRUE}\}$, l'espressione viene detta *espressione logica* o *espressione booleana* o *condizione*.

Esempio 7.9.1 - In alcune situazioni il tipo di un'espressione può essere dedotto dalla forma dell'espressione. Osservando che $<$ è l'operatore principale che compare nella seguente espressione, si deduce immediatamente che la seguente è un'espressione logica:

$$(4+3) < (2*(7-5))$$

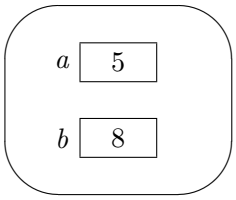
7.10 Valutazione in un ambiente

Un'espressione risulta valutabile direttamente se non contiene identificatori. Nel caso contenga identificatori, un'espressione risulta valutabile se si esegue una fase di elaborazione preventiva in cui ogni identificatore venga sostituito dal valore ad esso associato. A questo scopo risulta utile la seguente definizione.

DEFINIZIONE 9. Dicesi *ambiente di valutazione* un insieme di identificatori con il loro valore associato, ossia in insieme di coppie della forma (id, val) dove id è un generico identificatore e val è il valore ad esso associato. Useremo la notazione $(id, \perp)$ per indicare che l'identificatore id non è associato ad alcun

valore o, equivalentemente, che è associato al *valore nullo*. Un ambiente di valutazione dicesi *completo* rispetto ad un'espressione E se contiene la definizione dei valori associati a ciascun identificatore che compare nell'espressione E.

Esempio 7.10.1 - Un ambiente A di valutazione può essere descritto mediante uno schema grafico come segue:



oppure mediante una notazione testuale della forma

$$A = \{(a, 5), (b, 8)\}$$

Tale ambiente risulta completo rispetto all'espressione $a * a + 2 * b$ mentre non lo è rispetto all'espressione $a * x + b$. In questo ambiente l'espressione $a * a + 2 * b$ può essere valutata mediante i seguenti passi:

$$a * a + 2 * b \rightarrow 5 * 5 + 2 * 8 \rightarrow 25 + 16 \rightarrow 41$$

Osservazione. Nel caso in cui la soluzione di una classe di problemi sia costituita da un'espressione, per risolvere una specifica istanza è sufficiente sostituire le variabili che compaiono nell'espressione con i valori della specifica istanza, ottenendo un'espressione composta solo da valori. L'ultimo passo consiste nel valutare l'espressione così generata ottenendo, alla fine, il valore che costituisce il risultato dell'istanza del problema.

7.11 Espressioni equivalenti

Si possono avere espressioni *formalmente* diverse ma *equivalenti*, come specificato dalla seguente definizione.

DEFINIZIONE 10. Due espressioni si dicono *equivalenti* se, per ogni possibile ambiente di valutazione completo rispetto ad entrambe, le espressioni forniscono lo stesso valore.

Esempio 7.11.1 - A seguire sono riportate delle coppie di espressioni algebriche diverse ma equivalenti.

a) $x + x$ è equivalente a $2 * x$

b) $x * x$ è equivalente a x^2

c) $a + b + a$ è equivalente a $2 * a + b$

d) $a * x * x + b * x + c$ è equivalente a $(a * x + b) * x + c$

In certe situazioni l'equivalenza fra due espressioni è meno evidente e richiede una spiegazione più profonda, come attesta il seguente esempio.

Esempio 7.11.2 - L'espressione $1 + 2 + 3 + \cdots + n$ è equivalente all'espressione $n * (n + 1)/2$.

Esempio 7.11.3 - L'operatore ternario if permettere di esprimere in modo equivalente gli operatori logici, come descritto dalla seguente lista di equivalenze

a) $x \wedge y$ è equivalente a $if(x, y, \texttt{FALSE})$
b) $x \vee y$ è equivalente a $if(x, \texttt{TRUE}, y)$
c) $\neg x$ è equivalente a $if(x, \texttt{FALSE}, \texttt{TRUE})$
d) $x\ xor\ y$ è equivalente a $if(x, y = \texttt{FALSE}, y)$
e) $x \Rightarrow y$ è equivalente a $if(x, y, \texttt{TRUE})$
f) $x \equiv y$ è equivalente a $if(x, y, \neg y)$

L'ultima scrittura può essere espansa nella seguente:

$$if(x, y, if(y, \texttt{FALSE}, \texttt{TRUE}))$$

7.12 Analisi di un'espressione

Adesso che è stato definito il processo di valutazione lo studio di un'espressione si arricchisce, oltre che per l'aspetto sintattico concernente la forma, di un aspetto semantico rappresentato dal valore.

Analizzare un'espressione significa svolgere un processo preventivo di indagine, prima di procedere alla sua valutazione, come descritto nell'algoritmo 1. Durante il processo di valutazione di un'espressione possono insorgere degli errori. I casi più frequenti ricadono nelle seguenti categorie:

- errori di sintassi dovuti all'errata forma dell'espressione; ad esempio la mancanza, l'esubero o l'incoerente disposizione delle parentesi
- errori dovuti ai particolari valori che non rendono possibile l'esecuzione di un'operazione; ad esempio una *divisione per zero*.

In tutti questi casi si procede dando una segnalazione di errore oppure ritornando come risultato il *valore nullo* oppure, nei linguaggi che lo prevedono, ricorrendo a dei meccanismi di controllo del flusso detti *gestione delle eccezioni*.

Algoritmo 1 - Analisi di un'espressione

Input: espressione aritmetica E da analizzare
Output: valore dell'espressione o segnalazione di errore

```
1: if E è sintatticamente corretta then
2:    if E non contiene variabili then
3:       valutare E
4:    else
5:       stabilire per quali tipi delle variabili è corretta
6:       if E è costante then
7:          determinarne il valore
8:       end if
9:    end if
10: else
11:    individuare il (primo) punto di errore
12:    qualificare con un messaggio l'errore
13: end if
```

7.13 Limiti delle espressioni

Il fatto che fra *dati* e *risultati* di un problema esista una relazione funzionale della forma $risultati = algoritmo(dati)$, può lasciar supporre che un problema possa essere risolto mediante un'espressione che assolve al compito della trasformazione dell'algoritmo. Ma non tutti i problemi ammettono una soluzione esprimibile mediante un'espressione in quanto non tutte le funzioni sono esprimibili mediante un'espressione. Ad esempio la soluzione del problema: *Determinare l'n-esimo numero primo.* oppure: *Decidere se un dato numero naturale è primo.* richiede un algoritmo più articolato.

Per risolvere la totalità dei problemi risolvibili servono dei meccanismi più potenti. Adottando il paradigma imperativo si fa uso di variabili che memorizzano lo stato della computazione; tali variabili vengono modificate mediante delle assegnazioni, come si vedrà nel prossimo capitolo; il tutto viene poi corredato dalla possibilità di fare ricorso a dei controlli (in particolare dei cicli che permettono la ripetizione delle istruzioni). Alternativamente, adottando il paradigma funzionale si fa affidamento sul meccanismo delle funzioni ricorsive.

ESERCIZI

7.1 Risolvere l'equivoco del problema di Peano riportato nella citazione all'inizio di questo capitolo.

7.2 Gianni va dal cartolaio con 4 monete da 2 euro, 2 monete da 50 centesimi per prendere 3 quaderni da 1.4 euro l'uno, 4 penne da 45 centesimi di euro l'una, 4 buste del costo di 10 centesimi di euro l'una, con corrispondenti francobolli da 81 centesimi di euro l'uno. Con i soldi che rimangono, Gianni acquista delle bustine di figurine; ciascuna bustina costa 16 centesimi di euro e contiene 5 figurine. Scrivere un'espressione aritmetica che, valutata, fornisca il numero di figurine che Gianni riesce a comperare.

7.3 Si possiede una somma di s euro con la quale acquistare delle pentole con corrispondenti coperchi. Una pentola costa p euro mentre un coperchio ne costa c. Quante pentole complete di coperchio si possono acquistare? Quanti euro rimangono?

7.4 Discutere la differenza di forma dei problemi proposti nei due precedenti esercizi.

7.5 Determinare il numero di sbarrette (intere) di ferro di lunghezza d che si possono ritagliare da n sbarre di lunghezza l. Determinare la lunghezza complessiva del ferro sprecato.

7.6 Determinare il (massimo) numero di quadrati di lato fissato che si possono ritagliare da una forma rettangolare di date dimensioni.

7.7 Determinare il (massimo) numero di rettangoli di date dimensioni che si possono ritagliare da una forma rettangolare di date dimensioni. Si noti che questo problema, pur sembrando una banale generalizzazione del problema precedente, risulta di difficile soluzione. In questo problema risulta operativamente interessante anche determinare la disposizione dei rettangoli da ritagliare. Problemi di questo genere rientrano nella vasta classe dei *problemi di allocazione delle risorse* trattati in un ramo della Matematica noto come Ricerca Operativa e Programmazione Lineare.

7.8 Fissato un numero naturale k, determinare il successivo di un numero naturale n, $n \leq k$, in sequenza circolare secondo il seguente schema:

$$0 \to 1,\ \ 1 \to 2,\ \ 2 \to 3,\ \ \ldots,\ \ k-1 \to k,\ \ k \to 0$$

7.9 Dato il numero naturale n, considerato in base dieci, determinare

1. la cifra delle unità di n
2. la cifra delle decine di n
3. la cifra delle centinaia di n

7.10 Dato un numero naturale n di due cifre decimali e non multiplo di 10, determinare il numero ottenuto da n invertendo le sue cifre. Analogamente per il caso in cui n sia costituito da tre cifre decimali.

7.11 Dato il numero naturale n, determinare

1. il più piccolo numero pari maggiore o uguale a n

2. il numero di decine di n
3. il valore di n troncato alle centinaia
4. il valore di n arrotondato alle centinaia
5. il più piccolo numero divisibile per 10 e per n

7.12 Dato il numero naturale n, scrivere delle espressioni corrispondenti alle seguenti condizioni:

1. n è pari
2. n è dispari
3. n è divisibile per 10
4. n è composto da 2 cifre decimali

7.13 Siano m, n due numeri naturali. Utilizzando gli operatori aritmetici $+, -, *, \mathsf{div}$, scrivere un'espressione che corrisponda alla funzione δ di Kronecker definita come segue:

$$\delta(m,n) = \begin{cases} 0 & \text{se } m \neq n \\ 1 & \text{se } m = n \end{cases}$$

7.14 Dati due numeri naturali m e n, Usando solo le quattro operazioni aritmetiche $+, -, *, \mathsf{div}$, determinare

1. il resto della divisione intera fra m ed n
2. il valore naturale arrotondato della divisione fra m ed n
3. il più grande multiplo di m minore o uguale ad n
4. il valore di n arrotondato al più vicino multiplo di k

7.15 Siano m, n due generici numeri naturali. Utilizzando gli operatori aritmetici $+, -, *, \mathsf{div}$ e sfruttando la definizione che $m - n = 0$ se $m \leq n$ (*sottrazione parziale*), determinare

1. il minimo fra m ed n
2. il massimo fra m ed n
3. la distanza $|m - n|$ fra m ed n
4. il confronto di ordine fra m ed n (1 se $m \leq n$, 0 altrimenti)
5. il segno di n (0 se $n = 0$; 1 altrimenti)
6. la parità di n (0 se n è pari, 1 altrimenti).

7.16 Sia $[a, b]$ l'intervallo chiuso dei numeri reali compresi fra a e b. Usando gli operatori di confronto e gli operatori logici, scrivere delle espressioni equivalenti ai seguenti due predicati:

1. $x \in [a, b]$
2. $x \notin [a, b]$

7.17 Sia $\{a, b\}$ un insieme di due elementi. Usando gli operatori di confronto e gli operatori logici, scrivere delle espressioni equivalenti ai seguenti due predicati:

1. $x \in \{a, b\}$
2. $x \notin \{a, b\}$

7.18 Dati tre numeri reali, determinare

1. il minimo dei tre numeri
2. il massimo dei tre numeri
3. il numero di mezzo dei tre numeri
4. l'ampiezza del più piccolo intervallo contenente i tre numeri
5. la somma dei due numeri più grandi fra i tre numeri
6. il numero dei numeri distinti fra i tre numeri
7. la somma dei due numeri più piccoli
8. la somma dei due numeri più grandi

7.19 Dati quattro numeri reali, determinare:

1. il minimo ed il massimo fra quattro numeri
2. la somma dei due numeri più grandi

7.20 Date le lunghezze a, b, c dei lati di un triangolo, scrivere delle espressioni logiche corrispondenti alle seguenti condizioni:

1. il triangolo è scaleno
2. il triangolo è isoscele
3. il triangolo è equilatero
4. il triangolo è acutangolo
5. il triangolo è rettangolo
6. il triangolo è ottusangolo

7.21 Decidere se tre assegnati numeri naturali costituiscono una terna pitagorica.

7.22 Decidere se un triangolo di lati di lunghezze a, b, c è copribile mediante un altro triangolo di lati di lunghezze x, y, z.

7.23 Decidere se tre dati numeri reali possono costituire le misure dei lati di un triangolo.

7.24 Decidere se quattro numeri reali possono costituire le misure dei lati di un quadrilatero.

7.25 Discutere se l'espressione

$$(x = y) = (y = z)$$

esprime il fatto che i tre elementi x, y, z sono tutti uguali fra loro. Spiegare perché, nella formulazione di questo quesito, x, y, e z possono essere elementi qualsiasi (e non necessariamente numeri).

7.26 Siano a, b, c le lunghezze dei lati di un triangolo. Stabilire il tipo di triangolo (scaleno, isoscele, equilatero), sapendo che soddisfano alla condizione

$$(a = b) = (b = c)$$

7.27 Siano x, y, z tre numeri reali. Stabilire quali delle seguenti condizioni esprimono il fatto che i tre numeri x, y, z siano tutti distinti fra loro.

1. $(x \neq y) \wedge (y \neq z)$
2. $(x - y) * (x - z) * (y - z) \neq 0$
3. $(x = y) < ((x \neq z) \wedge (y \neq z))$
4. $\neg((x = y) \vee (x = z) \vee (y = z))$
5. $(x < y) \wedge (y > z)$

7.28 Siano x, y, z tre numeri reali. Scrivere delle espressioni logiche corrispondenti alle seguenti condizioni:

1. I tre numeri sono in ordine crescente.
2. I tre numeri sono in ordine decrescente.
3. I tre numeri sono in ordine (crescente o decrescente).
4. I tre numeri sono non sono in alcun ordine.

7.29 Usando in notazione funzionale gli operatori aritmetici, gli operatori di confronto (eq, ne, lt, gt, le, ge) e l'operatore condizionale ternario if, scrivere un'espressione che rappresenti la distanza fra due dati numeri reali a e b.

7.30 Usando l'operatore condizionale *if*, esprimere gli operatori logici *and*, *or*, *not* e *xor* (*or* esclusivo).

7.31 Usando solamente gli operatori di confronto, esprimere l'operatore logico *xor* (*or* esclusivo).

7.32 Determinare la lunghezza del più piccolo intervallo contenente tre dati numeri reali.

7.33 Determinare la massima superficie d'appoggio di un parallelepipedo.

7.34 Determinare la diagonale di un parallelepipedo.

7.35 Siano a, b, x, y quattro numeri distinti, con $a < b$. Scrivere delle espressioni che esprimano i seguenti fatti:

1. il numero x è interno all'intervallo $[a, b]$
2. i due numeri x ed y sono entrambi interni all'intervallo $[a, b]$
3. i due numeri x ed y sono entrambi esterni all'intervallo $[a, b]$

4. i due numeri x ed y sono uno interno e l'altro esterno all'intervallo $[a, b]$
5. i due numeri x ed y sono entrambi esterni all'intervallo $[a, b]$ e da parti opposte

7.36 Siano a, b due numeri reali che individuano un punto sull'asse reale. Stabilire se i due punti stanno da bande opposte rispetto all'origine dell'asse.

7.37 Dati quattro numeri x_1, x_2, x_3, x_4 da interpretarsi come coordinate di due intervalli $[x_1, x_2]$, $[x_3, x_4]$, $x_1 \leq x_2$, $x_3 \leq x_4$, in un riferimento cartesiano su una retta, scrivere delle espressioni che rappresentino:

1. la condizione di intersezione dei due intervalli
2. la lunghezza dell'intervallo intersezione
3. la lunghezza del più piccolo intervallo contenente entrambi gli intervalli.

7.38 Fissato un sistema di riferimento cartesiano, decidere se un dato segmento di estremi (x_1, y_1), (x_2, y_2) interseca l'asse delle ascisse.

7.39 Siano (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) le coordinate di quattro punti in un sistema di riferimento cartesiano. Decidere se i quattro punti sono

1. vertici di un quadrato
2. vertici di un rettangolo
3. vertici di un trapezio
4. allineati

7.40 Dato nel piano cartesiano un punto di coordinate (x, y), scrivere un'espressione equivalente alla seguente condizione: *Il punto sta sulla croce posta sugli assi cartesiani, avente centro nell'origine ed avente i* 4 *lati di lunghezza* 1.

7.41 Scrivere delle espressioni che risolvano i seguenti problemi nel contesto di un sistema di riferimento cartesiano:

1. Decidere se un punto è interno ad una circonferenza.
2. Decidere se un segmento è interno ad una circonferenza.
3. Stabilire se un segmento interseca una circonferenza.
4. Stabilire se una circonferenza interseca gli assi coordinati.
5. Stabilire se una data circonferenza è completamente contenuta nel primo quadrante.
6. Stabilire se un segmento è completamente contenuto all'interno di una circonferenza.
7. Determinare la distanza di un punto da una circonferenza.

7.42 Siano (x, y) le coordinate di un punto P del piano cartesiano. Scrivere delle espressioni che esprimano le seguenti condizioni:

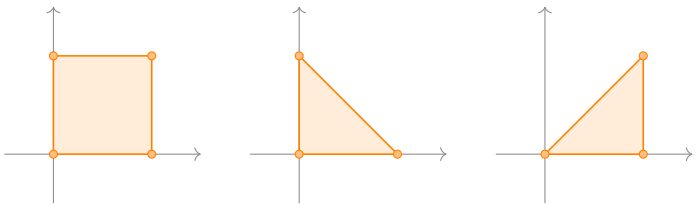
1. il punto P si trova nel primo quadrante
2. il punto P sta sul primo o terzo quadrante

3. il punto P sta su uno degli assi
4. il punto P non sta su un asse
5. il punto P non sta sull'origine
6. il punto P sta sugli assi ma è diverso dall'origine
7. il punto P sta su una delle bisettrici dei quadranti
8. il punto P sta sull'asse x ma è diverso dall'origine
9. il punto P dista dall'origine meno di 1 unità
10. il punto P sta su una delle semirette positive degli assi
11. il punto P sta su un asse o su una delle due bisettrici dei quadranti
12. il punto P sta sulla croce posta sugli assi cartesiani, avente centro nell'origine ed avente i 4 lati di lunghezza 1 ed è diverso dall'origine
13. il punto P sta all'interno della circonferenza unitaria incentrata nell'origine.
14. il punto P dista dagli assi meno di un'unità

7.43 Dato nel piano cartesiano un punto di coordinate (x, y), scrivere dei sistemi di equazioni corrispondenti alle seguenti condizioni (evidenziate nella figura che segue):

1. *Il punto si trova all'interno del quadrato* $[0,1] \times [0,1]$.
2. *Il punto sta all'interno del triangolo avente per vertici i punti di coordinate* $(0,0)$, $(1,0)$, $(0,1)$.
3. *Il punto sta all'interno del triangolo avente per vertici i punti di coordinate* $(0,0)$, $(1,0)$, $(1,1)$.

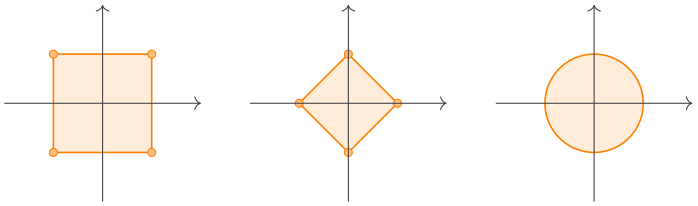
Tradurre ciascun sistema di condizioni in un'unica espressione.



7.44 Dato nel piano cartesiano un punto di coordinate (x, y), scrivere dei sistemi di equazioni corrispondenti alle seguenti condizioni (evidenziate nella figura che segue):

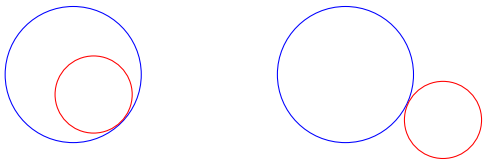
1. *Il punto si trova all'interno del quadrato* $[^-1,1] \times [^-1,1]$.
2. *Il punto sta all'interno del quadrato avente per vertici i punti di coordinate* $(1,0)$, $(0,1)$, $(^-1,0)$, $(0,^-1)$.
3. *Il punto sta all'interno della circoferenza unitaria incentrata nell'origine.*

Tradurre ciascun sistema di condizioni in un'unica espressione.

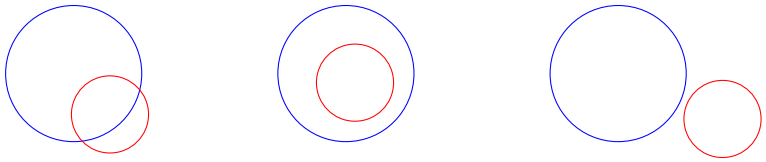


7.45 Dati i raggi r_1 ed r_2 e la distanza d dei centri di due *circonferenze*, scrivere delle condizioni che esprimano le seguenti situazioni:

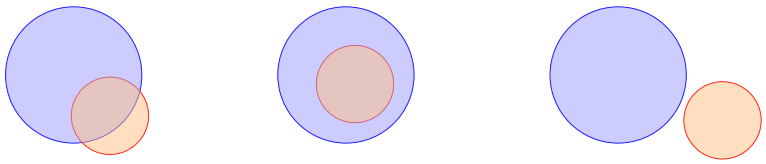
1. le due circonferenze sono fra loro tangenti (internamente o esternamente).



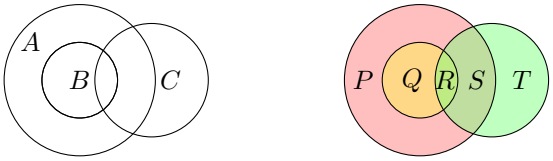
2. le due circonferenze si intersecano. Con riferimento alla figura che segue: *vero* nel primo caso, *falso* negli altri due.



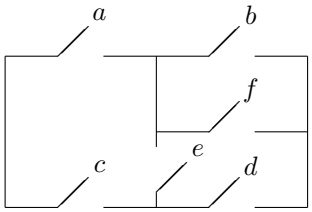
7.46 Dati i raggi r_1 ed r_2 e la distanza d dei centri di due *cerchi*, scrivere un'espressione logica che esprima il fatto che le due circonferenze si intersecano. Con riferimento alla figura che segue: *vero* nei primi due casi, *falso* nel terzo.



7.47 Si considerino 3 insiemi A, B, C descritti mediante il seguente diagramma di Venn (parte sinistra della figura). Usando gli operatori insiemistici $\cap$, $\cup$ e $\setminus$ scrivere delle espressioni corrispondenti ai 5 diversi sottoinsiemi P, Q, R, S, T descritti dalle zone di diverso colore riportate nella parte destra della figura che segue.



7.48 Con riferimento alla seguente piantina di una casa, indicando con x il fatto che *la porta x è aperta*, scrivere una condizione che esprima il fatto che ci sia corrente d'aria.



Stabilire se dall'espressione è possibile risalire alla topologia della piantina.

7.49 Valutare le seguenti espressioni:

1. $(1 < 2) = (2 = 3)$
2. $((1 < 2) < (3 < 4)) < (5 < 6)$

7.50 Valutare nell'ambiente $\{(a, 1), (b, 2), (c, 3)\}$ le seguenti espressioni:

1. $(a + 2) < b$
2. $\neg((a + b) > c)$
3. $((a + b) = c) \wedge ((b + c) < a)$

7.51 Supponendo che gli operandi che compaiono nella seguente espressione siano di tipo numerico, dedurre qual è il tipo della seguente espressione:

$$(a < (b - 1)) \ \wedge \ ((2 > b) \ \vee \ (\neg(a < 10)))$$

Valutare l'espressione nell'ambiente $\{(a, 2), (b, 1)\}$.

7.52 Se a, b, x, y sono delle variabili di un tipo ordinato, dimostrare che la seguente espressione rappresenta un'espressione di tipo booleano:

$$\neg((a \leq b) \wedge ((x = 1) \vee (y = 1)))$$

Valutare l'espressione nell'ambiente $\{(a, 2), (b, 7), (x, 3), (y, 1)\}$.

7.53 Dire quali fra le seguenti espressioni sono semanticamente corrette, assumendo che i valori che intervengono siano tutti numeri. Nel caso di espressioni semanticamente corrette, valutarne il tipo (numerico o logico). Nei casi in cui sia possibile, determinarne il valore.

1. $a = a + 1$
2. $\neg(a - b)$
3. $(a < b) \ \vee \ ((a + 1) > b)$

7.54 Dire quali fra le seguenti espressioni sono semanticamente corrette, assumendo che le variabili m ed n siano di tipo numerico e p e q di tipo logico. Nel

caso di espressioni sintatticamente corrette, valutarne il tipo. Nei casi in cui sia possibile, determinarne il valore.

1. $(m = n) = (p = q)$
2. $m < (p \wedge q)$
3. $m = (m = n)$
4. $p = (p = q)$

7.55 Stabilire il tipo delle variabili coinvolte nella seguente espressione affinché risulti corretta.

$$(x < y) < z$$

Discutere se e quando l'operatore di confronto $<$ è associativo.

7.56 Stabilire il tipo delle seguenti espressioni, assumendo che le variabili p, q, r siano di tipo logico e le variabili x, y, z siano di tipo numerico:

1. $p = (x = y)$
2. $(p = q) = r$
3. $(x = y) = (y = z)$
4. $(x < y) < (p < q)$
5. $(p < (x < y)) < q$

7.57 Discutere l'equivalenza delle seguenti espressioni, dove x è una variabile logica:

1. $\neg x$
2. $x = \texttt{FALSE}$
3. $x \neq x$
4. $x < \texttt{TRUE}$
5. $if(x, \texttt{FALSE}, \texttt{TRUE})$

7.58 Stabilire quali fra le seguenti espressioni sono equivalenti fra loro:

1. $(x = y) < x$
2. $(x < y) = (x = y)$
3. $x > y$
4. $x \wedge \neg y$

7.59 Siano a, b, x, y variabili numeriche e p, q variabili logiche. Stabilire se le seguenti coppie di espressioni sono equivalenti:

1. $(x * y) = 0 \quad \equiv \quad (x = y) \wedge (x = 0)$
2. $(x = a) \vee (y = b) \quad \equiv \quad (x - a) * (y - b) = 0$
3. $p \vee q \quad \equiv \quad p \neq q$

7.60 Sia x una variabile logica. Dimostrare l'equivalenza

$$\neg x \quad \equiv \quad x = (x \neq x)$$

7.61 Esprimere mediante l'operatore condizionale *if*, senza fare uso degli operatori logici, un'espressione equivalente alla seguente:

$$(p \wedge q) \vee r$$

7.62 Stabilire il tipo delle seguenti espressioni, assumendo che le variabili che compaiono siano tutte di tipo numerico.

1. $if(a < b, a, b)$
2. $if(a > b, a < b, a = b)$
3. $if(a = b, a, if(a > b, a - b, b - a))$

7.63 Esprimere mediante gli operatori logici (in modo cortocircuitato) delle espressioni equivalenti alle seguenti:

1. $if(x = y, y = z, x = z)$
2. $if(c_1, \texttt{FALSE}, if(c_2, \texttt{TRUE}, \alpha))$

7.64 Dimostrare che la seguente espressione è vera se e solo se i tre oggetti x, y, z sono uguali fra loro:

$$if(x = y, x = z, x = y)$$

7.65 Siano c_1 e c_2 due generiche condizioni. Dimostrare che, se c_2 è valutabile, le seguenti due espressioni sono equivalenti:

1. $if(c_1, if(c_2, x, y), y)$
2. $if(c_1 \wedge c_2, x, y)$

7.66 Discutere se ed in quali casi le seguenti coppie di espressioni sono fra loro equivalenti.

1. $if(c_1 \wedge c_2, x, y) \quad \equiv \quad if(c_1, if(c_2, x, y), y)$
2. $if(c_1 \vee c_2, x, y) \quad \equiv \quad if(c_1, x, if(c_2, x, y))$

7.67 Usando l'operazione condizionale if, determinare

1. massimo fra due numeri x e y
2. minimo fra tre numeri x, y, z
3. mediano fra tre numeri x, y, z
4. somma dei due più grandi fra tre numeri x, y, z

8

ASSEGNARE

Svanisce, una volta effettuata un'operazione, la differenza tra conoscenza aritmetica procedurale e conoscenza aritmetica dichiarativa: infatti il risultato di un'operazione viene immagazzinato esattamente nel modo in cui lo sarebbe stato se si fosse cominciato con un elemento di conoscenza dichiarativa.

Douglas R. Hofstadter,
Concetti fluidi e analogie creative

In questo capitolo verranno descritti i meccanismi per definire un ambiente e per modificarlo. Avere a disposizione un ambiente (modificabile) equivale ad introdurre il concetto di "stato di una elaborazione"; ciò costituisce il meccanismo portante di tutta la programmazione a paradigma imperativo. In un linguaggio di programmazione ciò si traduce nell'avere a disposizione delle *variabili* sulle quali poter eseguire delle *assegnazioni*. Questa possibilità permette di creare e gestire liberamente la memoria dell'esecutore e di indirizzare il processo di valutazione di un'espressione e, più in generale, il processo di elaborazione di un insieme di istruzioni.

8.1 Assegnazioni

L'operazione fondamentale sulle variabili, che giustifica il loro uso e che caratterizza tutti i linguaggi di programmazione di tipo imperativo, è quella di *assegnazione* (o *assegnamento*) che consiste nell'attribuire un valore ad una variabile. Tale operazione viene indicata con diversi formalismi, a seconda del linguaggio di programmazione usato [1] . Nel seguito sarà usata la notazione

$$id \leftarrow valore$$

Più in generale un'assegnazione può avere il seguente formato:

$$id \leftarrow espressione$$

Con una tale scrittura si intende che all'identificatore *id* viene associato il valore ottenuto dalla valutazione dell'espressione che compare sulla destra del segno $\leftarrow$. È da tenere presente la sequenza di esecuzione dell'operazione di assegnazione è la seguente:

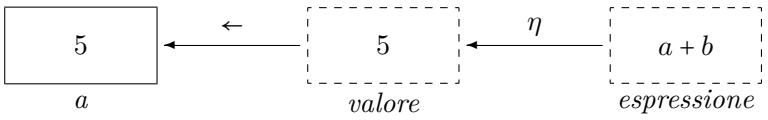
1. elaborazione dell'*espressione* nell'ambiente attuale, ottenendo un valore
2. assegnazione alla variabile *id* del valore ottenuto dalla valutazione

Pertanto, il significato da dare alla precedente scrittura è il seguente:

$$id \leftarrow \eta(espressione)$$

dove con η si denota la funzione di valutazione. Questa osservazione è indispensabile quando nell'espressione compare la variabile da assegnare.

Esempio 8.1.1 - Nell'ambiente $\{(a, 2), (b, 3)\}$ il processo di esecuzione dell'assegnazione $a \leftarrow a + b$ può essere descritto mediante il seguente schema di flusso che viene eseguito da destra a sinistra:



Per seguire il flusso dell'esecuzione delle assegnazioni si può fare uso di una tavola di traccia come la seguente:

	a	b
inizio	2	3
$a \leftarrow a + b$	5	3

[1] Le notazioni più utilizzate per indicare l'operazione di assegnazione sono le seguenti:
id ← *espressione* (notazione generica)
id := *espressione* (linguaggi Algol, Pascal, Ada, Peano, ...)
id = *espressione* (linguaggi BASIC, FORTRAN, C/C++, Java, Python, ...)
set [*id*] *to* [*espressione*] (linguaggi Scratch, Snap!, Blockly, ...)

Le assegnazioni permettono di risolvere semplici problemi; in particolare permettono di spezzare un'espressione complessa in una sequenza di espressioni più semplici.

Esempio 8.1.2 - Un'assegnazione della forma

$$x \leftarrow 100$$

ha l'effetto di inizializzare la variabile x al valore 100. Una successiva assegnazione della forma

$$x \leftarrow x + 1$$

ha l'effetto di incrementare di 1 il valore della variabile x. Una successiva assegnazione

$$y \leftarrow x$$

ha l'effetto di assegnare alla variabile y il valore della variabile x. In questo caso le due variabili x ed y mantengono comunque un'identità distinta ed una modifica ad una delle due non si ripercuote sull'altra. Coerentemente con tutto ciò, un controllo della forma

$$x = y$$

è vero se e solo se le due variabili hanno valori uguali.

Esempio 8.1.3 - Dato un numero naturale di 3 cifre (in base 10), determiniamo il numero più grande che può essere costruito usando le tre cifre del numero dato. Ad esempio, il più grande numero costruibile con le cifre del numero 427 è 742. Un possibile algoritmo è il seguente:

Algoritmo 1 - massimo numero costruibile con le tre cifre di un numero

Input: numero naturale n di tre cifre
Output: massimo numero costruibile con le tre cifre di n

1: scomponi n nelle sue tre cifre a (unità), b (decine), c (centinaia)
2: $p \leftarrow min(a, b, c)$
3: $q \leftarrow max(a, b, c)$
4: $t \leftarrow (a + b + c) - (p + q)$
5: $r \leftarrow q * 100 + t * 10 + p$
6: **return** r

Ad un ulteriore livello di dettaglio questo algoritmo si esprime come segue:

Algoritmo 2 - massimo numero costruibile con le tre cifre di un numero

Input: numero naturale n di tre cifre
Output: massimo numero costruibile con le tre cifre di n

1: $a \leftarrow n \bmod 10$
2: $b \leftarrow (n \operatorname{div} 10) \bmod 10$
3: $c \leftarrow n \operatorname{div} 100$
4: $p \leftarrow if(a < b, if(a < c, a, c), if(b < c, b, c))$
5: $q \leftarrow if(a > b, if(a > c, a, c), if(b > c, b, c))$
6: $t \leftarrow (a + b + c) - (p + q)$
7: $r \leftarrow q * 100 + t * 10 + p$
8: **return** r

8.2 Assegnazioni fra riferimenti ad oggetti

Una differenza significativa fra variabili ed oggetti emerge nelle operazioni di assegnazione e nelle comparazioni.

Se a è un identificatore (riferimento) ed exp è un'espressione che, valutata, produce un oggetto, un'assegnazione della forma

$$a \leftarrow exp$$

ha l'effetto di valutare l'espressione exp generando un oggetto che viene referenziato da a.

Se a è un riferimento ad un oggetto *alfa*, con una successiva assegnazione della forma

$$b \leftarrow a$$

si ottiene l'effetto che anche il riferimento b referenzia l'oggetto referenziato da a; l'oggetto *alfa* risulta accessibile indifferentemente mediante uno dei due riferimenti a o b; la situazione è descritta nella figura 8.1.

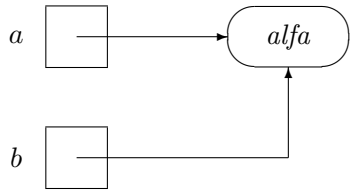


Figura 8.1: Riferimenti a e b che referenziano l'oggetto *alfa*.

Una successiva assegnazione della forma

$$a \leftarrow exp_1$$

dove exp_1 è un'espressione che, valutata, genera un oggetto *beta*, genera la situazione descritta nella figura 8.2.

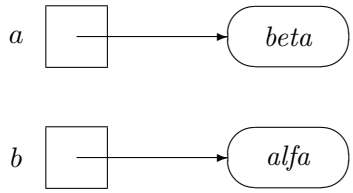


Figura 8.2: Riferimenti a e b che referenziano due oggetti distinti.

Per i riferimenti bisogna distinguere fra le seguenti due forme di comparazione di uguaglianza:

- i due riferimenti referenziano lo stesso oggetto [2]
- i due riferimenti referenziano oggetti uguali [3]

Osservazione. Le due diverse modalità secondo le quali si comportano le operazioni di assegnazione e comparazione fra *variabili che contengono valori* e *riferimenti che referenziano oggetti* vanno sotto il nome di *semantica dei valori* e *semantica dei riferimenti*.

8.3 Asserzioni

L'istruzione di assegnazione produce una modifica dell'ambiente in quanto modifica lo stato delle variabili. Per seguire l'evoluzione dello stato delle variabili si può usare, oltre alle tavole di traccia, le *terne di Hoare* della forma $\{P\}S\{Q\}$ dove P e Q sono asserzioni che descrivono lo stato prima (P) e dopo (Q) delle variabili modificate da S.

Esempio 8.3.1 - Con riferimento all'esempio 8.1.1, l'evoluzione dello stato delle variabili mediante le terne si Hoare si esprime come segue:

$$\{(a,2),(b,3)\}$$
$$a \leftarrow a+b$$
$$\{(a,5),(b,3)\}$$

Le terne di Hoare contengono spesso valori generici denotati mediante degli identificatori, come descritto negli esempi che seguono.

Esempio 8.3.2 - Assegnazione fra variabili:

$$\{(x,\alpha),(y,\beta)\}$$
$$x \leftarrow y$$
$$\{(x,\beta),(y,\beta)\}$$

[2] $a == b$ in Java; $a\ \ is\ \ b$ in Python
[3] $a.equals(b)$ in Java; $a == b$ in Python

Esempio 8.3.3 - Incremento di una variabile numerica:

$$\{(x,\alpha)\}$$
$$x \leftarrow x+1$$
$$\{(x,\alpha+1)\}$$

Esempio 8.3.4 - Scambio di variabili:

$$\{(x,\alpha),(y,\beta)\}$$
$$t \leftarrow x$$
$$\{(x,\alpha),(y,\beta),(t,\alpha)\}$$
$$x \leftarrow y$$
$$\{(x,\beta),(y,\beta),(t,\alpha)\}$$
$$y \leftarrow t$$
$$\{(x,\beta),(y,\alpha),(t,\alpha)\}$$

8.4 Assegnazioni semplici

Nella programmazione vengono spesso utilizzate *assegnazioni semplici* della forma $a \leftarrow b \circ c$, aventi nella parte destra delle *espressioni semplici* costituite da un solo operatore binario $\circ$. Questa forma di espressioni corrisponde alle capacità elaborative di base di un computer che è in grado di eseguire una sola operazione alla volta. Con il ricorso all'assegnazione ed alle variabili ausiliarie per memorizzare i risultati intermedi, un'espressione complessa può essere valutata mediante una sequenza di espressioni semplici.

Molti problemi possono essere risolti in modo sequenziale con l'uso dei soli strumenti costituiti dalle variabili e dall'operazione di assegnazione.

Esempio 8.4.1 - Supponiamo di disporre di un esecutore capace di eseguire solamente le quattro operazioni $+, -, *, \mathsf{div}$ fra numeri naturali. Usando solamente *assegnazione semplici* scriviamo una sequenza di assegnazioni che abbia l'effetto di assegnare alla variabile r il resto della divisione intera fra due numeri naturali m ed n. Il resto della divisione intera fra due numeri naturali m ed n è esprimibile mediante le quattro operazioni aritmetiche fondamentali come descritto nell'algoritmo 3.

Algoritmo 3 - resto della divisione intera fra due numeri naturali

Input: numeri naturali m ed n
Output: resto della divisione intera fra m ed n

1: $r \leftarrow m \,\mathsf{div}\, n$
2: $r \leftarrow r * n$
3: $r \leftarrow m - r$
4: **return** r

Esempio 8.4.2 - Descriviamo un algoritmo per calcolare la somma dei primi n numeri naturali, nell'ipotesi che siano utilizzabili solo assegnazioni semplici. L'espressione della somma è $n * (n+1)\,\mathsf{div}\,2$; da questa espressione può essere ricavato l'algoritmo 4.

Algoritmo 4 - sommatoria di una sequenza di numeri naturali consecutivi

Input: numero naturale n che specifica l'ultimo addendo
Output: sommatoria dei numeri naturali $1 + 2 + \cdots + n$

1: $r \leftarrow n + 1$
2: $r \leftarrow r * n$
3: $r \leftarrow r\,\mathsf{div}\,2$
4: **return** r

8.5 Scomposizione di espressioni

La scomposizione di un'espressione in una sequenza di espressioni semplici costituisce una forma di preelaborazione di un'espressione ed indirizza univocamente il processo di elaborazione, incidendo anche sull'efficienza del processo di valutazione.

Esempio 8.5.1 - L'espressione

$$a^3 - ab^2$$

può essere valutata mediante la sequenza di assegnazioni semplici riportata nell'algoritmo 5.

Algoritmo 5 - valutazione dell'espressione $a^3 - ab^2$

Input: numeri a e b
Output: valore dell'espressione $a^3 - ab^2$

1: $p \leftarrow a * a$
2: $p \leftarrow t * a$
3: $q \leftarrow a * b$
4: $q \leftarrow q * b$
5: $r \leftarrow p - q$
6: **return** r

In alternativa, scomponendo

$$a^3 - ab^2 \rightarrow a(a+b)(a-b)$$

si può scrivere l'algoritmo 6, che risulta equivalente all'algoritmo 5, ma è più efficiente.

Algoritmo 6 - valutazione dell'espressione $a^3 - ab^2$

Input: numeri a e b
Output: valore dell'espressione $a^3 - ab^2$
1: $p \leftarrow a + b$
2: $q \leftarrow a - b$
3: $r \leftarrow a * p$
4: $r \leftarrow r * q$
5: **return** r

Osservazione. Gli esempi precedenti suggeriscono la seguente domanda: *È possibile valutare qualsiasi espressione disponendo di un esecutore capace di valutare solo espressioni semplici?* La risposta affermativa a questa domanda discende direttamente dal meccanismo di costruzione delle espressioni, in base a cui è possibile scrivere un algoritmo che genera un algoritmo per la valutazione di espressioni generali.

8.6 Ottimizzazione nella valutazione

Nonostante l'operazione di valutazione di un'espressione sia una capacità di base di un generico esecutore costituito da un calcolatore, in taluni casi, per avere un diretto controllo sull'esecuzione delle operazioni elementari, si preferisce dirigere direttamente il calcolo, scomponendo l'espressione, al fine di migliorare l'efficienza del calcolo; in particolare si indirizza il procedimento del calcolo con l'obiettivo di minimizzare il numero di moltiplicazioni che vengono eseguite, a costo di aumentare il numero di addizioni. Questo intento è motivato dal fatto che su tutte le architetture attuali di calcolatori l'esecuzione di una moltiplicazione richiede nettamente più tempo di un'addizione.

Esempio 8.6.1 - Un esempio interessante, notato già dal grande matematico Carl Friedrich Gauss (1777-1855), riguarda il prodotto di due numeri complessi $z_1 = a + ib$ e $z_2 = c + id$. Calcoliamo il numero complesso $z_3 = z_1 z_2 = e + if$. Eseguendo il calcolo applicando la regola del prodotto fra due binomi si ottiene:

$$(a + ib)(c + id) \rightarrow ac - bd + (bc + ad)i$$

che porta alla seguente sequenza di assegnazioni che comporta l'esecuzione di 4 moltiplicazioni:

$$e \leftarrow ac - bd$$
$$f \leftarrow bc + ad$$

Un metodo alternativo più efficiente discende dalla seguente uguaglianza:

$$bc + ad = (a + b)(c + d) - ac - bd$$

che porta alla seguente sequenza di assegnazioni che comporta l'esecuzione di

sole 3 moltiplicazioni:

$$\begin{aligned} p &\leftarrow ac \\ q &\leftarrow bd \\ r &\leftarrow (a+b)(c+d) \end{aligned}$$

che vengono poi utilizzate per il calcolo dei coefficienti e e f:

$$\begin{aligned} e &\leftarrow p - q \\ f &\leftarrow r - p - q \end{aligned}$$

Problema 8.6.1 Usando solamente assegnazioni semplici valutare l'espressione

$$1 + x + x^2 + x^3$$

essendo x un valore numerico dato.

Soluzione. Analizzando sequenzialmente i termini dell'espressione si arriva al seguente algoritmo:

Algoritmo 7 - valutazione di un'espressione polinomiale

Input: numero reale x
Output: valore di $1 + x + x^2 + x^3$

1: $r \leftarrow 1 + x$
2: $t \leftarrow x * x$
3: $r \leftarrow r + t$
4: $t \leftarrow t * x$
5: $r \leftarrow r + t$
6: **return** r

Basandosi sulla scomposizione

$$1 + x + x^2 + x^3 = ((x+1)x + 1)x + 1$$

si ottiene la seguente soluzione:

Algoritmo 8 - valutazione di un'espressione polinomiale

Input: numero reale x
Output: valore di $1 + x + x^2 + x^3$

1: $y \leftarrow x + 1$
2: $y \leftarrow y * x$
3: $y \leftarrow y + 1$
4: $y \leftarrow y * x$
5: $y \leftarrow y + 1$
6: **return** y

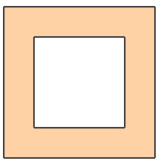
La precedente scomposizione permette di scrivere in notazione funzionale l'espressione, come segue:

$$1 + x + x^2 + x^3 = add(mul(add(mul(add(x, 1), x), 1), x), 1)$$

□

8.7 Corone quadrate

Una *corona quadrata* è una figura piana delimitata da due quadrati concentrici e coassiali, come descritto nella figura che segue.



Si tratta di una figura semplice (essendo delimitata da quadrati) ma sufficientemente articolata da offrire lo spunto per alcuni problemi geometrici risolvibili con istruzioni sequenziali e risulta un'accessibile palestra per esercitarsi sulle assegnazioni. Nei problemi che seguono ammetteremo che siano utilizzabili le quattro operazioni aritmetiche fondamentali e l'operazione di estrazione di radice quadrata.

Problema 8.7.1 Date le lunghezze L ed l dei lati del quadrato grande e piccolo che delimitano una corona quadrata, determinarne l'area.

Soluzione. La soluzione di questo problema deriva direttamente dall'osservazione che l'area della corona quadrata è uguale alla differenza fra l'area del quadrato esterno e l'area del quadrato interno; pertanto il problema è risolto dalla seguente espressione che rappresenta l'area in funzione dei dati:

$$L^2 - l^2$$

Questa espressione può essere descritta mediante un algoritmo in cui compaiono solamente espressioni semplici (algoritmo 9).

Algoritmo 9 - Area di una corona quadrata

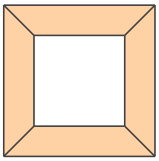
Input: lato L del quadrato esterno, lato l del quadrato interno
Output: area della corona

1: $Q \leftarrow L * L$
2: $q \leftarrow l * l$
3: $a \leftarrow Q - q$
4: **return** a

□

Problema 8.7.2 Data l'area a e la lunghezza b del bordo (interno ed esterno) di una corona quadrata, determinarne lo spessore.

Soluzione. Consideriamo uno dei quattro trapezi che si individuano dividendo la corona in 4 parti equivalenti mediante dei segmenti che uniscono i vertici corrispondenti dei due quadrati che delimitano la corona, come evidenziato nella figura che segue.



Sapendo che l'area di un trapezio è data dalla formula

$$area = \frac{sommabasi * altezza}{2}$$

si ricava

$$altezza = \frac{2 * area}{sommabasi}$$

Questa formula porta direttamente all'algoritmo 10.

Algoritmo 10 - Spessore di una corona quadrata

Input: area a della corona, perimetro esterno p
Output: spessore della corona
1: $areatrapezio \leftarrow a/4$
2: $sommabasi \leftarrow p/4$
3: $s \leftarrow 2 * areatrapezio/sommabasi$
4: **return** s

Un metodo alternativo (algebrico) è il seguente: indicando x il lato del quadrato esterno e con y il quadrato del lato interno, si imposta e si risolve il seguente sistema di due equazioni in 2 incognite:

$$\begin{cases} 4x + 4y &= p \\ x^2 - y^2 &= a \end{cases} \tag{8.1}$$

□

Problema 8.7.3 Data l'area a ed il perimetro esterno p della corona, determinarne lo spessore.

Soluzione. La lunghezza L del lato grande esterno della corona è data dall'espressione

$$L = p/4$$

Indicando con l la lunghezza del lato piccolo interno della corona, vale l'identità

$$L^2 - l^2 = a$$

da cui si ricava

$$l = \sqrt{L^2 - a}$$

Lo spessore s della corona è fornito quindi dall'espressione

$$s = \frac{L - l}{2}$$

Tutti i precedenti passaggi sono riassunti nel seguente algoritmo 11.

Algoritmo 11 - calcolo dello spessore di una corona quadrata

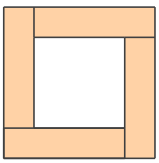
Input: area a della corona, perimetro esterno p
Output: spessore della corona

```
1: L ← p/4              ▷ lato esterno
2: l ← √(L² − a)        ▷ lato interno
3: s ← (L − l)/2        ▷ spessore
4: return s
```

Componendo le assegnazioni riportate nell'algoritmo precedente si nota che l'algoritmo corrisponde alla seguente unica trasformazione funzionale:

$$f(a,p) \mapsto \frac{1}{2}\left(\frac{p}{4} - \sqrt{\left(\frac{p}{4}\right)^2 - a}\right)$$

Un metodo alternativo per il calcolo dello spessore della corona quadrata si basa sulla partizione della corona quadrata in 4 rettangoli uguali:



Indicando con s lo spessore e con L la lunghezza del lato grande esterno, l'area di uno dei quattro rettangoli in cui è suddivisa la corona è data da

$$s(L - s)$$

da cui si ricava l'identità

$$4s(L - s) = a$$

Svolgendo i calcoli si arriva all'equazione di secondo grado in s

$$4s^2 - 4sl + a = 0$$

che risolta fornisce il valore dello spessore s dello spessore:

$$s = \frac{L - \sqrt{L^2 - a}}{2}$$

Essendo $L = p/4$ si deduce che questa formula è equivalente all'algoritmo 11.

ESERCIZI

8.1 Dedurre dal contesto il tipo di ciascuna delle variabili che compaiono nelle seguenti espressioni ed assegnazioni in modo che risultino corrette:

1. $x \leftarrow x < y$
2. $t \leftarrow if(x \wedge (y < z), y + 1, z)$

8.2 Dimostrare che la seguente sequenza di assegnazioni ha l'effetto di assegnare alla variabile p il valore `TRUE` se e solo se r è una radice dell'equazione di secondo grado $ax^2 + bx + c = 0$. Suggerimento: eseguire la traccia dell'esecuzione.

$$\begin{aligned} y &\leftarrow a * r \\ y &\leftarrow y + b \\ y &\leftarrow y * r \\ y &\leftarrow y + c \\ p &\leftarrow y = 0 \end{aligned}$$

Esprimere la precedente sequenza di assegnazioni mediante un'unica istruzione di assegnazione.

8.3 Stabilire qual è l'effetto della seguente sequenza di assegnazioni fra variabili numeriche:

$$\begin{aligned} x &\leftarrow x + y \\ y &\leftarrow x - y \\ x &\leftarrow x - y \end{aligned}$$

8.4 Stabilire se la seguente sequenza di assegnazioni fra variabili numeriche è equivalente a quella riportata nell'esercizio precedente:

$$\begin{aligned} x &\leftarrow x - y \\ y &\leftarrow x + y \\ x &\leftarrow y - x \end{aligned}$$

8.5 Mediante una sequenza di assegnazioni semplici della forma $x \leftarrow p \circ q$, dove x, p, q sono generiche variabili e $\circ$ è un generico operatore aritmetico binario ($+$, $-$, $*$, $/$), valutare nel modo più efficiente possibile le seguenti espressioni. Dire, motivando le affermazioni, se ed in quali casi si possono verificare degli errori nella valutazione delle espressioni in fase di esecuzione.

1. $(a + b) * c - (d/e)$
2. $(a + b)^2 - 2 * (a - b) * (a + b)$
3. $x - ((x/y) * y)$
4. $x^3 + x^2$
5. $a\,x^4 + b\,x^2 + c$
6. $1/(x^4 - x^2 + 1)$

8.6 Dimostrare che qualsiasi espressione algebrica può essere valutata mediante assegnazioni aventi la parte destra costituita da un'espressione semplice.

8.7 Dato un numero naturale di 3 cifre (in base 10), decidere se è composto da tre cifre consecutive, indipendentemente dall'ordine con il quale compaiono. Ad esempio i numeri 453, 102, 765 sono composti da tre cifre consecutive.

8.8 Determinare il più grande numero naturale n composto dalle cifre di un dato numero naturale m composto di al più tre cifre decimali ($0 \leq m \leq 999$). Ad esempio, dato il numero $m = 376$, il risultato è $n = 763$.

8.9 Ad un distributore automatico si possono usare monete da 1 euro, 50, 20, 10, 5 centesimi. Il distributore fornisce delle caramelle del costo di 5 centesimi l'una. Si può introdurre una sola moneta e scegliere il numero di caramelle da acquistare. Assumendo come dati il valore della moneta introdotta (in centesimi di euro) ed il numero di caramelle selezionato, determinare il numero di monete dei diversi tagli che si ottengono come resto, tenendo conto che il distributore dà come resto il minor numero possibile di monete (senza imbrogliare). Ad esempio, introducendo una moneta da 50 centesimi e desiderando avere 3 caramelle, si ottengono 3 monete di resto : una da 20, una da 10 e una da 5 centesimi.

8.10 Mediante delle assegnazioni aventi come parte destra delle espressioni semplici, usando gli operatori *min* e *max* in notazione funzionale, scrivere una sequenza di assegnazioni per valutare l'espressione

$$max(min(a, b), min(c, d))$$

8.11 Mediante delle assegnazioni, usando gli operatori *min* e *max* in notazione funzionale, ordinare quattro valori numerici a, b, c, d.

8.12 In una vasca a forma di parallelepipedo rettangolo, posta a livello e piena d'acqua fino ad una data altezza, viene immerso un corpo pesante (che si adagia sul fondo) a forma di parallelepipedo rettangolo. Determinare la distanza fra il livello dell'acqua ed il bordo della vasca e di quanto sporge o è coperto il corpo immerso.

8.13 Dato il perimetro di un quadrato, determinarne l'area.

8.14 Si consideri la corona circolare delimitata da due circonferenze concentriche, l'una inscritta e l'altra circoscritta ad un medesimo quadrato. Determinare il lato del quadrato conoscendo l'area della corona.

9

SPOSTARE

Solo in casi estremi il materiale è completamente inerte e inutile. Un cavallo intrappolato in un angolo forse un giorno potrà scappare e giocare un ruolo decisivo nella battaglia.

G. Kasparov, *Gli scacchi, la vita*

Spostare un oggetto da una posizione ad un'altra costituisce una delle azioni più frequentemente eseguite, sia in contesti di programmazione che in situazioni della vita reale. I problemi di spostamento si caratterizzano dal contesto in cui avvengono gli spostamenti, dai vincoli imposti agli spostamenti e dall'obiettivo da raggiungere. Combinando più operazioni di spostamento si riesce a risolvere problemi di scambio di due oggetti e, nel caso in cui si definito un criterio d'ordine, ad ordinare due o più oggetti.

Nel suo significato più tradizionale *spostare* significa prendere un oggetto da un posto x e metterlo in un altro posto y. E' importante notare che x ed y sono identificatori che denotano dei *posti* dove si trova e dove mettere l'*oggetto* da spostare; in un situazione di programmazione, x ed y potrebbero essere i nomi di due variabili in cui sono memorizzati dei valori. Uno stesso posto può essere denotato con nomi diversi; ad esempio la frase "Spostare il libro dal tavolo piccolo al tavolo grande" potrebbe essere equivalente, nel caso in cui il tavolo piccolo sia bianco e quello grande grigio, alla frase "Spostare il libro dal tavolo bianco al tavolo grigio"; nel contesto della programmazione "tavolo piccolo" e "tavolo bianco" sarebbero due diversi riferimenti ad uno stesso posto. Si possono avere situazioni in cui il "posto" può diventare l'"oggetto" da spostare come nella seguente situazione: "Scambiare di posto il secondo cassetto con il terzo (con i loro contenuti)".

9.1 Spostare

Guardando all'operazione di spostamento come ad un problema, si riconosce che esso è caratterizzato dai due dati x ed y che rappresentano i nomi che identificano i posti dove eseguire lo scambio. Per indicare che in x è presente un oggetto α useremo la notazione (x, α) mentre per indicare che il posto x è vuoto useremo la notazione $(x, \_)$. Per denotare una situazione articolata, ad esempio che in x è presente un oggetto α ed in y un oggetto β scriveremo $\{(x, \alpha), (y, \beta)\}$.

Per descrivere l'effetto di un'operazione di spostamento basterà indicare la sua precondizione e postcondizione. La postcondizione che si genera a seguito di uno spostamento dipende dal diverso significato attribuito al termine spostare. In particolare si differenzia a seconda che ci si riferisca ad una situazione della vita reale oppure al contesto della programmazione; nel primo caso gli oggetti vengono spostati *fisicamente*: il posto origine viene lasciato vuoto e l'oggetto viene spostato in un altro posto che deve essere vuoto al fine di poter ospitare l'oggetto che viene spostato; nel caso di un linguaggio di programmazione avviene, invece, una *copia*: l'oggetto (valore) continua ad esistere anche nel posto sorgente e l'eventuale valore presente nella destinazione viene sovrascritto e perso oppure viene modificato dal nuovo valore. Per distinguere queste diverse situazioni vengono utilizzati termini e scritture diverse, come descritto nei casi che seguono dove con x ed y vengono denotati il posto destinazione ed il posto origine dello spostamento.

1. *copiare* il valore di una variabile in un'altra: si tratta della tradizionale operazione di *assegnazione* tipica della programmazione. L'operazione è specificata dalla seguente semantica:

$$\begin{gathered}\{(x, \alpha), (y, \beta)\} \\ x \leftarrow y \\ \{(x, \beta), (y, \beta)\}\end{gathered}$$

2. *muovere* un libro da un punto ad un altro della libreria: si ottiene l'effetto che il libro non è più presente nel posto originale e viene trasferito in un altro posto vuoto adatto ad ospitarlo. L'operazione è specificata dalla seguente semantica:

$$\begin{gathered}\{(x, \_), (y, \beta)\} \\ x \leftarrow y \\ \{(x, \beta), (y, \_)\}\end{gathered}$$

   In programmazione questa situazione è ottenibile con un'assegnazione $x \leftarrow y$ seguita da una successiva assegnazione che azzera o annulla la variabile y.

3. *travasare* nella terrina dei tuorli gli albumi sbattuti a neve: si ottiene la situazione di avere una terrina contenente sia i tuorli che gli albumi, mentre l'altra terrina risulta vuota. L'operazione è specificata dalla seguente

semantica:

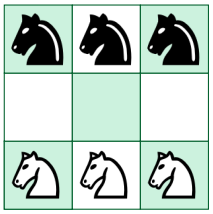
$$\{(x, \alpha), (y, \beta)\}$$
$$x \twoheadleftarrow y$$
$$\{(x, \alpha \cup \beta), (y, \_)\}$$

dove l'operazione $\cup$ di unione deve essere interpretata in base al contesto (addizione fra numeri, unione di due liquidi, ...). Nella programmazione questa situazione si presenta, ad esempio, quando un contenitore di dati viene trasferito in un altro contenitore.

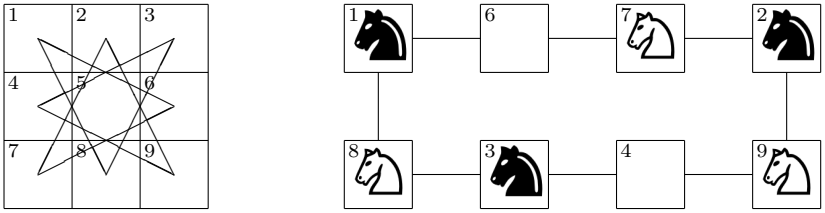
9.2 Spostare i cavalli degli scacchi

Consideriamo il seguente problema:

Su una scacchiera 3×3 *sono disposti tre cavalli bianchi e tre cavalli neri, posti ai lati opposti della scacchiera, come descritto nella figura che segue. Il problema consiste nello scambiare fra loro i cavalli bianchi con quelli neri, nel minor numero possibile di mosse, alternando le mosse del bianco e del nero e rispettando le modalità di movimento dei cavalli secondo le regole del gioco degli scacchi.*



Operando direttamente sulla scacchiera, ci si convince che il problema non è di immediata soluzione. Il problema si sbroglia rappresentando la scacchiera tramite un grafo in cui ogni nodo rappresenta una casella della scacchiera; due nodi del grafo vengono congiunti se si può passare da uno all'altro mediante una mossa del cavallo. Convenendo di etichettare ciascuna casella con un numero naturale da 1 a 9, si ottiene la rappresentazione e successiva trasformazione, come indicato nella figura che segue.



Muovendo i cavalli con riferimento alla figura a destra è facile ricavare la seguente soluzione (fra le tante che esistono):

$$[1 \to 6, 8 \to 1, 3 \to 8, 9 \to 4, 4 \to 3, 2 \to 9, 7 \to 2, 6 \to 7]$$

9.3 Travasi fra recipienti

Nella letteratura degli enigmi logici sono classici i problemi di travaso fra recipienti. Si tratta solitamente di eseguire delle operazioni di travaso fra due recipenti fino ad ottenere una desiderata quantità in uno dei due (o in entrambi). In questi problemi vengono imposte delle condizioni molto restrittive: i recipienti utilizzati non sono graduati ed inoltre non è possibile prendere dei segni di riferimento da un recipiente ad un altro. Vengono inoltre ammesse solo le seguenti operazioni:

1. *riempi* x : riempimento del recipiente x
2. *svuota* x : svuotamento del recipiente x
3. *travasa* x : travasamento del recipiente x all'altro recipiente, fino a che si riempie l'altro recipiente oppure si svuota x

Nelle ipotesi sopra descritte consideriamo il seguente problema:

> *Dati due recipienti da* 5 *e* 3 *litri, eseguire dei travasi in modo da avere* 4 *litri nel recipiente di capacità* 5.

Un metodo per risolvere il problema consiste nel fare qualche operazione e poi proseguire passo-passo, 'a vista', senza una precisa strategia. La bussola di bordo è basata sullo stato attuale, ossia sulla conoscenza della quantità di liquido presente nei due recipienti. Il percorso-risultato viene descritto da una sequenza di passi di transizione di stato. Indicando con A il recipiente da 5 litri e con B quello da 3, una soluzione è sintetizzata nella tavola di traccia descritta nella tabella 9.1.

operazione	A	B
	0	0
riempi B	0	3
travasa B	3	0
riempi B	3	3
travasa B	5	1
svuota A	0	1
travasa B	1	0
riempi B	1	3
travasa B	4	0

Tabella 9.1: Una soluzione del problema di travasi fra recipienti.

Tutto questo lavorio di risoluzione ed il metodo impiegato risultano poco produttivi in quanto il problema affrontato è relativo ad una specifica istanza ed il procedimento adottato non lascia intravvedere alcuna strategia generale.

Per avere una visione più nitida dello spazio di ricerca del percorso, rappresentiamo uno stato mediante una coppia (h, k) che denota che il primo recipiente contiene h unità ed il secondo k. Se le capacità dei due recipienti sono m ed n, lo spazio degli stati può essere rappresentato mediante un reticolo come illustrato nella figura 9.1.

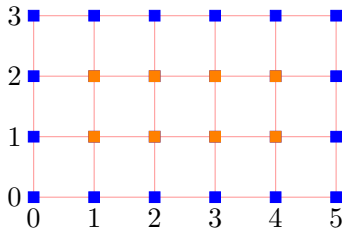


Figura 9.1: Lo spazio degli stati per due recipienti di capienza 3 e 5 unità.

Questa schematizzazione permette di fare delle considerazioni di carattere generale:

1. il punto $(0,0)$ rappresenta lo stato in cui entrambi i recipienti sono vuoti, mentre (m,n) rappresenta la situazione in cui entrambi i recipenti sono pieni
2. dalla stato $(0,0)$ ci si può spostare solo riempendo uno dei due recipenti
3. lo stato (m,n), corrispondente ad avere entrambi i recipienti pieni, rappresenta un punto morto dal quale non si può fare altro che tornare indietro svuotando uno dei due recipenti
4. l'operazione di riempimento di un recipiente è rappresentata da un segmento verticale o orizzontale (a seconda del recipiente) che congiunge un punto dell'asse orizzontale o dell'asse verticale al punto opposto situato sul bordo del reticolo
5. un'operazione di travaso viene rappresentata mediante un segmento orientato, con inclinazione diagonale $\searrow$ oppure $\nwarrow$, che congiunge due stati
6. Le operazioni 1., 2., 3. e 4. indicate sopra portano sempre ad uno stato posto sul bordo del reticolo; i punti interni al reticolo sono irragiungibili

In ogni momento la situazione si può evolvere eseguendo le operazioni *riempi*/*svuota*/*travasa* su uno dei due recipienti; sono quindi potenzialmente eseguibili 6 operazioni; poiché in ogni istante ciascun recipiente è o vuoto o pieno e, quindi, lo spazio degli stati assumibili dal sistema si trovano sul bordo del rettangolo di stati descritti nella figura 9.1; in questo modo le le possibilità si restringono a 4; una di queste è l'operazione inversa dell'ultima eseguita e risulta dunque inefficace nell'avvicinamento all'obiettivo in quanto porterebbe allo stato precedente. Basandosi su queste osservazioni si può descrivere un algoritmo che esamina ad ogni passo le possibili evoluzioni dello stato fino ad incontrare uno stato finale che rappresenta l'obiettivo cercato.

Individuiamo ora un percorso basandosi intanto su una strategia di buon senso. Partendo dalla situazione iniziale in cui entrambi i recipienti sono vuoti, è evidente che il primo passo deve essere il riempimento di uno dei due recipienti. Nello spazio degli stati questi passi corrispondono ad un percorso in orizzontale o in verticale. I successivi passi consistono in operazioni di travaso fra i recipenti, corrispondenti ad un percoso obliquo. Tutto ciò è descritto nella figura 9.2.

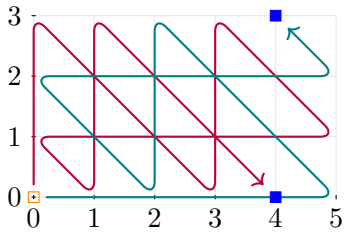


Figura 9.2: Due possibili percorsi risolutivi.

La seconda soluzione (linea azzurra) è descritta mediante la tavola di traccia riportata nella tabella 9.2.

operazione	A	B
	0	0
riempi A	5	0
travasa A	2	3
svuota B	2	0
travasa A	0	2
riempi A	5	2
travasa A	4	3

Tabella 9.2: Un'altra soluzione del problema.

9.4 Scambiare

Solitamente, nei vari ambienti di programmazione ed in altri contesti operativi, l'operazione di scambio non è predisposta in forma atomica, già pronta, ma deve essere realizzata, considerandola come un problema.

Il problema dello scambio di posto di due oggetti viene, nella stragrande maggioranza delle situazioni, liquidato come problema di scambio del contenuto di due variabili. Nonostante questo problema costituisca un banale esercizio di programmazione, esso può diventare lo stimolo per interessanti approfondimenti e, se analizzato a fondo, risulta interessante ed offre svariati spunti anche all'interno della programmazione, dando luogo a diverse soluzioni a seconda della tipologia dei valori contenuti nelle variabili.

Il problema di scambio si basa sui concetti di posizione e di oggetto localizzato ad una data posizione. In generale si tratta di cambiare di posto due oggetti localizzati in due posizioni, in modo tale che, effettuato lo scambio, i due oggetti si trovino uno al posto dell'altro. L'operazione è specificata dalla seguente semantica:

$$\{(x,\alpha),(y,\beta)\}$$
$$y \leftrightarrow x$$
$$\{(x,\beta),(y,\alpha)\}$$

Per poter procedere all'analisi ed alla successiva soluzione di questo problema è necessario precisare le capacità dell'esecutore: supporremo che l'esecutore sia in grado di "spostare un oggetto da un posto ad un altro"; in un linguaggio/ambiente di programmazione questo corrisponde all'operazione di assegnazione fra variabili.

Scambio di variabili

Il problema dello scambio di due variabili consiste nella modifica dei valori contenuti in due variabili, come schematizzato nella figura 9.3.

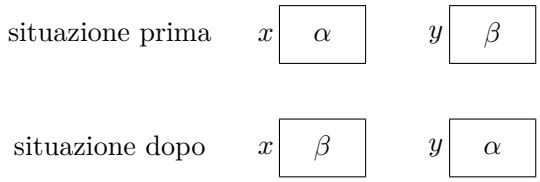


Figura 9.3: Problema dello scambio di due variabili.

Assegnate due variabili x e y scriviamo le istruzioni per scambiare i loro contenuti. Osserviamo che la sequenza di istruzioni $[x \leftarrow y, y \leftarrow x]$ non sarebbe corretta in quanto le azioni vengono eseguite sequenzialmente e pertanto il valore che viene assegnato alla variabile y nella seconda assegnazione è uguale al precedente valore assunto dalla variabile x che è stata precedentemente assegnata uguale al valore iniziale di y. Queste argomentazioni risultano ancora più chiare se si descrive la *tavola di traccia* che permette di seguire passo per passo l'evoluzione dell'ambiente:

istruzione	x	y
inizio	α	β
$x \leftarrow y$	β	β
$y \leftarrow x$	β	β

Il procedimento che esegue lo scambio di due variabili dipende dalla possibilità d'uso di una variabile ausiliaria e dalla tipologia dei valori contenuti. Nel caso di due variabili generiche lo scambio può essere eseguito mediante il ricorso ad una variabile ausiliaria come descritto nell'algoritmo 1.

Algoritmo 1 - scambio di due variabili generiche

Input: variabili x e y
Output: x e y sono scambiate
1: $t \leftarrow x$
2: $x \leftarrow y$
3: $y \leftarrow t$

Una dimostrazione di correttezza di questo algoritmo è fornita dalla seguente tavola di traccia:

istruzione	x	y	t
inizio	α	β	$\perp$
$t \leftarrow x$	α	β	α
$x \leftarrow y$	β	β	α
$y \leftarrow t$	β	α	α

Osservazione. Il problema dello scambio di due variabili come sopra descritto è equivalente allo scambio di due oggetti qualsiasi o allo scambio dei blocchi di testa di due pile.

Scambio di variabili numeriche

Nel caso in cui le variabili x ed y da scambiare siano di un tipo numerico, si può adottare il seguente algoritmo 2 che non richiede l'uso di alcuna variabile ausiliaria. Per convincersene basta generare la tavola di traccia dell'algoritmo, con valori generici delle due variabili x ed y da scambiare.

Algoritmo 2 - scambio di due variabili numeriche

Input: variabili numeriche x e y
Output: x e y sono scambiate

1: $x \leftarrow x + y$
2: $y \leftarrow x - y$
3: $x \leftarrow x - y$

La dimostrazione di correttezza dell'algoritmo può essere basata sulla seguente tavola di traccia:

istruzione	x	y
inizio	α	β
$x \leftarrow x + y$	$\alpha + \beta$	β
$y \leftarrow x - y$	$\alpha + \beta$	α
$x \leftarrow x - y$	β	α

Scambio di variabili booleane

Lo scambio di due variabili booleane x, y può essere svolto mediante il seguente algoritmo.

Algoritmo 3 - scambio di 2 variabili booleane

Input: variabili booleane x, y da scambiare
Output: variabili booleane x, y scambiate

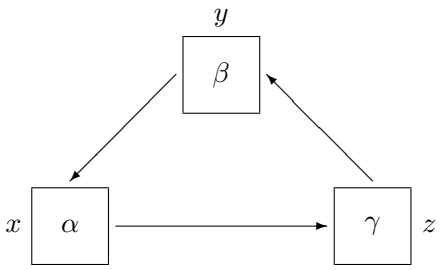
1: $x \leftarrow x \neq y$
2: $y \leftarrow x \neq y$
3: $x \leftarrow x \neq y$

La dimostrazione che questo algoritmo di scambio è corretto può essere svolta mediante una tavola di traccia che evidenzia l'evoluzione dello stato delle variabili in corrispondenza delle possibili instanziazioni iniziali (F*alse*, T*rue*) delle variabili x ed y.

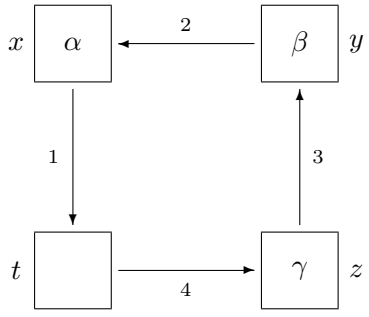
istruzione	x	y	x	y	x	y	x	y
inizio	F	F	F	T	T	F	T	T
$x \leftarrow x \neq y$	F	F	T	T	T	F	F	T
$y \leftarrow x \neq y$	F	F	T	F	T	T	F	T
$x \leftarrow x \neq y$	F	F	T	F	F	T	T	T

Rotazione di variabili

Consideriamo ora il problema di scambio ciclico di tre variabili come descritto dallo schema che segue:



Un procedimento spontaneo di soluzione è quello che usa una variabile ausiliaria t; il risultato desiderato viene raggiunto in quattro passi consecutivi descritti dalla seguente figura:



Le istruzioni che descrivono questo procedimento sono:

$$t \leftarrow x$$
$$x \leftarrow y$$
$$y \leftarrow z$$
$$z \leftarrow t$$

Ci chiediamo ora, nel caso in cui le variabili x, y e z siano numeriche, se lo scambio ciclico di tre variabili sia possibile anche senza l'ausilio di una variabile ausiliaria come sopra. In questo caso non è facilmente applicabile l'accorgimento di somme e sottrazioni applicato al caso precedente dello scambio di due variabili numeriche. Tale difficoltà sorge dal fatto che non si riesce a dominare completamente il problema. Una strada per la soluzione del problema emerge notando che lo scambio ciclico delle tre variabili x, y e z può essere svolto risolvendo i seguenti due sottoproblemi:

P_1 : scambia x con y

P_2 : scambia y con z

Chiaramente le precedenti due istruzioni non costituiscono la soluzione effettiva del problema assegnato in quanto non sono direttamente eseguibili dalla macchina (a meno che non ammettiamo che la macchina sappia *scambiare due variabili* come operazione elementare) ma costituiscono il primo passo verso la soluzione; si potrebbe dire che abbiamo posto un punto di appoggio per la costruzione del cammino risolutivo del problema. La scomposizione in due sottoproblemi (di scambio di due variabili) ha, inoltre, il vantaggio che adesso non è più necessario avere sotto controllo tutto il problema e saper gestire contemporaneamente tre variabili ma è sufficiente focalizzare separatamente sui singoli sottoproblemi che coinvolgono due sole variabili. La soluzione del problema originale si ottiene unendo le due soluzioni dei due sottoproblemi. È questa una situazione ideale per poter facilmente rispondere al quesito sopra postoci. La soluzione di P_1 si esprime come segue:

$$
\begin{aligned}
x &\leftarrow x + y \\
y &\leftarrow x - y \\
x &\leftarrow x - y
\end{aligned}
$$

mentre la soluzione di P_2 come segue:

$$
\begin{aligned}
y &\leftarrow y + z \\
z &\leftarrow y - z \\
y &\leftarrow y - z
\end{aligned}
$$

Data la simmetria con cui compaiono i due operandi x e y nella formulazione del problema P_1, la soluzione complessiva del problema di scambio ciclico può essere espressa anche nella seguente formulazione:

$$
\begin{aligned}
y &\leftarrow y + x \\
x &\leftarrow y - x \\
y &\leftarrow y - x \\
y &\leftarrow y + z \\
z &\leftarrow y - z \\
y &\leftarrow y - z
\end{aligned}
$$

Tale soluzione può essere resa più compatta raggruppando la terza e la quarta assegnazione nella singola assegnazione $y \leftarrow y - x + z$.

9.5 L'aritmetica dei recipienti d'acqua

Consideriamo il seguente problema. Ci troviamo davanti ad una fontana ed abbiamo due recipenti trasparenti, di forme diverse, che denoteremo con A e B, ciascuno contenente dell'acqua. Il problema consiste nello scambiare le quantità d'acqua contenute nei due recipenti, in modo da avere delle quantità invertite rispetto alla situazione di partenza. Ammettiamo che siano possibili, su dei generici secchi x ed y, le seguenti azioni elementari:

- *segna*(x): segna il livello dell'acqua del recipiente x
- *svuota*(x): svuota il recipiente x (gettando l'acqua)
- *riempi*(x): riempie (con accqua della fontana) il recipiente x fino al segno
- *versa*(x, y): versa acqua dal recipiente x fino al segno del recipiente y
- *rovescia*(x, y): rovescia tutta l'acqua dal recipiente x al recipiente y

Supponiamo che i due recipienti siano sufficientemente capienti a contenere delle quantità d'acqua quanto serva. Avendo supposto che i due recipienti possano essere di forme diverse, risulta inutile, oltre che non ammesso in base alle ipotesi fissate sopra, l'eventuale possibilità di segnare su un recipiente una tacca con riferimento al livello contenuto nell'altro recipiente.

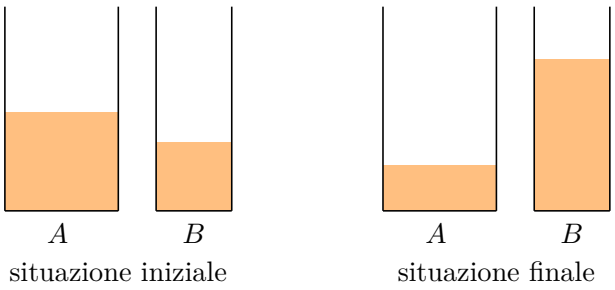


Figura 9.4: Il problema dello scambio dell'acqua di due recipienti A e B.

Se si pensa ad un recipiente come ad una variabile numerica (non negativa) che contiene un valore uguale al numero dei litri in esso contenuti, si rileva facilmente che il problema di scambio del contenuto di due recipienti è analogo a quello di scambio di due variabili (senza fare ricorso ad alcuna variabile ausiliaria). Sulla base di questa analogia potremmo supporre che l'algoritmo risolutivo del problema sia della forma

$$\begin{aligned} A &\leftarrow A + B \\ B &\leftarrow A - B \\ A &\leftarrow A - B \end{aligned}$$

Affinché tali *scritture* costituiscano la descrizione di un algoritmo risolutivo del problema, dobbiamo attribuire ad esse un *significato* ed inoltre l'azione che esse descrivono deve essere eseguibile in termini delle azioni elementari di

cui è capace l'esecutore. Affinché l'algoritmo porti al risultato desiderato, il significato da attribuire alle diverse istruzioni deve essere corrispondente a quanto avviene per le variabili numeriche. Ad esempio, il significato da attribuire alla scrittura

$$A \leftarrow A + B$$

è il seguente:

> metti in A una quantità d'acqua pari a quella che c'era prima in A più la quantità di B (senza modificare la quantità di B)

Notiamo che in questa azione deve essere rispettata la condizione di non-overflow che impone di avere dei recipienti sufficientemente capienti. Osserviamo inoltre che tale azione potrebbe essere erroneamente interpretata con *Travasa in A il contenuto di B* ma, chiaramente, questa azione non rappresenta quello che intendiamo, in quanto comporterebbe lo svuotamento del recipiente B. Risulta dunque chiaro che per poter eseguire l'azione rappresentata da $A \leftarrow A + B$ dovremmo attingere acqua dalla fontana. Similmente, il significato della scrittura

$$B \leftarrow A - B$$

è:

> metti in B una quantità d'acqua pari alla differenza dei due recipienti A e B (senza modificare la quantità di A)

ed il significato della scrittura

$$A \leftarrow A - B$$

è:

> metti in A una quantità d'acqua pari alla differenza dei due recipienti A e B (senza modificare la quantità di B)

In questa due azioni non c'è pericolo di incorrere in una condizione di underflow in quanto, a questo punto, nel problema da noi considerato, si avrà sicuramente che la quantità d'acqua di A è superiore a quella di B.

In definitiva, e azioni elementari che risolvono il compito sono:

- $A \leftarrow A + B$:
  1. $segna(B)$
  2. $rovescia(B, A)$
  3. $riempi(B)$
- $B \leftarrow A - B$:
  1. $segna(A)$
  2. $segna(B)$

3. $svuota(B)$
4. $versa(A, B)$
5. $svuota(B)$
6. $rovescia(A, B)$
7. $riempi(A)$

- $A \leftarrow A - B$:

1. $segna(B)$
2. $svuota(B)$
3. $versa(A, B)$

Il procedimento risolutivo risulta così completamente descritto essendo definito in termini delle azioni elementari che si era ammesso fossero effettivamente eseguibili dall'esecutore.

Osservazione. È interessante notare come un banale esercizio di programmazione relativo allo scambio di due variabili diventi uno spunto di idee profonde. In particolare si possono attivare diverse strategie a seconda del tipo dei valori da scambiare.

9.6 Ordinamento di elementi mediante scambi

Molti procedimenti di ordinamento si fondano sull'operazione di scambio di due elementi. Gli algoritmi di ordinamento diventano interessanti se gli elementi da ordinare sono molti; sarà questo l'argomento di un capitolo più avanti.

Esempio 9.6.1 - Per ordinare due elementi x ed y di un insieme su cui sia definita una relazione d'ordine è sufficiente eseguire uno scambio, usando una variabile ausiliaria, come indicato nell'algoritmo 4.

Algoritmo 4 - ordinamento di due elementi

Input: elementi x e y
Output: x e y sono ordinati
1: **if** $x > y$ **then**
2: scambia $x \leftrightarrow y$
3: **end if**

Un procedimento alternativo è descritto nell'algoritmo 5.

Algoritmo 5 - ordinamento di due elementi

Input: elementi x e y
Output: x e y sono ordinati
1: $t \leftarrow max(x, y)$
2: $x \leftarrow min(x, y)$
3: $y \leftarrow t$

Problema 9.6.1 Ordinare tre elementi x, y, z.

Soluzione. Si può prendere spunto dai due algoritmi visti nell'esempio 9.6.1, ottenendo i due algoritmi 6 e 7.

Algoritmo 6 - ordinamento di tre elementi

Input: elementi x, y, z
Output: x, y e z sono ordinati
1: **if** $x > y$ **then**
2: scambia $x \leftrightarrow y$
3: **end if**
4: **if** $y > z$ **then**
5: scambia $y \leftrightarrow z$
6: **end if**
7: **if** $x > y$ **then**
8: scambia $x \leftrightarrow y$
9: **end if**

Algoritmo 7 - ordinamento di tre elementi

Input: elementi x, y, z
Output: x, y, z sono ordinati
1: $a \leftarrow min(x, y, z)$
2: $b \leftarrow max(x, y, z)$
3: $c \leftarrow x + y + z - a - b$
4: $x \leftarrow a$
5: $y \leftarrow c$
6: $z \leftarrow b$

□

9.7 Il gioco del 15

Una delle più elementari azioni esercitate in un ambiente popolato da oggetti consiste nello spostare un oggetto da una posizione ad un'altra. Solitamente l'operazione di spostamento di un oggetto è inserita in un processo costituito da una sequenza di spostamenti che trasformano una situazione iniziale assegnata in una finale obiettivo.

Intorno al 1870 l'americano Sam Loyd, uno dei più famosi enigmisti ed inventori di giochi, propose il *Gioco del 15*, un puzzle che si gioca su una scacchiera 4×4 dove delle tessere numerate da 1 a 15 vengono fatte scorrere, orizzontalmente e verticalmente, grazie ad una posizione vuota. Il gioco consiste nel rimettere in ordine le tessere partendo da una configurazione casuale.

Figura 9.5: Il *Gioco del 15*. Nella situazione descritta in figura si può eseguire uno dei seguenti spostamenti: muovere la tessera 1 in basso, la tessera 5 a sinistra oppure la tessera 8 in alto.

Loyd mise in palio la cifra di mille dollari come premio per chi fosse riuscito a risolvere la configurazione del gioco costituita dai numeri ordinati a partire da 1 ma con i numeri 14 e 15 scambiati. L'autore sapeva che nessuno avrebbe mai potuto esigere il pagamento del premio in quanto la situazione obiettivo è irragiungibile.

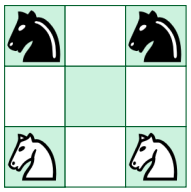
In generale, il problema del gioco del 15 si esprime mediante una delle seguenti due formulazioni:

- stabilire se una configurazione è risolvibile, ossia se esiste una sequenza di mosse che conduce alla situazione ordinata (la configurazione proposta da Loyd non è risolvibile)
- determinare una sequenza di mosse che conduce alla situazione ordinata (possibilmente la più breve)

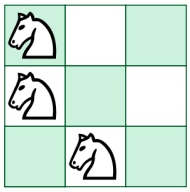
Questi problemi vengono risolti con considerazioni di tipo matematico basate sulle *permutazioni* di elementi.

ESERCIZI

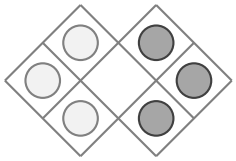
9.1 Su una scacchiera 3×3 sono disposti due cavalli bianchi e due cavalli neri come descritto nella figura che segue. Il problema consiste nello scambiare fra loro i cavalli bianchi con quelli neri, nel minor numero possibile di mosse, alternando le mosse del bianco e del nero e rispettando le modalità di movimento dei cavalli secondo le regole del gioco degli scacchi.



9.2 Su una scacchiera 3×3 sono disposti tre cavalli come descritto nella figura che segue. Il problema consiste nello spostare i cavalli in modo da allinearli (orizzontalmente, verticalmente o lungo una diagonale) nel minor numero possibile di mosse.



9.3 Su una scacchiera come descritta nella figura che segue sono collocate 6 pedine, 3 bianche e 3 nere, disposte come si vede.



Le pedine possono essere spostate rispettando le seguenti regole:

1. una pedina può essere spostata in una casella adiacente, se essa è libera
2. si può saltare una pedina adiacente, purché la casella nella quale si arriva sia libera (senza eliminare la pedina saltata)

Scambiare di posto, nel minor numero possibile di mosse, le pedine bianche con quelle nere.

9.4 Siano A e B due recipienti di capacità m ed n, contenenti delle quantità h e k di liquido ($h \leq m$, $k \leq n$). Discutere, in funzione dei valori di h e k, l'effetto della seguente sequenza di azioni: *travasa*(A), *travasa*(B), determinando le quantità h e k alla fine presenti nei due contenitori.

9.5 Dati due recipienti da 7 e 4 litri, eseguire dei travasi in modo da avere 5 litri nel recipiente di capacità 7.

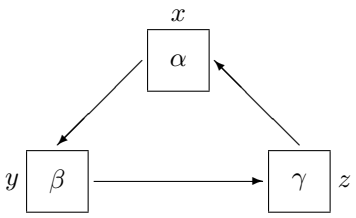
9.6 Nel contesto della programmazione con uso di varibili numeriche, scrivere delle assegnazioni che abbiano l'effetto di rispettare la seguente specifica che descrive un'operazione di "travaso" di valori numerici:

$$\{(x, \alpha), (y, \beta)\}$$
$$\textit{assegnazioni}$$
$$\{(x, \alpha + \beta), (y, 0)\}$$

9.7 Dato un ambiente costituito da tre variabili x, y, z con i valori associati, scrivere una sequenza di assegnazioni che abbia l'effetto di assegnare a ciascuna variabile la somma delle altre due (con riferimento ai valori iniziali dell'ambiente). Svolgere l'esercizio nei seguenti due casi:

1. utilizzando delle altre variabili ausiliarie
2. senza utilizzare altre variabili ausiliarie

9.8 Si vuole ruotare in modo circolare 3 variabili numeriche secondo lo schema sottoriportato.



Risolvere il problema nelle seguenti tre ipotesi:

1. l'esecutore è in grado di eseguire azioni di scambio fra due variabili
2. sono consentite assegnazioni ed è possibile usare delle variabili ausiliarie di appoggio
3. sono consentite assegnazioni ma non è possibile usare variabili ausiliarie di appoggio

Discutere come in questo esercizio si evidenzia il *principio di complessità della soluzione*. Evidenziare dove e come sono intervenute le metodologie *top-down* e *bottom-up*.

9.9 Stabilire se i seguenti due algoritmi sono equivalenti. Suggerimento: usare una tavola di traccia.

Algoritmo A:	Algoritmo B:
scambia $x \leftrightarrow y$	scambia $x \leftrightarrow z$
scambia $y \leftrightarrow z$	scambia $x \leftrightarrow y$

9.10 Dimostrare che i seguenti due algoritmi sono equivalenti e producono l'effetto di ruotare le quattro variabili x, y, z, w in modo circolare in senso antiorario. Suggerimento: usare una tavola di traccia.

Algoritmo A:	Algoritmo B:
scambia $x \leftrightarrow y$	scambia $x \leftrightarrow z$
scambia $y \leftrightarrow w$	scambia $x \leftrightarrow y$
scambia $y \leftrightarrow z$	scambia $z \leftrightarrow w$

9.11 Ruotare in modo ciclico sinistrorso i contenuti di quattro variabili numeriche x, y, z, t, senza fare uso di altre variabili: $x \leftarrow y$, $y \leftarrow z$, $z \leftarrow t$, $t \leftarrow x$. Suggerimento: scomporre il problema in sottoproblemi.

9.12 Ruotare in modo ciclico sinistrorso 3 variabili booleane x, y, z, senza fare uso di altre variabili.

9.13 Discutere se e per quali tipi di numeri la seguente porzione di algoritmo esegue lo scambio delle variabili x e y.

$$x \leftarrow x - y$$
$$y \leftarrow x + y$$
$$x \leftarrow y - x$$

9.14 Discutere se la seguente porzione di algoritmo esegue lo scambio delle variabili booleane x e y.

$$x \leftarrow x = y$$
$$y \leftarrow x = y$$
$$x \leftarrow x = y$$

9.15 Ordinare tre variabili mediante degli scambi.

9.16 Dati 3 oggetti in tre posizioni P, Q e R, ruotarli in modo ciclico secondo lo schema $(P \to Q, Q \to R, R \to P)$. È possibile appoggiare un oggetto su un altro, costruendo delle pile. Non è ammesso usare altre pile ausuliarie.

9.17 Dimostrare che il *gioco del* 15 non ammette soluzione.

Parte III

ROBOTICA EDUCATIVA

10

ROBOT

Gli algoritmi che si incontrano in robotica sono astrazioni che descrivono atti di movimento e di percezione che, quando eseguiti nel mondo reale, consentono di raggiungere obiettivi definiti in termini di oggetti fisici.

K. Goldberg ed altri,
Algorithmic Foundation of Robotics

Il termine "robot" deriva dal termine ceco "robota" che significa "lavoro forzato", "lavoro pesante". Il termine venne usato per la prima volta dal scrittore ceco Karel Capek in un suo romanzo.

Nell'immaginario comune, astraendo da qualsiasi sua realizzazione fisica, un *robot* è un meccanismo autonomo in grado di eseguire delle azioni in un dato ambiente. A differenza di una generica macchina, un robot ha una caratterizzazione fisica, e di conseguenza tecnologica, che deriva dall'esigenza di agire ed interagire in un ambiente.

Lo spazio in cui si muove ed agisce un robot risulta un ambiente in cui sperimentare strategie di comportamento che, alla fine, vengono concretizzate in algoritmi che hanno un immediato aggancio con il mondo fisico. Una efficace sintesi di tutto ciò è fornita dalla citazione di Goldberg riportata sopra.

Con la robotica risulta netta e tangibile la distinzione fra solutore ed esecutore; il solutore percepisce il suo ruolo di "comandante", coerentemente con l'impostazione imperativa di tutta la presentazione.

Questo capitolo si pone l'obiettivo di evidenziare che la robotica, o almeno nella accezione qui proposta, non è un (ulteriore) contenuto e tantomeno una tecnologia, ma è essenzialmente una palestra in cui allenare il pensiero e la mente.

10.1 Automi e macchine

I termini *automa* e *macchina* vengono utilizzati per denotare, in generale, un meccanismo con stato in grado di svolgere delle azioni in modo autonomo e deterministico.

Esempio 10.1.1 - Consideriamo una macchina sequenziale in grado di memorizzare un valore numerico naturale. Da questa definizione si capisce che questa macchina assolve alle tradizionali funzionalità di una *variabile* dei linguaggi di programmazione. Lo stato della macchina coincide con il valore memorizzato internamente. Questa macchina risulta caratterizzata dalle seguenti operazioni e comandi [1]:

- `SET`(n): memorizza nella variabile il valore n (l'eventuale valore precedentemente memorizzato viene sovrascritto e perso)
- `GET`: ritorna il valore memorizzato nella variabile

Usando queste operazioni di base risulta possibile costruire delle istruzioni più articolate, ad esempio `SET`$(2 * $`GET`$())$ cha ha l'effetto di raddoppiare il valore memorizzato nella variabile.

Basandosi sulle operazioni elementari è possibile realizzare un insieme di operazioni che forniscono nuove funzionalità; in questo modo una *variabile* può essere "estesa" e diventa un *contatore* in grado di contare:

- `INC`: incrementa di un'unità il valore
- `DEC`: decrementa di un'unità il valore
- `VAL`: valore attuale del contatore

□

10.2 Robot

Una definizione del termine robot che coglie efficacemente il significato che noi oggi assegnamo a questo termine, in particolare nel contesto della Robotica Educativa, la possiamo dedurre da quanto affermava George Bekey agli inizi di questo secolo: definiva un robot come *una macchina che sente, pensa ed agisce*. Questa definizione ha il pregio di individuare le tre principali componenti di un robot:

- *sensori*: sono le componenti corrispondenti ai sensi dell'uomo (tatto, vista, udito); permettono di acquisire informazioni dall'ambiente circostante (sensori di contatto, di vista, di colore, di suono, di temperatura, ...)
- *attuatori*: sono le componenti che permettono di agire sull'ambiente esterno, azionando motori, ruote, pinze, ...

[1] Nella definizione delle operazioni che seguono supponiamo di operare su una sola variabile, in modo da non avere la necessità di far comparire il nome della variabile; equivalentemente possiamo supporre di utilizzare una notazione orientata agli oggetti, nel qual caso le varie funzioni e comandi dovranno essere preceduti dal nome della variabile, mediante la tradizionale notazione puntata, come ad esempio nella scrittura x.`SET`(n).

- *sistema di controllo*: espleta le funzioni che in un essere umano sono svolte dal cervello; permette di eseguire delle elaborazioni e di prendere delle decisioni su come/quando azionare gli attuatori

Un robot va opportunamente programmato in modo da definirne il comportamento che deve esibire, a seconda del contesto esterno che incontrerà.

Le caratteristiche sopra descritte sono sintetizzate nello schema riportato nella figura 10.1.

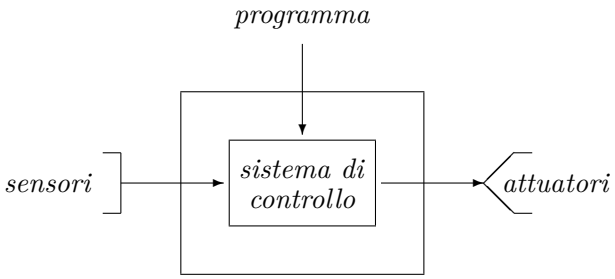


Figura 10.1: Struttura generale di un robot.

Tipologie dei robot

Nella letteratura tecnica e scientifica i robot vengono classificati in base a criteri costruttivi, come segue:

- robot *fisico*: oggetto reale, costituito da componenti meccaniche ed elettroniche, che si muove nello spazio fisico (su un tavolo, in una stanza, sul terreno, nello spazio)
- robot *virtuale*: oggetto grafico che si muove sul video di un computer, con il quale si interagisce mediante l'uso di periferiche (tastiera, mouse, joistick, ...)

Oggigiorno la dicotomia fisico-virtuale diventa sempre più labile; da un punto di vista didattico e metodologico e dal punto di vista della sua programmazione risulta più utile la seguente classificazione:

- robot *meccanico*: meccanismo (fisico o virtuale) che esegue con velocità e precisione delle azioni ripetitive
- robot *sensibile*: costituisce un'evoluzione del robot meccanico in quanto è dotato di *sensori* (fisici o virtuali) che gli permettono di percepire alcuni parametri (temperatura, luce, pressione, ...) dell'ambiente esterno
- robot *intelligente*: entità che reagisce a delle situazioni esterne, in modo tale da *sembrare* [2] un essere intelligente e dotato di volontà.

[2]Dietro alla parola *sembrare* si nascondono delicate questioni e difficili problemi di ordine tecnico, sociale, culturale e filosofico. Qui il discorso si fa estremamente impegnativo ed arduo in quanto si entra nelle tematiche e problematiche relative alla cosidetta *intelligenza artificiale.*

Le potenzialità del robot dipendono principalmente dalle capacità elaborative interne, dalla tipologia e dalla sensibilità, precisione e potenza dei sensori e degli attuatori (di movimento, di disegno e di altre tipologie).

Osservazione. Per studiare e gestire un robot si devono fare alcune ipotesi semplificative sulla sua struttura e sulle sue capacità. Nei capitoli che seguono verranno descritte diverse tipologie di robot, che si possono collocare all'interno di un intervallo di complessità strutturale avente a sinistra i robot più elementari. Questo intervallo ha all'estremo inferiore di massima semplificazione un robot puntiforme, privo di sensori ed attuatori, in grado di avanzare ed invertire il senso di marcia; dall'altra banda dell'intervallo, che possiamo considerare *aperto a destra*, si trovano i robot con una struttura molto articolata, corredati di sensori ed attuatori di diverse tipologie.

Programmabilità dei robot

Un robot può essere programmato in diverse modalità; si possono evidenziare (almeno) i seguenti paradigmi:

- paradigma *strumento*: similmente a come utilizziamo una penna per scrivere o una bicicletta per spostarci, l'uomo aziona direttamente lo strumento in base alla propria volontà e scelta; i comandi dati uno alla volta dall'esterno; lo strumento diventa una protesi del corpo dell'uomo e viene controllato dal cervello dell'uomo. La situazione è analoga alla movimentazione di un'automobilina mediante un telecomando o un joistick
- paradigma *solutore-esecutore*: il solutore istruisce l'esecutore sul come deve agire e comportarsi; una volta istruito e attivato, l'esecutore agisce autonomamente e l'uomo perde qualsiasi controllo su di esso. La programmazione deve prevedere la gestione di situazioni non note a priori. È questa la situazione più interessante dal punto di vista algoritmico e per le applicazioni dei robot reali.
- paradigma *auto-apprendimento*: viene precisato l'obiettivo ed il robot cerca autonomamente di raggiungerlo, scegliendo una adeguata strategia di comportamento, imparata autonomamente [3] dalle precedenti esperienze che il robot ha avuto.

10.3 La robotica

La robotica coinvolge diverse discipline; ciò viene evidenziato dalle seguenti affermazioni:

1. un *robot* è una macchina in grado di esibire un comportamento autonomo
2. un robot è in grado di svolgere un limitato insieme di *azioni elementari* che, combinate opportunamente possono conseguire obiettivi anche avanzati
3. il comportamento del robot viene definito dal *solutore* mediante un *algoritmo*

[3] Anche qui il discorso si complica: tutto ruota attorno al significato che vogliamo attribuire ai termini "imparare autonomamente".

4. per essere eseguite, le azioni descritte nell'algoritmo vengono tradotte in un *programma* scritto mediante uno specifico *linguaggio di programmazione*
5. un robot agisce in un ambiente o *spazio*; l'analisi dello spazio comporta l'approfondimento delle sue caratteristiche geometriche descritte in termini di proprietà mediante il linguaggio della Matematica

10.4 Dati, strutture, istruzioni, comandi, processi

In molte argomentazioni dell'Informatica (nei Sistemi Operativi, nella Robotica) sorge l'esigenza di differenziare il concetto di "azione" da quello di "esecuzione di un'azione". Trasversalmente a questa distinzione si inserisce la distinzione fra il concetto di "dato elementare" e "struttura di dati". Tutte queste considerazioni e classificazioni vengono precisate nelle seguenti definizioni.

I *dati* sono le unità di informazione; possono essere distinti in

- *valori*: 23, `TRUE`, '`ciao`'
- *azioni*: *mangiare la mela*, *fare dietro-front*

I dati possono essere organizzati in *strutture* composte da più dati; esempi:

- *sequenza*: sequenza di numeri $[3, 4, 2, 4]$, sequenza di figure $[\triangle, \square, \bigcirc, \square]$
- *insieme*: insieme di numeri $\{2, 3, 5, 7\}$, insieme di figure $[\triangle, \square, \bigcirc]$

Un *programma* è costituito da una struttura di azioni. Un esempio di programma è il seguente:

$$[\textit{sbucciare la mela}, \textit{tagliare a fettine}, \textit{mangiare le fettine}]$$

Oltre a questa semplice strutturazione in *sequenza*, nei linguaggi di programmazione sono disponibili ulteriori forme quali: strutture *condizionali*, strutture *cicliche*, ed altre ancora.
Un *comando* è l'invito rivolto ad un esecutore ad eseguire un'azione; l'esecuzione di un'azione provoca un effetto; esempi:

- $\textit{visualizza}(23)$
- $\textit{esegui}(\textit{mangiare la mela})$

Nel linguaggio naturale la richiesta di esecuzione di un'azione viene solitamente sottolineata dal modo imperativo della formulazione, magari enfatizzato dal punto esclamativo finale (*mangia la mela*!). In Informatica, se x è una generica azione o un programma, con $\textit{exec}(x)$ si denota il comando che consiste nel richiedere l'esecuzione di x. L'effettiva esecuzione in un dato momento, da parte dell'esecutore, di x genera un *processo*.

Osservazione. La distinzione fra dati e comandi corrisponde alla distinzione nei linguaggi di programmazione fra sottoprogrammi che eseguono delle elaborazioni ritornando dei valori come risultato e sottoprogrammi che eseguono comandi provocando degli effetti.

10.5 Esempi di robot

Ogni robot è caratterizzato da:

1. operazioni elementari che è in grado di eseguire
2. insieme di sensori in dotazione
3. caratteristiche dell'ambiente in cui opera

Un robot puntiforme di nome Puntino

Consideriamo un robot estremamente elementare, dotato di nessuna consistenza fisica, in grado di muoversi lungo delle posizioni su una retta. Le posizioni sono individuate dai numeri naturali 0, 1, ... Inizialmente Puntino si trova nella posizione 0 dell'origine. Puntino è dotato di un verso di avanzamento che inizialmente lo porta verso posizioni di coordinata maggiore (verso destra). La situazione è descritta nella seguente figura:

Puntino è comandabile mediante un insieme di comandi elementari che permettono di avanzare di 1 passo (passare alla posizione successiva), di invertire il senso di marcia. Queste azioni corrispondono all'insieme $\mathcal{M} = \{\texttt{A}, \texttt{I}\}$ di azioni elementari [4] corrispondenti ciascuno ai seguenti movimenti:

- `A`: avanza di un passo nella direzione corrente
- `I`: inverti il senso di avanzamento

Oltre alle precedenti due istruzioni elementari `A` ed `I`, Puntino è in grado di fornire delle informazioni sul proprio stato attuale, mediante le seguenti funzioni elementari che ritornano un numero naturale:

- `N`: numero corrispondente alla posizione attuale
- `V`: verso attuale (0: verso sinistra, 1: verso destra)
- `S`: test di segno (0 se `N`=0, 1 se `N`>0)
- `Z`: test di zero (0 se `N`>0, 1 se `N`=0)

Queste funzioni permettono la definizione di programmi più articolati, in grado di gestire delle condizioni ed effettuare degli spostamenti in funzione dello stato corrente.

Durante il suo percorso Puntino conta tutte le operazioni elementari che esegue e ne memorizza il numero in due contatori:

- `P`: numero di avanzamenti eseguiti
- `Q`: numero di inversioni eseguite

[4] In alternativa alle operazioni elementari `A` e `I` si potrebbero considerare le operazioni `L` e `R` aventi l'effetto di spostare Puntino di una posizione a sinistra o destra.

Quadretto: un robot che si muove nel piano quadrettato

Generalizzando, passando dalla dimensione 1 dello spazio-retta dove si muove Puntino, consideriamo un analogo robot, denominato Quadretto, che si muove su uno spazio bidimensionale, discreto e finito. La situazione è descritta nella figura che segue. Tale tipologia di robot sarà descritta ed analizzata in un successivo capitolo (*Muovere*), dopo che nel capitolo *Spazio* sarà analizzato lo spazio in cui si muoverà Quadretto.

Consideriamo lo spazio a scacchiera descritto nel capitolo precedente: una porzione limitata di piano, suddiviso in un reticolo di 8×8, caselle unitarie, ciascuna individuata da una coppia di numeri interi, secondo il metodo delle coordinate cartesiane. In questo spazio si muove un robot elementare che in ogni istante occupa una casella, è in grado di avanzare di un passo alla volta, ha una direzione e verso di avanzamento ed è in grado di ruotare, a sinistra ed a destra, di un angolo retto. Chiamiamo questo robot con il termine *Quadretto*. Come ogni robot, Quadretto ha una direzione di avanzamento; assumeremo l'ipotesi che le possibili rotazioni siano ad angolo retto (a sinistra o a destra).

Lo *spazio di movimento* di Quadretto risulta essere una porzione di piano discreto suddiviso in caselle quadrate. La situazione è descritta nella figura 10.2.

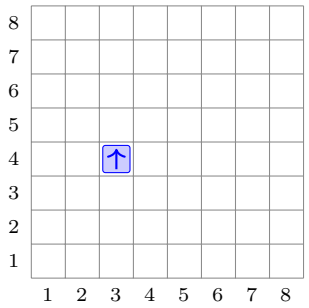


Figura 10.2: Il robot *Quadretto* nel suo ambiente. La freccia denota la direzione di avanzamento.

Quadretto è comandabile mediante un insieme di comandi elementari che permettono di avanzare di 1 passo (passare alla casella davanti), di indietreggiare di 1 passo (passare alla casella dietro), e di ruotare a sinistra o destra di un angolo retto.

Quadretto è in grado di eseguire dei movimenti elementari che vengono descritti, rispetto alla propria posizione ed al proprio verso di avanzamento, dall'insieme di valori $\mathcal{M} = \{\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}\}$ corrispondenti ciascuno ai seguenti movimenti:

- `F` : avanzamento di un passo
- `B` : indietreggiamento di un passo
- `L` : rotazione a sinistra di un angolo retto
- `R` : rotazione a destra di un angolo retto

10.6 Programmi per i robot

Combinando le istruzioni elementari si costruiscono i *programmi*. I robot eseguono le azioni descritte nel programma una alla volta, in sequenza. Ci si potrebbe quindi attendere che un programma debba essere composto da una sequenza di comandi e, quindi, da una stringa. Fortunatamente, i linguaggi offrono dei meccanismi descrittivi che consentono al solutore di accorciare i programmi condensandoli in stringhe molto più corte che vengono poi espanse al momento dell'esecuzione. Infatti, un programma può essere definito ricorrendo ad operazioni che combinano porzioni di programma per generare un programma più articolato. La costruzione dei programmi si basa sull'applicazione combinata di pochi schemi di base, come descritto nei seguenti sottoparagrafi. Negli esempi che seguono faremo riferimento al robot Quadretto.

Azioni elementari

La forma più elementare di programma è costituito da una singola azione elementare.

Esempio 10.6.1 - Le azioni elementari `F`, `B`, `L`, `R` di Quadretto costituiscono quattro programmi. □

Concatenazione di azioni elementari

Se a ed b sono due azioni elementari, mediante la loro *concatenazione*, scritta nella forma $a + b$, o semplicemente nella forma ab, si ottiene il programma costituito dalle azioni a e b che verranno eseguite nell'ordine indicato: prima a e poi b. La concatenazione di azioni elementari può essere generalizzata ad una sequenza di azioni elementari a_1, a_2, ..., a_n, e la scrittura

$$a_1 a_2 \ldots a_n$$

denota il programma costituito dalle sequenza delle azioni indicate, che verranno eseguite in sequenza, una dopo l'altra.

Esempio 10.6.2 - Nel caso di Quadretto, `FFFRFF` denota il programma che fa avanzare di 3 passi, ruotare a destra ed avanzare di altri 2 passi. □

Ripetizione di azioni elementari

Se n è un numero naturale ed a una generica azione elementare, con la scrittura

$$n\,a$$

si denota il programma costituito dalla concatenazione di n azioni a, ossia $aaa \ldots a$, n volte a.

Esempio 10.6.3 - La scrittura `5F` denota il programma che fa avanzare di 5 passi. □

Sequenze di azioni elementari

Un programma può essere costituito da *sequenze* di azioni elementari. Una sequenza di n azioni elementari $a_1, a_2, \ldots a_n$ viene indicato nella forma parentizzata $[a_1, a_2, \ldots, a_n]$.

Esempio 10.6.4 - La scrittura `[F,F,L,F,F]` denota il programma che fa eseguere un percorso ad angolo verso sinistra, con entrambi i lati di lunghezza 2 passi. □

I meccanismi di *concatenazione*, *ripetizione* e *sequenziazione* sono generali e possono applicarsi non solo alle azioni elementari ma a generici programmi, come descritto a seguire.

Concatenazione di programmi

Due programmi α e β possono essere *concatenati* per produrre come risultato un programma, denotato con $\alpha + \beta$, costituito dalle istruzioni del primo programma, seguite dalle istruzioni del secondo programma. Nel caso in cui i due programmi α e β non contengano identificatori si può eseguire direttamente l'operazione di concatenazione e l'operazione $\alpha + \beta$ può essere scritta nella forma $\alpha\beta$.

Esempio 10.6.5 - L'operazione `FLF+RFFF` produce come risultato il programma `FLFRFFF`. □

Sequenziazione di programmi

In generale, gli elementi di una sequenza possono essere dei generici programmi, secondo la seguente definizione. Se α_1, α_2, ..., α_n sono programmi, con

$$[\alpha_1, \alpha_2, \ldots, \alpha_n]$$

si denota il programma costituito dalla concatenazione dei programmi α_i. Nel caso in cui i programmi α_i siano costanti, cioè senza identificatori, si può scrivere più semplicemente $\alpha_1\alpha_2 \ldots \alpha_n$.

Esempio 10.6.6 - Componendo in sequenza i programmi `3F`, `R`, `FF` si ottiene il programma

$$[\texttt{F}, \texttt{R}, \texttt{FF}]$$

che equivale al programma `FRFF`. □

Ripetizione di programmi

Coerentemente con la sua natura di indefesso esecutore, un robot può essere programmato ricorrendo a dei controlli che permettono di *ripetere* delle azioni. Si può ripetere una singola istruzione o un programma: se n è un numero naturale ed α è un programma, con

$$n * \alpha$$

si ottiene il programma costituito dalla concatenazione di n programmi α.

Esempio 10.6.7 - Il programma

$$[3\texttt{F} + 2\texttt{RF} + 4\texttt{LF}]$$

è equivalente al programma `FFFRRLFLFLFLF`. □

Nel caso in cui n sia una costante ed α un programma costante si può omettere il segno di moltiplicazione. Nel caso in cui α sia un singolo comando o un programma della forma $[\dots]$, il segno di operazione $*$ può essere omesso. L'operazione di composizione $[\dots]$ ha priorità massima rispetto alle altre operazioni, mentre l'operazione $*$ di ripetizione ha priorità rispetto all'operazione + di concatenazione.

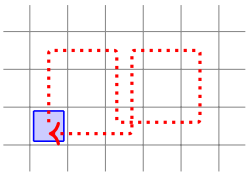
Esempio 10.6.8 - Il programma $4*\texttt{F}$ può essere scritto semplicemente come 4F e $3*[\texttt{F},\texttt{F},\texttt{R}]$ come $3[\texttt{F},\texttt{F},\texttt{R}]$ (ed anche come 3[FFR]). Considerando le priorità degli operatori, l'espressione 3*[F,R] produce il programma FRFRFR mentre 3*FR+FF produce il programma FFFRFF. □

I meccanismi di *concatenazione*, *ripetizione* e *sequenziazione* possono essere combinati fra loro, in tutte le combinazioni possibili; si possono ottenere così programmi formati, ad esempio da "sequenze di ripetizioni di sequenze" e strutture simili.

Esempio 10.6.9 - Il programma

$$[3[2\texttt{F},\texttt{R}],2\texttt{L},4[2\texttt{F},\texttt{L}],2\texttt{R},3\texttt{F}]$$

muove Quadretto su un percorso ad "otto", come descritto nella figura che segue.



□

10.7 Inversa di un'azione

Con θ indichiamo l'*azione nulla* ossia l'azione che non comporta alcun cambiamento di stato. Per una generica azione x, indichiamo con $inv(x)$ o con $-x$ l'azione *inversa di* x che, per definizione, è l'azione tale che, eseguita dopo x ripristina lo stato precedente l'esecuzione, ossia:

$$x(-x) = \theta$$

Esempio 10.7.1 - Ciascuno dei movimenti elementari di Quadretto ammette un movimento inverso descritto dalla seguente tabella:

m	$inv(m)$
F	B
B	F
L	R
R	L

□

PROPRIETÀ 1. Data una generica azione x, vale la seguente proprietà:

$$inv(inv(x)) = x$$

□

PROPRIETÀ 2. Date delle azioni elementari $x_1, x_2, \ldots, x_n$, vale la seguente proprietà:

$$inv([x_1, x_2, \ldots, x_n]) = [inv(x_n), inv(x_{n-1}), \ldots, inv(x_2), inv(x_1)]$$

ossia, ricorrendo al funzionale *map*:

$$inv([x_1, x_2, \ldots, x_n]) = rev(map(inv, [x_1, x_2, \ldots, x_n]))$$

□

Esempio 10.7.2 - Per Quadretto:

$$inv(\texttt{2FRF}) = inv(\texttt{F})inv(\texttt{R})\texttt{2}\ inv(\texttt{F})$$

□

Come conseguenza della precedente proprietà si ha la seguente

PROPRIETÀ 3. Data una generica azione x ed un numero naturale n, vale la seguente proprietà:

$$inv(n\ x) = n\ inv(x)$$

Dimostrazione.

$$inv(n\ x) = inv([x, x, \ldots, x]) = [inv(x), \ldots, inv(x)] = n\, inv(x)$$

□

10.8 Algebra dei programmi

Le operazioni di concatenazione +, ripetizione $*$ e di aggregazione $[\ldots]$ sopra descritte costituiscono una particolare algebra con regole di trasformazione e semplificazione che per programmi costanti possono essere riassunte come segue:

Regole di trasformazione:

1. è lecito sostituire n caratteri x consecutivi con nx
2. è lecito sostituire n parole α consecutive con $n[\alpha]$
3. è lecito sostituire $n\alpha + m\alpha$ con $(m+n)\alpha$
4. è lecito sostituire $\alpha + \beta$ con $\alpha\beta$
5. è lecito sostituire $[\alpha_1, \alpha_2, \ldots \alpha_n]$ con $\alpha_1\alpha_2\cdots\alpha_n$

Con queste regole si possono costruire programmi di movimento che sono del tutto equivalenti ad una struttura dati composta da atomi costituiti dai movimenti elementari.

Esempio 10.8.1 - Con riferimento a Quadretto, a seguire sono riportate, in corrispondenza di ciascune delle regole di trasformazione riportate sopra, una loro applicazione:

$$\begin{aligned} \texttt{FFRFFF} &\rightarrow \texttt{2FR3F} \\ \texttt{FRFFRFFRFF} &\rightarrow \texttt{F3[R2F]} \\ \texttt{2[FL]} + \texttt{3[FL]} &\rightarrow \texttt{5[FL]} \\ \texttt{FLFFR} + \texttt{RFRL} &\rightarrow \texttt{FLFFRRFRL} \\ \texttt{[F,F,R,F]} &\rightarrow \texttt{FFRF} \end{aligned}$$

□

Semplificazione di un programma

Semplificare un programma significa trasformarlo in uno equivalente di complessità minore.

Problema 10.8.1 Semplificare il programma `[[F,F,R]+[L,F,F,R]]`.

Soluzione. : Semplificando:

$$\texttt{[F,F,R]} + \texttt{[L,F,F,R]} \rightarrow \texttt{FFR} + \texttt{LFRF} \rightarrow \texttt{2FRLFRF} \rightarrow \texttt{3FRF}$$

□

Problema 10.8.2 Muovere Quadretto 3 *passi indietro*.

Soluzione. Analizziamo la frase *Fai 3 passi indietro*. Può essere scomposta in due diversi modi, come segue:

1. fai (3 passi indietro)
2. per 3 volte (fai 1 passo indietro)

Queste due diverse scomposizioni possono essere sviluppate rispettivamente come segue:

1. [*inverti*, 3∗*avanti*, *inverti*]
2. per 3 volte [*inverti*, *avanti*, *inverti*]

Queste due scomposizioni portano alle due diverse soluzioni fornite dai seguenti due programmi P_1 e P_2 funzionalmente equivalenti:

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \texttt{[2R,3F,2L]} \\ P_2 &\stackrel{\text{def}}{=} \texttt{3[2R,F,2L]} \end{aligned}$$

Questi due programmi hanno le seguenti complessità:

$$\begin{aligned} compl(P_1) &= 2 + 3 + 2 = 8 \\ compl(P_2) &= 3(2 + 1 + 2) = 15 \end{aligned}$$

Da ciò si conclude che P_1 è più efficiente rispetto a P_2. Il programma P_1 tiene traccia della scomposizione del problema; può essere scritto in forma più compatta con la stringa `2R3F2L`. Il programma P_2 può essere semplificato come segue:

```
3[2R,F,2L]  →  2RF2L2RF2L2RF2L
            →  2RFFF2L
            →  2R3F2L
```

□

Esempio 10.8.2 - La seguente sequenza di trasformazioni illustra l'applicazione delle proprietà delle operazioni di addizione e moltiplicazioni fra sequenze di istruzioni:

```
[F]+2*[R,2*F]+[R,F]  →  [F]+2*[R,F,F]+[R,F]
                     →  [F]+[R,F,F,R,F,F]+[R,F]
                     →  [F,R,F,F,R,F,F,R,F]
                     →  3*[F,R,F]
                     →  3[FRF]
```

□

10.9 Un linguaggio per comandare un robot

Come avviene per ogni linguaggio si devono precisare le parole di base costituenti il linguaggio (assiomi) e le regole grammaticali (regole di produzione) mediante le quali generare le frasi corrette del linguaggio. Nel caso di Puntino e Quadretto le parole di base corrispondono ai movimenti (avanzamento e rotazioni). Un programma può essere costruito direttamente, partendo dagli assiomi, usando le regole del linguaggio. Un semplice linguaggio $\mathcal{L}$ è definito informalmente come segue:

1. *Assiomi*:
   a) ogni azione elementare è un programma
2. *Regole di generazione (produzione)*:
   b) la concatenazione di due programmi è un programma
   c) è lecito mettere una coppia di parentesi [ ] attorno ad un programma
   d) è lecito mettere un numero naturale davanti ad un programma

Oltre a queste regole verranno definite, nel paragrafo 10.8, ulteriori regole di trasformazione.

Osservazione. La modalità descrittiva indicata sopra per descrivere un linguaggio è la più immediata ed informale che si possa utilizzare; avremmo potuto utilizzare una modalità più criptica, derivata dal tradizionale linguaggio della Matematica, definendo il linguaggio $\mathcal{L}$ come segue:

a) $c \in \{\texttt{A}, \texttt{I}\} \Rightarrow c \in \mathcal{L}$

b) $\alpha, \beta \in \mathcal{L} \Rightarrow \alpha\beta \in \mathcal{L}$

c) $\alpha \in \mathcal{L} \Rightarrow [\alpha] \in \mathcal{L}$

d) $n \in \mathbb{N}, \alpha \in \mathcal{L} \Rightarrow n\alpha \in \mathcal{L}$

In questo modo avremmo fatto solo un piccolo passo in avanti, non decisivo, in quanto avremmo mantenuto delle ambiguità ed aggiunto delle incoerenze (ad esempio, in questo nuovo formalismo, avremmo mischiato il concetto di *carattere* con il concetto di *stringa*, e mischiato il concetto di *numero* con quello di *rappresentazione* di un numero). Tutte queste ambiguità vengono risolte ricorrendo alla teoria dei "linguaggi formali"; ma al momento non ci serve.

Esempio 10.9.1 - Basandosi sulle regole definite sopra si possono generare i seguenti programmi:

```
F
FRFL
3[12FR]2R[3F2L]
```

□

La generazione di un programma può essere descritta mediante un *albero sintattico* come descritto nell'esempio che segue.

Esempio 10.9.2 - Utilizzando le regole sopra definite il programma `3[FR]4[L2F]` può essere generato mediante un processo di generazione descritto dal seguente albero sintattico:

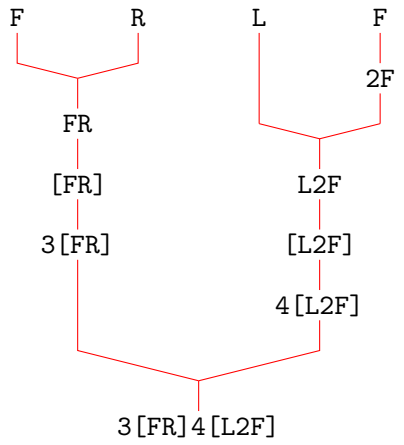


Figura 10.3: Albero sintattico del programma `3[FR]4[L2F]`.

□

10.10 I sottoprogrammi

Nella maggior parte dei linguaggi di programmazione è prevista la possibilità di definire un *sottoprogramma*, ossia una parte di programma, denotata con un identificatore e richiamata in altre parti del programma. I sottoprogrammi permettono di:

- usare dei nomi per identificare un programma
- applicare le metodologie top-down e bottom-up
- aumentare la leggibilità di un programma

Esempio 10.10.1 - Un sottoprogramma può essere identificato mediante un nome identificativo; ad esempio con la scrittura

$$duepassi \stackrel{\text{def}}{=} \texttt{FF}$$

si definisce un programma di nome *duepassi* la cui esecuzione fa avanzare Puntino, nella direzione corrente, di 2 passi. Come avviene per qualsiasi sottoprogramma, in tutti i linguaggi di programmazione, la *definizione* permette solo di specificare l'effetto del sottoprogramma, mentre, affinche le azioni vengano svolte si deve *richiamare* il sottoprogramma scrivendo come istruzione il nome identificativo con il quale è stato definito.

Esempio 10.10.2 - Un sottoprogramma può essere definito in funzione di alcuni argomenti che permettono di specificare, al momento della chiamata del sottoprogramma, alcuni valori che verranno utilizzati all'interno del sottoprogramma per la specifica chiamata; ad esempio, la definizione

$$avanti(n) \stackrel{\text{def}}{=} n * \texttt{F}$$

definisce il sottoprogramma *avanza*, che richiamato, ad esempio, nella forma $avanti(10)$ fa avanzare Puntino di 10 passi nella direzione attuale.

Esempio 10.10.3 - Un programma definito mediante un nome può essere richiamato all'interno della definizione di un altro sottoprogramma; ad esempio, per far fare 2 passi indietro a Quadretto si può definire e successivamente richiamare il seguente sottoprogramma:

$$indietroduepassi \stackrel{\text{def}}{=} [inverti, duepassi, inverti]$$

Esempio 10.10.4 - Un programma definito mediante un nome può essere richiamato all'interno della definizione di un altro sottoprogramma; ad esempio, per far avanzare Puntino di n passi, invertire la posizione e ritornare al punto di partenza si può definire e successivamente richiamare il seguente sottoprogramma:

$$avantindietro(n) \stackrel{\text{def}}{=} [2 * [avanti(n), inverti]]$$

□

Esempio 10.10.5 - Si possono definire i seguenti programmi:

$$inverti \stackrel{\text{def}}{=} \texttt{LL}$$

$$avanti(n) \stackrel{\text{def}}{=} n * \texttt{A}$$

$$origine \stackrel{\text{def}}{=} [sinistra, avanti(\texttt{N}), inverti]$$

□

Osservazione. Dal punto di vista operativo la definizione di un sottoprogramma costituisce un comodo artificio per condensare in una parola una parte articolata di programma; dal punto di vista metodologico risulta lo strumento principale per attuare le metodologie top-down e bottom-up; dal punto di vista dell'interazione fra solutore e esecutore la definizione di un sottoprogramma rappresenta la possibilità offerta al solutore di istruire l'esecutore, aumentando il repertorio delle capacità di base dell'esecutore che possono essere richiamate.

10.11 Semantica del linguaggio

Come ogni linguaggio, il linguaggio $\mathcal{L}$ di Puntino definito nel precedente paragrafo ha bisogno di una interpretazione per risultare efficace. Per questo linguaggio l'interpretazione avviene specificando come devono essere eseguiti i programmi. *Eseguire* un programma significa eseguire in sequenza le singole istruzioni di cui è composto il programma. L'esecuzione di un programma P viene attivata con l'istruzione $exec(P)$. L'esecuzione di un programma genera un *processo*. Un programma P può essere visto come una funzione che allo stato iniziale S_i fa corrispondere lo stato finale S_f; in notazione funzionale:

$$S_f = P(S_i)$$

L'interpretazione del linguaggio avviene precisando come devono essere eseguite le parole (programmi); le regole sono le seguenti:

1) l'interpretazione di un singolo carattere è data dall'esecuzione della corrispondente azione di movimento
2) l'interpretazione di una sequenza di caratteri avviene eseguendo in sequenza ciascun carattere
3) l'interpretazione di un numero davanti ad una parola avviene ripetendo l'esecuzione della parola tante volte pari al numero

Esempio 10.11.1 - Applicando le precedenti regole:

`F`	fa avanzare di 1 passo
`L`	fa ruotare a sinistra di 1 angolo retto
`FRF`	fa fare un angolo a destra
`4F`	fa avanzare di 4 passi
`4[2FR]`	fa percorrere un quadrato di lato uguale a 3 passi

□

La regola 3) permette di *comprimere* un programma avente un motivo che si ripete; risulta simmetricamente definita un'operazione di *espansione* di un programma in uno equivalente.

Esempio 10.11.2 - A seguire è riportato un esempio di compressione ed espansione basati sulla regola 3).

$$\texttt{F2FR3F2R} \to \texttt{3FR3FRR} \to \texttt{2[3FR]R}$$
$$\texttt{2[L3FR]} \to \texttt{2[LFFFR]} \to \texttt{LFFFRLFFFR}$$

□

Ogni problema ha una sua specifica vocazione. Il seguente mira a: verificare la conoscenza della notazione, il concetto di ciclo, la capacità di simulare un procedimento e la capacità di visualizzazione spaziale.

Problema 10.11.1 Determinare lo stato finale dopo aver eseguito il seguente programma a partire dallo stato iniziale $(3, 2, \texttt{E})$:

$$\texttt{4[2F3[RF]]}$$

Soluzione. Simulando l'esecuzione del programma passo-dopo-passo (a mente, con carta e penna, su una scacchiera oppure al computer) si ricava che alla fine si raggiunge lo stato $(3, 2, \texttt{E})$. □

Osservazione. La distinzione fra il concetto di *programma* da quello della sua *esecuzione* permette di:

- evidenziare la differenza fra *programma* e *processo*
- rappresentare un programma mediante una stringa
- fare operazioni sui programmi (concatenazione, semplificazione, ...)
- analizzare la complessità computazionale di un programma
- realizzare algoritmi che generano programmi

10.12 Analisi dei programmi

Analizzare un programma significa valutare sulla carta, senza eseguire il programma, le principali caratteristiche che emergono dalla sua esecuzione. Analizzare un programma sulla carta implica capacità di astrazione, quella tipica che serve ad un progettista di qualsiasi cosa che deve anticipare con l'idea la costruzione ed il successivo utilizzo.

Le caratteristiche che vengono solitamente considerate nell'analisi di un programma sono riportate nei sotto paragrafi che seguono.

Invarianza di un programma

Un programma è *invariante* rispetto allo stato se non modifica lo stato, ossia se lo stato finale raggiunto dopo aver eseguito il programma risulta uguale a quello iniziale.

Esempio 10.12.1 - Il programma 4[2AI] è invariante rispetto allo stato. □

Equivalenza di programmi

Coerentemente con quanto già definito per gli algoritmi, due programmi si dicono *equivalenti* se generano processi uguali; si dicono *funzionalmente equivalenti* se, a partire da stati iniziali uguali, fanno compiere lo stesso percorso e portano a stati finali uguali. Due programmi equivalenti sono anche funzionalmente equivalenti. Esiste una situazione intermedia di equivalenza: due programmi si dicono *equivalenti rispetto al percorso* se portano Quadretto a visitare la stessa *sequenza* di caselle.

Esempio 10.12.2 - I due programmi P_1 e P_2 che seguono sono equivalenti ma non uguali:

$$P_1 \stackrel{\text{def}}{=} \texttt{3F2L}$$
$$P_2 \stackrel{\text{def}}{=} \texttt{FFF+LL}$$

I due programmi Q_1 e Q_2 che seguono sono funzionalmente equivalenti ma non equivalenti:

$$Q_1 \stackrel{\text{def}}{=} \texttt{F3LF}$$
$$Q_2 \stackrel{\text{def}}{=} \texttt{FRF}$$

I due programmi R_1 e R_2 che seguono sono equivalenti rispetto al percorso:

$$R_1 \stackrel{\text{def}}{=} \texttt{F2RF}$$
$$R_2 \stackrel{\text{def}}{=} \texttt{F2LF}$$

□

Complessità di un programma

La *complessità* di un programma è data dal numero di operazioni elementari che vengono eseguite nell'esecuzione del programma. Da questa definizione discende che la complessità è proporzionale al tempo di esecuzione del programma. Dati due programmi equivalenti, per questioni di efficienza, risulta preferibile quello avente la minore complessità.

Esempio 10.12.3 - Il seguente programma ha complessità 12:

```
4[2FR]
```

□

Semplificazione di un programma

Semplificare un programma significa trasformarlo in uno equivalente di complessità minore.

Problema 10.12.1 Semplificare il programma `[[F,F,R]+[L,F,F,R]]`.

Soluzione. : Semplificando:

$$\texttt{[F,F,R]} + \texttt{[L,F,F,R]} \rightarrow \texttt{FFR} + \texttt{LFRF} \rightarrow \texttt{2FRLFRF} \rightarrow \texttt{3FRF}$$

□

Problema 10.12.2 Muovere Quadretto 3 *passi indietro*.

Soluzione. Analizziamo la frase *Fai 3 passi indietro*. Può essere scomposta in due diversi modi, come segue:

1. fai (3 passi indietro)
2. per 3 volte (fai 1 passo indietro)

Queste due diverse scomposizioni possono essere sviluppate rispettivamente come segue:

1. [*inverti*, 3∗*avanti*, *inverti*]
2. per 3 volte [*inverti*, *avanti*, *inverti*]

Queste due scomposizioni portano alle due diverse soluzioni fornite dai seguenti due programmi P_1 e P_2 funzionalmente equivalenti:

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \texttt{[2R,3F,2L]} \\ P_2 &\stackrel{\text{def}}{=} \texttt{3[2R,F,2L]} \end{aligned}$$

Questi due programmi hanno le seguenti complessità:

$$\begin{aligned} compl(P_1) &= 2 + 3 + 2 = 8 \\ compl(P_2) &= 3(2 + 1 + 2) = 15 \end{aligned}$$

Da ciò si conclude che P_1 è più efficiente rispetto a P_2. Il programma P_1 tiene traccia della scomposizione del problema; può essere scritto in forma più compatta con la stringa `2R3F2L`. Il programma P_2 può essere semplificato come segue:

$$\begin{aligned} \texttt{3[2R,F,2L]} &\rightarrow \texttt{2RF2L2RF2L2RF2L} \\ &\rightarrow \texttt{2RFFF2L} \\ &\rightarrow \texttt{2R3F2L} \end{aligned}$$

□

10.13 I processi

L'esecuzione di un programma P fa compiere a Quadretto un percorso λ che dipende, oltre che da P, dallo stato iniziale s_0 di Quadretto. La forma del percorso λ è individuata dal programma P, mentre la localizzazione (posizione ed orientamento) di λ all'interno del reticolo dipende dallo stato iniziale s_0. Per indicare che un programma P porta Quadretto dallo stato inziale s_0 allo

stato finale s_F lungo un percorso λ si utilizza un *grafo di transizione di stato* come quello descritto nella figura 10.4.

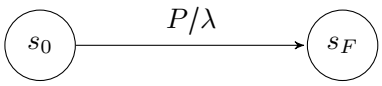


Figura 10.4: Grafo di transizione dallo stato s_0 allo stato s_F eseguendo il programma P.

Un grafo di transizione di stato come quello riportato nella figura 10.4 può essere descritto in modo testuale mediante la notazione delle asserzioni di Hoare della forma

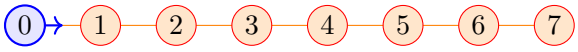
$$\{s_0\}\, P/\lambda\, \{s_F\}$$

che può essere letta come segue: a partire dallo stato s_0, eseguendo il programma P, si percorre il cammino λ e si raggiunge lo stato s_F. Da questa impostazione si delineano diverse classi di problemi in cui date 3 delle 4 variabili s_0, P, λ, s_F si deve determinare il valore della variabile incognita. Una importante classe di problemi corrisponde al caso in cui sono dati i valori di s_0, λ e s_F e si deve determinare il programma P in modo da soddisfare l'asserzione $\{s_0\}\, P/\lambda\, \{s_F\}$. In molti casi il vincolo di percorrere un prefissato cammino λ non viene imposto (e si accettano tutti i possibili percorsi). Un'altra importante classe di problemi corrisponde al caso in cui sono dati lo stato iniziale s_0, il programma P e si deve determinare lo stato finale s_f ed il percorso λ.

Un programma P che assolve ad un compito è *corretto* se nelle ipotesi iniziali A assunte garantisce una data post-condizione B; questo fatto viene indicato con il formalismo delle asserzioni di Hoare: $\{A\}P\{B\}$. In un algoritmo le condizioni di Hoare vengono solitamente espresse mediante le clausole *Require* (condizione iniziale richiesta) e *Ensure* (condizione finale raggiunta).

10.14 Limitazione dello spazio di movimento di un robot

Interessanti situazioni si verificano quando ad un robot si impongono delle limitazioni, in particolare al suo spazio di movimento. Supponiamo di limitare sulla destra le posizioni di movimento di Puntino; ad esempio ipotizziamo che l'ultima posizione sia identificata dal numero 7.



Una modifica di questo tipo impone che venga precisato come il robot si comporta ai confini dello spazio, nel nostro caso come Puntino si comporta quando un'azione di movimento lo porterebbe fuori dalle posizioni ammesse. Si possono, ad esempio, adottare uno delle seguenti modalità di comportamento:

1. si ferma
2. inverte il senso di marcia e rimbalza
3. rientra dall'altra parte, come se le posizioni fossero in circolo chiuso

10.15 Generalizzazioni dei robot

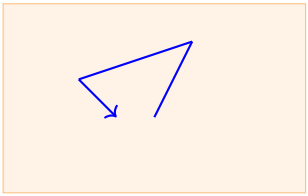
Quanto visto nelle procedenti pagine costituisce una elementare introduzione alle problematiche della robotica; si tratta inoltre di una semplicistica interpretazione del concetto di robot. Tutto ciò si presta, comunque, a delle interessanti generalizzazioni, avvicinandosi a situazioni più articolate, più interessanti e più realistiche.

Si possono intraprendere diverse linee di generalizzazione:

- passaggio ad uno spazio di *dimensione superiore*
- passaggio dal *discreto* al *continuo*
- passaggio dal robot *monoblocco* ad uno con struttura *composita*
- passaggio da un robot *virtuale* ad uno *fisico*

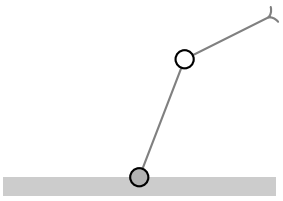
Esempio 10.15.1 - Il passaggio ad una dimensione superiore riguarda lo spazio; nei paragrafi precedenti si è visto un semplice esempio di passaggio ad una dimensione superiore, passando dallo spazio unidimensionale (retta) di Puntino allo spazio bidimensionale (piano) di Quadretto. □

Esempio 10.15.2 - Passando dallo spazio discreto in cui si muove Quadretto ad uno spazio continuo si arriva ad un robot, tradizionalmente noto come *tartaruga*, che si muove nel piano euclideo; similmente a Quadretto può ruotare ed avanzare. Tartaruga è inoltre caratterizzata da una penna che lascia traccia del movimento, in modo tale da poter disegnare figure.



□

Esempio 10.15.3 - Un semplice esempio di robot composito è descritto nella figura che segue: si tratta di un robot, ancorato in modo fisso ad una base, è costituito da un braccio con una giuntura ed ha una pinza prensile all'estremità.



□

ESERCIZI

10.1 Definire le operazioni di un automa-robot che sta fermo sul posto ed è in grado di eseguire delle operazioni di rotazione a sinistra/destra di un angolo retto.

10.2 Con riferimento all'esempio 10.1.1 scrivere delle espressioni/comandi corrispondenti alle funzioni/comandi `INC`, `DEC`, `VAL`.

10.3 Stabilire la successione di valori generati da Puntino eseguendo in sequenza i seguenti due programmi, a partire dalla situazione iniziale $(1, \texttt{RIGHT})$ (`N` denota il numero corrispondente alla posizione corrente):

1. `NII`
2. `NI2I`

10.4 Stabilire dove viene a trovarsi Puntino dopo l'esecuzione delle seguenti istruzioni:

$$3[3\texttt{A}, 2\texttt{A}, \texttt{I}, 4\texttt{A}]$$

10.5 Dimostrare che

$$n\texttt{I} \equiv (n \bmod 2)\texttt{I}$$

10.6 Orientare Puntino verso destra, qualsiasi sia il verso corrente.

10.7 Scrivere dei programmi per:

1. incrementare di 1 la posizione corrente
2. diminuire di 1 la posizione corrente
3. `NI2I`

10.8 Sia P un programma. Stabilire se vale la seguente uguaglianza:

$$m * (n * P) \equiv (m * n) * P$$

Specificare il significato del simbolo asterisco che compare nella precedente scrittura.

10.9 Descrivere mediante degli alberi sintattici la generazione dei seguenti programmi:

1. `FLFRRFFL`
2. `[FF][L[FR]]`
3. `3F4[FR]2L`

10.10 Stabilire l'effetto dei due programmi `3I3A` e `3[IA]`; in particolare stabilire se sono equivalenti.

10.11 Descrivere l'effetto e valutare la complessità dei seguenti programmi:

1. `3FRF2L`
2. `2[3FR3[FL]2L]`

10.12 Semplificare e ridurre il programma `[[F,R,F]+[R,F]]`.

10.13 Stabilire se il programma `FR` è equivalente al programma `RF`.

10.14 Esprimere $inv(\alpha)$ per un generico programma α. Suggerimento: scrivere un algoritmo ricorsivo.

10.15 Definire delle possibili azioni elementari per movimentare il braccio robotico descritto nell'esempio 10.15.3. Definire la condizione affinchè un dato punto sia raggiungibile dall'estremità del braccio. Precisare in cosa consiste lo stato attuale del robot ed in base ad esso muovere il braccio in modo che la sua estremità di presa raggiunga una data posizione.

11

MUOVERE

Ogni lungo cammino inizia con un primo passo.
Lao Tzu *(filosofo cinese del VI secolo a.C.)*

Il camminare è una delle capacità sviluppate per prime dai bambini, in quanto si svolge in modo spontaneo, senza richiedere la maturazione di avanzate esperienze né l'acquisizione di concetti astratti. Il camminare non ha bisogno di numeri e tantomeno di calcoli: richiede solo di essere in grado di fare un passo alla volta, di cambiare direzione e di avere una strategia di avanzamento. Nei bambini la *volontà* di camminare, il *procedimento* del camminare e l'*azione* del camminare sono assommate ed indistinte. Nell'ambito della programmazione, e più in particolare nel campo della robotica, queste tre componenti vengono distinte ed analizzate singolarmente: la volontà di camminare è data dall'esigenza di risolvere un dato problema che richiede un movimento, il procedimento di come camminare viene descritto mediante un algoritmo ideato dal solutore e l'azione del camminare viene delegata ad un'entità separata costituita da un robot o altro generico esecutore che si muove sotto i diretti comandi impartitigli dall'esterno oppure in base ad un procedimento interno precedentemente programmato nel robot; in entrambi i casi la responsabilità di come muoversi è tutta esterna al robot. In questa contestualizzazione l'azione propria del camminare viene trasferita ad un'entità esterna che viene comandato; mettendosi nell'ottica del solutore l'azione di camminare viene così più correttamente denominata *muovere*.

11.1 Muoversi nel piano

Il movimento è una relazione fra due entità. Generalmente queste due entità hanno un ruolo asimmetrico: una sta ferma e viene qualificata come *spazio*, l'altra ha un ruolo attivo e *si muove* nello spazio.

Nel seguito considereremo una situazione elementare in cui il movimento avviene nel piano. Per analizzare il rapporto fra spazio e movente è necessario definire un sistema di riferimento; nel caso di un piano è sufficiente, ad esempio, definire un sistema di assi cartesiani. Con queste premesse il rapporto fra le due entità spazio e movente è sintetizzato dal concetto di *stato* che, in una situazione semplificata, unisce l'informazione relativa alla posizione ed alla direzione di avanzamento corrente del movente rispetto allo spazio. Un'entità che si muove autonomamente in uno spazio in base ad un programma viene spesso denotata con il termine *robot*, indipendentemente dalla sua costituzione fisica.

Un'utile semplificazione del movimento consiste nella sua scomposizione in *sequenza di passi elementari*; è quello che abbiamo fatto da piccoli quando abbiamo imparato a camminare, facendo un passo dopo l'altro. Questa scomposizione passo-dopo-passo del movimento corrisponde ad una discretizzazione del piano in cui ci si muove e nel tempo. In questa trattazione semplificata non si considereranno parametri cinematici quali velocità ed accelerazioni ed il tempo verrà ridotto al concetto di sequenza temporale, in pratica al solo concetto di prima-dopo e, spesso, al solo inizio-fine. Estrapolando il movimento dalla sua contestualizzazione temporale vengono persi alcuni suoi attributi cinematici quali la velocità e l'accelerazione; con questa semplificazione il movimento si geometrizza e si riduce ad una linea. Spingendo ancora oltre questa semplificazione si può discretizzare lo spazio e ancor di più limitare lo spazio ad una zona circoscritta. Sarà questo il contesto spaziale che verrà utilizzato nel seguito di questo capitolo.

Il movimento ha come suoi atomi componenti l'avanzamento e la rotazione. In un contesto di spazio discretizzato il movimento può essere ottenuto mediante un avanzamento unitario e la rotazione a sinistra o a destra di un angolo retto. Nella sua forma più elementare il muovimento può essere svolto combinando pochissime operazioni elementari: avanzamento di un passo e rotazione, unitamente alla possibilità di organizzare queste operazioni in sequenza e mediante la ripetizione.

11.2 La programmazione di Quadretto

Pur nella sua semplicità, la movimentazione di Quadretto riflette ed evidenzia una duplice modalità di impiego di un artefatto da parte dell'uomo; si possono evidenziare (almeno) i seguenti due paradigmi:

- paradigma *strumento*: similmente a come utilizziamo una penna per scrivere o una bicicletta per spostarci, l'uomo aziona direttamente lo strumento in base alla propria volontà e scelta: lo strumento diventa una protesi del corpo dell'uomo e viene controllato dal cervello dell'uomo. In questa modalità Quadretto viene mosso utilizzando i tradizionali tasti freccette

presenti sulla tastiera, con i seguenti effetti:

▲ : avanzamento di un passo
▼ : indietreggiamento di un passo
◀ : rotazione a sinistra di un angolo retto
▶ : rotazione a destra di un angolo retto

Questa modalità permette all'utente, usando un programma di simulazione, di immedesimarsi in Quadretto, di sperimentare ed esercitarsi sul concetto di destra/sinistra e di rinforzare il senso di orientamento nel piano.

- paradigma *solutore-esecutore*: l'uomo istruisce l'esecutore sul come deve agire e comportarsi; una volta istruito e attivato l'esecutore agisce autinomamente e l'uomo perde qualsiasi controllo sull'esecutore. La programmazione deve precedere la gestione di situazioni non note a priori ed addirittura. È questa la situazione più interessante dal punto di vista algoritmico e per le applicazioni dei robot reali.

In talune situazioni serve considerare *il movimento nullo*, denotato con θ, che corrisponde a "nessun movimento".

Osservazione. Il fatto che Quadretto si sposti in modo regolare e preciso nelle caselle semplifica la sua gestione, permettendo di concentrarsi sulla logica del movimento, filtrando tutti quegli aspetti pratici, approssimativi ed imponderabili che si incontrano nella gestione di un robot reale.

Per muovere Quadretto serve impartirgli dei comandi e per comunicare con esso serve un linguaggio. In un contesto di programmazione, come quello relativo a Quadretto descritto nel precedente paragrafo, una parola del linguaggio viene generalmente detta *programma*.

I robot eseguono le azioni una alla volta, in sequenza. Ci si potrebbe quindi attendere che un programma debba essere composto da una sequenza di comandi e, quindi, da una stringa. Fortunatamente, i linguaggi offrono dei meccanismi descrittivi che consentono al solutore di accorciare i programmi consensandoli in stringhe molto più corte che vengono poi espanse al momento dell'esecuzione.

Esempio 11.2.1 - La stringa `FFRFFRFFRFFR` permette di muovere Quadretto lungo un quadrato di 3 unità di lato, ritornando al punto di partenza. Questa stringa può essere condensata nella stringa `4[2FR]` che genera lo stesso processo della stringa indicata all'inizio. Questo esempio lascia intuire alcuni meccanismi per la compressione dei programmi. □

Esempio 11.2.2 - Un programma può essere definito in funzione di alcuni argomenti. Per avanzare di un numero n di passi si può definire

$$avanti(n) \stackrel{\text{def}}{=} n * \texttt{F}$$

Per indietreggiare di n passi, applicando ciecamente la metodologia bottom-up, si può ricorrere al seguente programma:

$$indietro(n) \stackrel{\text{def}}{=} n * indietro$$

È evidente che un tale algoritmo risulta inefficiente in quanto obbliga Quadretto ad eseguire delle inutili coppie di comandi *inverti*. Lo stesso effetto può essere ottenuto più efficientemente mediante il seguente programma:

$$indietro(n) \stackrel{\text{def}}{=} [inverti, n * \texttt{F}, inverti]$$

□

11.3 Problemi di movimento

A seguire sono presentati diversi problemi di movimento relativi a Quadretto.

Problema 11.3.1 Percorrere in senso orario lungo il bordo di una scacchiera 8×8, partendo dalla prima casella in basso a sinistra e con la direzione iniziale verso nord.

Soluzione. Il percorso può essere generato mediante la ripetizione di comandi come descritto nel programma che segue:

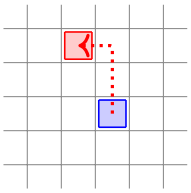
```
4[7FR]
```

Questa soluzione può essere generalizzata a percorrere un quadrato di n unità di lato, definendo il seguente sottoprogramma:

$$quadrato(n) \stackrel{\text{def}}{=} 4 * [n * \texttt{F}, \texttt{R}]$$

□

Problema 11.3.2 Muovere Quadretto a mossa di cavallo degli scacchi, come descritto dalla figura che segue.

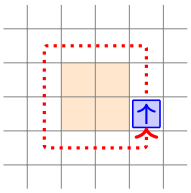


Soluzione. La soluzione è fornita dalla definizione del seguente sottoprogramma:

$$cavallo \stackrel{\text{def}}{=} \texttt{2FLF}$$

□

Problema 11.3.3 Muovere Quadretto lungo un percorso di lato pari a 4 unità, attorno ad un quadrato di lato 2 unità, come descritto nella figura che segue.

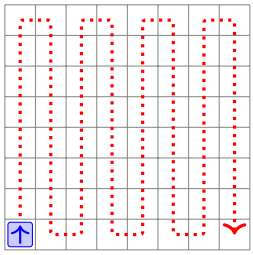


Soluzione. Richiamando il sottoprogramma *cavallo* definito nel precedente esempio, il percorso a quadrato può essere definito come segue:

$$quadrato \stackrel{\text{def}}{=} 4 * cavallo$$

È un semplice esempio di applicazione della metodologia bottom up. Si noti che nella parte destra della definizione di *cavallo* c'è una stringa mentre nella definizione di *quadrato* il 4 viene scritto come numero. □

Problema 11.3.4 Muovere Quadretto lungo un percorso a serpentina, a partire dallo stato iniziale $(1, 1, \texttt{NORTH})$, fino a raggiungere lo stato finale $(8, 1, \texttt{SOUTH})$.



Soluzione. Analizzando il percorso si può distinguere che esso è costituito da 3 componenti a forma di ⎍ connessi tra loro da 2 tratti unitari orizzontali _ che formano una curva a sinistra; ciascuna componente a forma di ⎍ è composta da 2 tratti rettilinei di 7 unità di lunghezza, connessi con un tratto unitario orizzontale _ che forma una curva a destra. Queste considerazioni possono essere scritte in modo simbolico mediante una sequenza di assegnazioni che costituiscono un algoritmo che genera il percorso richiesto:

```
1: t ← 7F                ▷ tratto
2: d ← RFR               ▷ curva a destra
3: s ← LFL               ▷ curva a sinistra
4: u ← [t, d, t]         ▷ percorso a ⊓
5: p ← [u, 3[s, u]]      ▷ percorso completo
6: return p
```

□

11.4 Muoversi nell'ambiente

Nei precedenti paragrafi Quadretto è stato movimentato definendo dapprima la lista p delle operazioni da eseguire e poi facendola eseguire mediante il comando $exec(p)$. Una modalità alternativa e più duttile consiste nell'organizzare mediante dei controlli la struttura dell'esecuzione all'interno della quale collocare le singole azioni elementari.

L'esecuzione di una singola azione di movimento viene indicata mediante il comando

$$muovi(m)$$

dove l'argomento m indica uno dei quattro possibili movimenti:

- *avanti* : avanzamento di 1 passo
- *indietro* : indietreggiamento di 1 passo
- *sinistra* : rotazione a sinistra di 90 gradi
- *destra* : rotazione a destra 90 gradi un avanzamento di 1 passo

Osservazione. Per semplicità di scrittura, nella presentazione e negli algoritmi che seguono al posto di $muovi(m)$ scriveremo semplicemente m; ad esempio *avanti* indicherà il comando $muovi(avanti)$ e questo corrisponde al comando $exec(\texttt{F})$.

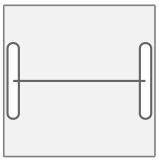
Esempio 11.4.1 - Il seguente programma muove Quadretto lungo un percorso quadrato di lato pari a 3 unità (caselle):

```
1: for 4 times
2:    for 2 times
3:       avanti
4:    end for
5:    sinistra
6: end for
```

Questo programma corrisponde a $exec(4\texttt{[2FL]})$. □

11.5 Movimento mediante i motori

Il movimento di un robot può essere gestito indirettamente agendo sui motori che determinano lo spostamento. Una situazione molto semplificata, ma comunque indicativa di molti robot fisici, è costituita da un robot dotato di due ruote poste ai lati, sull'asse orizzontale; nel caso di Quadretto lo schema è descritto nella figura che segue.



Le ruote vengono azionate separatamente mediante due motori indipendenti; l'azionamento indipendente dei due motori permette l'avanzamento, l'indietreggiamento, le curve e la rotazione sul posto. I motori possono far girare le ruote in senso orario (producendo un avanzamento del robot) oppure in senso antiorario (producendo un indietreggiamento del robot). La seguente tabella descrive le diverse situazioni.

Motore sinistro	Motore destro	Effetto
`AVANTI`	`AVANTI`	avanzamento rettilineo
`AVANTI`	`FERMO`	avanzamento e gira a destra
`AVANTI`	`INDIETRO`	rotazione a destra
`FERMO`	`AVANTI`	avanzamento e gira a sinistra
`FERMO`	`FERMO`	nessun movimento
`FERMO`	`INDIETRO`	indietreggiamento e giro a destra
`INDIETRO`	`AVANTI`	rotazione a sinistra
`INDIETRO`	`FERMO`	indietreggiamento e giro a sinistra
`INDIETRO`	`INDIETRO`	indietreggiamento

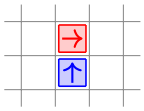
Figura 11.1: Tabella delle possibili combinazioni di movimentazione dei motori delle 2 ruote.

Il comando per azionare i motori è il seguente:

motore(versoMotoreSinistro, versoMotoreDestro, numeroCicli)

L'argomento *numeroCicli* denota le unità di ripetizioni del movimento specificato.

Con il comando *motore* si riesce a realizzare le movimentazioni esaminate al paragrafo precedente (*avanti*, *indietro*, *sinistra*, *destra*); in più si riesce a realizzare delle altre forme di movimento; ad esempio, la combinazione *motore*(`AVANTI`, `FERMO`, 1) corrisponde ad un avanzamento con rotazione a destra, come descritto nella figura che segue.



Osservazione. Nei robot reali si prendono in considerazione ulteriori caratteristiche delle ruote, quali:

- numero delle ruote
- raggio delle ruote
- distanza fra le ruote
- velocità di rotazione dei motore

11.6 Muoversi conoscendo lo stato

Una primitiva forma di interazione con l'ambiente circostante consiste nell'avere consapevolezza della propria collocazione rispetto all'ambiente stesso. Ciò è reso possibile dalle seguenti tipologie di sensori:

1. sensore di *posizione*: analogo ad un GPS, informa il robot della sua posizione rispetto ad un prefissato sistema di riferimento
2. sensore di *orientamento*: analogo ad una bussola, fornisce al robot la sua direzione di avanzamento rispetto ad una direzione di riferimento

In ogni istante la situazione di Quadretto è completamente descritta dal suo *stato*, costituito dalla *posizione* in cui si trova e dal *verso* (Nord, Est, Sud, West) di avanzamento. Lo stato risulta dunque descrivibile mediante una terna della forma (*riga*, *colonna*, *verso*) formata dalle coordinate di posizione e dal verso di avanzamento. Ad esempio $(3, 4, N)$ denota lo stato di Quadretto quando è posto nella posizione descritta nella figura 10.2. Chiameremo *stato iniziale* lo stato $(1, 1, N)$, ossia lo stato di Quadretto posto nell'angolo in basso a sinistra e rivolto verso Nord.

Le coordinate della posizione attuale ed il verso attuale di avanzamento sono forniti dalle seguenti funzioni senza argomenti:

posx	:	coordinata x attuale
posy	:	coordinata y attuale
verso	:	verso attuale

Problema 11.6.1 Stabilire se Quadretto è posizionato sul bordo del reticolo.

Soluzione. Ricorrendo ai sensori di stato *posx* e *posy* la condizione di essere sul bordo è espressa dalla seguente espressione:

$$(posx = 1) \vee (posx = 8) \vee (posy = 1) \vee (posy = 8)$$

□

Problema 11.6.2 Orientare Quadretto in un dato verso.

Soluzione. Il problema è risolto dall'algoritmo 1.

Algoritmo 1 - orienta verso v - *orienta*(v)

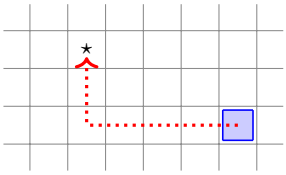
Input: verso v a cui orientarsi
Output: Quadretto è orientato nel verso v

```
1: while verso ≠ v
2:     sinistra
3: end while
```

□

Problema 11.6.3 Raggiungere una data posizione di coordinate (x, y).

Soluzione. La soluzione più semplice consiste nell'eseguire un movimento composto da due movimenti consecutivi lungo gli assi. Ciascuno di questi due movimenti dovrà essere preceduto da un corretto orientamento di verso. Una soluzione è fornita dall'algoritmo 3.



Algoritmo 2 - raggiungi la posizione (x, y) - *raggiungi*(x, y)

Input: coordinate x, y della posizione da raggiungere
Output: Quadretto si trova sulla posizione (x, y)

1: *orienta*(*if*($posx < x$, `EAST`, `WEST`))
2: *avanti*($|posx - x|$)
3: *orienta*(*if*($posy < y$, `NORTH`, `SOUTH`))
4: *avanti*($|posy - y|$)

□

Problema 11.6.4 Far raggiungere a Quadretto l'angolo più vicino. Si assuma l'ipotesi che il percorso sia sgombro da ostacoli.

Soluzione. Osserviamo che esiste un solo angolo *più vicino* alla posizione di Quadretto. Infatti Quadretto si trova in uno dei 4 quadranti ... . L'algoritmo si basa sull'esecuzione di due passi consecutivi:

1. individua l'angolo più vicino $[P_1]$
2. raggiungi l'angolo individuato $[P_2]$

I due sottoproblemi P_1 e P_2 sono risolti alle linee 1-2 e 3 nell'algoritmo che segue.

Algoritmo 3 - raggiungi l'angolo più vicino - *raggiungi_angolo*

Output: Quadretto ha raggiunto l'angolo più vicino

1: $x \leftarrow$ *if*($posx < 5, 1, 8$)
2: $y \leftarrow$ *if*($posy < 5, 1, 8$)
3: *raggiungi*(x, y)

□

11.7 Movimenti generalizzati

Una spontanea generalizzazione di *Quadretto* consiste nel passare dal piano discreto allo spazio continuo. A questo scopo i comandi vengono generalizzati come segue:

$avanti(d)$	avanza di tratto di lunghezza d
$sinistra(\alpha)$	ruota a sinistra di un angolo α
$destra(\alpha)$	ruota a destra di un angolo α

Esempio 11.7.1 - Il percorso lungo un esagono di lato 10 unità può essere generato mediante il seguente ciclo di comandi:

```
1: repeat 6
2:     avanti(10)
3:     sinistra(60)
4: end repeat
```

□

Per avvicinarsi a delle situazioni più realistiche e più vicine ai robot reali è necessario considerare dei parametri cinematici quali il *tempo* e la *velocità*. Per gestire questi parametri si può ricorrere alla seguente coppia di comandi elementari:

$velocita(v)$	imposta a v la velocita' di avanzamento del robot
$attendi(t)$	attende t unita di tempo; in questo intervallo di tempo il robot avanza con la velocita' impostata

Questi due comandi possono essere accorpati e sostituiti con il seguente unico comando che ha l'effetto di avanzare il robot con una velocità v per un tempo t.

$$avanti(v, t)$$

Per avanzare di un tratto s alla velocità v, ricorrendo all'usuale formula del moto rettilineo uniforme $s = v\,t$ si può usare il seguente algoritmo:

Algoritmo 4 - `avanti(`s, v`)`

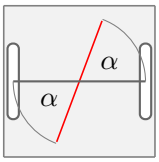
```
1: velocita(v)
2: attendi(s/v)
```

11.8 Un carretto con 2 ruote

Consideriamo una versione più dettagliata del robot *Quadretto* visto precedentemente. Supponiamo che sia composto da un asse trasversale di lunghezza d alle cui estremità sono attaccate due ruote di raggio r. Il movimento del robot viene generato da due motori indipendenti che fanno ruotare i due semiassi a cui sono attaccate le ruote. Indicando con V_L e V_R le velocità angolari di rotazione del motore sinistro e destro (rispetto alla direzione di avanzamento del robot), il tipo di movimento indotto sul robot è schematizzabile mediante la seguente tabella.

relazione fra V_L e V_R	tipo di movimento
$V_L = V_R$	il robot si muove in avanti
$V_L < V_R$	il robot gira a sinistra
$V_L > V_R$	il robot gira a destra
$V_L = -V_R$	il robot gira su se stesso

Il problema di ruotare il robot di un angolo α richiede il controllo della velocità dei motori e il tempo per cui questi debbono essere attivi. Per risolvere questo problema consideriamo la figura che segue:



Indicando con d la distanza fra le due ruote, per ottenere una rotazione di un angolo di ampiezza α (a sinistra o a destra) il punto di contatto fra la ruota con il pavimento deve percorrere una traiettoria circolare di lunghezza pari a

$$\frac{\alpha d}{2}$$

Pertanto i motori delle ruote devono ruotare di un angolo pari a

$$\frac{\alpha d}{2r}$$

ESERCIZI

11.1 Determinare un insieme minimale $\mathcal{M}_0$ di movimenti potenzialmente equivalente all'insieme dei movimenti $\mathcal{M} = \{\mathtt{F}, \mathtt{B}, \mathtt{L}, \mathtt{R}\}$. Esprimere ciascun movimento di $\mathcal{M} \setminus \mathcal{M}_0$ mediante combinazione dei movimenti di $\mathcal{M}_0$.

11.2 Quadretto si trova nello stato $(1, 1, N)$. Scrivere un programma per far percorre un percorso quadrato attorno al bordo interno del reticolo 8×8.

11.3 Stabilire la traiettoria di Quadretto che esegue i seguenti programmi:

1. `2[2FR2FL]`
2. `4[2FR]`
3. `2L3[FRFL]`

11.4 Quadretto si trova nello stato $(1, 1, N)$. Stabilire lo stato finale raggiunto al termine dell'esecuzione dei seguenti programmi:

1. `3[FR2FL]`
2. `3[2[FRFL]`
3. `[12[3FR]FL]`

11.5 Stabilire l'effetto dei seguenti programmi, essendo k un numero naturale.

1. k*[`FRFL`]
2. k*[`2[FR]2[FL]]`

11.6 Scrivere un programma per passare dallo stato $(1, 1, S)$ allo stato $(7, 5, S)$.

11.7 Descrivere l'effetto del seguente programma P rivolto a Quadretto.

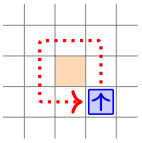
1: $a \leftarrow$ `2FRF`
2: $b \leftarrow$ `2L`
3: $p \leftarrow$ `2[`a, b, a`]`

11.8 Usando il programma *Cavallo* definito nell'esempio 11.3.3 portare Quadretto dallo stato $(1, 1, N)$ allo stato $(8, 8, N)$.

11.9 Traslare Quadretto di un passo verso destra (o sinistra), mantenendo direzione e verso di avanzamento, come indicato nella figura che segue. Traslare Quadretto di un passo in diagonale. Generalizzare al caso di uno spostamento di n passi.

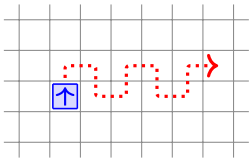
11.10 Determinare il programma di minima complessità che muove Quadretto dallo stato iniziale $(1, 1, N)$ allo stato $(8, 8, N)$

11.11 Muovere Quadretto lungo un percorso quadrato attorno ad una casella, come descritto dalla figura che segue.

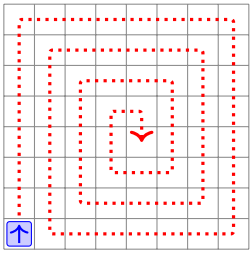


11.12 Muovere Quadretto dallo stato iniziale $(1, 1, \texttt{NORTH})$ allo stato finale $(8, 8, \texttt{NORTH})$ in modo che passi su tutte le caselle della diagonale principale $(1, 1) - (8, 8)$.

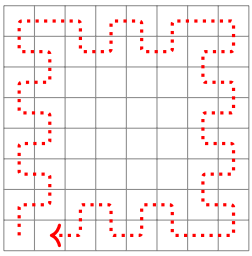
11.13 Muovere Quadretto su una traiettoria ad onda quadra, come descritto nella seguente figura:



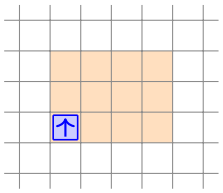
11.14 Muovere Quadretto lungo un percorso a spirale verso l'interno, a partire dallo stato iniziale $(1, 1, \texttt{NORTH})$.



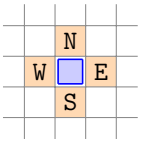
11.15 Muovere Quadretto, a partire dallo stato iniziale $(1, 1, N)$ su una traiettoria ad onda quadra, lungo il bordo come descritto dalla figura che segue.



11.16 Individuare una strategia e descrivere l'algoritmo per muovere *Quadretto* su tutte le caselle di una zona rettangolare di piano di dimensioni $m \times n$, partendo da un angolo, come illustrato nella figura che segue.



11.17 Analizzare e discutere la gestione di Quadretto nel caso in cui non sia definita alcuna direzione di avanzamento e sia ammesso, in alternativa all'insieme di movimenti elementari $\mathcal{M} = \{\texttt{F}, \texttt{B}, \texttt{L}, \texttt{R}\}$, l'insieme di movimenti elementari $\mathcal{M}' = \{\texttt{N}, \texttt{E}, \texttt{S}, \texttt{W}\}$, dove ciascun movimento ha l'effetto di traslare Quadretto alla casella contigua posta rispettivamente a nord, est, sud, ovest rispetto alla casella occupata da Quadretto, come descritto dalla figura che segue. Discutere come, in questo caso, dovrebbero essere adattati gli algoritmi presentati in questo capitolo. Stabilire se l'insieme $\mathcal{M}'$ è *completo* (permette di raggiungere tutte le caselle del reticolo) e se è *minimale* (nessun movimento può essere espresso come combinazione degli altri).



12

SENTIRE

È notevole che nella storia della scienza quasi tutti i fenomeni siano stati presto o tardi spiegati in termini di interazione fra parti prese a due a due.

Marvin Minsky, *La società della mente*

Il movimento diventa interessante se l'entità che si muove riesce ad acquisire informazioni dall'ambiente circostante e dalle altre entità presenti nell'ambiente. Questa capacità viene generalmente definita come *sentire* e viene espletata attraverso i *sensi*. Calandosi nel contesto della robotica, la facoltà di sentire viene realizzata mediante i *sensori*. L'uso di sensori permette un cambio di marcia nella programmazione del robot.

12.1 Acquisire informazioni dall'ambiente

La gestione del movimento in un ambiente può essere più articolata e raffinata se il robot è dotato di *sensori* che lo informano su alcune caratteristiche dell'ambiente circostante e gli permettono di rapportarsi ed interagire con esso. Un robot può essere dotato di diverse tipologie di sensori:

1. sensore di *contatto*: il robot sente quando una sua parte (solitamente la parte frontale) è in contatto con un ostacolo
2. sensore di *colore*: il robot percepisce il colore della casella sulla quale si trova
3. sensore di *distanza*: fornisce la distanza dall'ostacolo posto davanti al robot nella direzione di avanzamento; può essere considerato come un sensore di contatto frontale generalizzato
4. sensore di *vista*: consente al robot di ruotare per guardare verso un dato punto

Sensore di contatto

Una delle forme più primitive di sensore è costituita da un *sensore di contatto*. Il controllo di questo sensore avviene mediante un predicato *tocca* che ritorna `TRUE` se Quadretto è a contatto frontalmente con l'ostacolo.

Esempio 12.1.1 - L'algoritmo 1 fa compiere a Quadretto 1 passo, facendo un dietro-front nel caso si trovi inizialmente a contatto con l'ostacolo.

Algoritmo 1 - *passo*

```
1: if tocca then
2:     destra
3:     destra
4: end if
5: avanti
```

□

Sensore di distanza

Quadretto è dotato di un sensore *distanza* che fornisce la distanza, in unità casella, dall'ostacolo o dal bordo posto davanti. Nel caso particolare in cui Quadretto si trovi a contatto frontale con l'ostacolo, il sensore di distanza fornisce il valore 0.

Sensore di colore

Quadretto, mediante il sensore *colore* fornisce il valore del colore della casella sulla quale è posizionato.

In molti problemi le varie tipologie dii sensori vengono usati congiuntamente, come descritto nel problema che segue.

Problema 12.1.1 Portare Quadretto nello stato $(1, 1, \texttt{NORTH})$.

Soluzione. Il problema proposto è una sottoclasse del problema 11.6.3 ed è quindi risolto dall'algoritmo 3. Una soluzione alternativa può essere sviluppata utilizzando i sensori di stato e di contatto come descritto nell'algoritmo 2.

Algoritmo 2 - portati nello stato $(1, 1, \texttt{NORTH})$

Input: Quadretto si trova all'interno del reticolo
Output: Quadretto si trova nello stato $(1, 1, \texttt{NORTH})$

```
1: orienta(SOUTH)
2: for 2 times
3:     while ¬ tocca
4:         avanti
5:     end while
6:     destra
7: end for
```

□

Realizzare un sensore virtuale

Usando il sensore di contatto frontale è possibile realizzare virtualmente anche dei sensori di contatto laterale. Basandosi sul controllo *tocca* è possibile realizzare un analogo controllo *toccaSinistra* che rileva se il robot è a contatto con un ostacolo sul suo fianco sinistro, come descritto nell'algoritmo 3.

Algoritmo 3 - *toccaSinistra*

```
1: sinistra
2: r ← tocca
3: destra
4: return r
```

Analogamente si può realizzare un controllo *toccaDestra*. I tre controlli *tocca*, *toccaSinistra* e *toccaDestra* possono essere inglobati in un unico controllo parametrico $tocca(l)$ dove $l \in \{\texttt{LEFT}, \texttt{FRONT}, \texttt{RIGHT}\}$ indica il lato di contatto da considerare. Basandosi su questi tre controlli si ha a disposizione un repertorio di sensori che permettono di risolvere interessanti problemi.

12.2 Azioni elementari

Nella movimentazione di un robot si delineano spesso delle azioni che vengono composte fra loro per costruire delle movimentazioni più articolate. Alcune di queste sono descritte nei sottoparagrafi che seguono.

Arrivare a contatto con l'ostacolo

Usando il sensore di contatto mediante il controllo *tocca* è possibile far avanzare Quadretto fino ad entrare in contatto con un ostacolo che si trova davanti

lungo la direzione di avanzamento. Il procedimento descritto presuppone che l'ostacolo si trovi lungo la direzione di avanzamento del robot. Nel caso in cui Quadretto sia inizialmente già in contatto con l'ostacolo, non viene svolta alcuna azione. Il procedimento è descritto nell'algoritmo 4.

Algoritmo 4 - *contatta*

Input: l'ostacolo si trova sulla direzione di avanzamento
Output: *tocca*(FRONT)
1: **while** $\neg$ *tocca*(FRONT)
2: *avanti*
3: **end while**

Liberarsi da un ostacolo

Se Quadretto si trova a contatto frontale con l'ostacolo, per intraprendere un'azione di avanzamento è necessario ruotare fino a trovare una direzione di avanzamento libera. Per costeggiare l'ostacolo è necessario ruotare fino a trovare una direzione di avanzamento libera, rimanendo accostato sul fianco sinistro. Il procedimento è descritto nell'algoritmo 5.

Algoritmo 5 - *libera*

Input: *tocca*(FRONT)
Output: $\neg$ *tocca*(FRONT) $\wedge$ *tocca*(LEFT)
1: **while** *tocca*(FRONT)
2: *destra*
3: **end while**

Avanzare affiancato ad un ostacolo

In molte situazioni si presenta l'esigenza di avanzare mantenendosi a contatto su un fianco con un ostacolo. Consideriamo la precondizione di essere a contatto sul fianco sinistro, di avere libero il lato frontale e di voler avanzare di 1 passo in modo che il fianco sinistro di Quadretto si sposti a contatto del successivo tratto di muro. Per poter eseguire in modo ciclico questa azione è necessario, dopo ogni passo, riconquistare la precondizione *tocca*(LEFT) $\wedge$ $\neg$ *tocca*(FRONT). La strategia di movimento è descrivibile come segue: *avanza di* 1 *passo e riconquista la condizione di affiancamento a sinistra*. Le situazioni da analizzare e gestire sono le seguenti due:

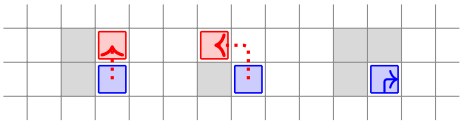


Figura 12.1: Quadretto avanza di 1 passo affiancato ad un ostacolo.

Il procedimento è descritto nell'algoritmo 6.

Algoritmo 6 - *striscia*

Input: *tocca*(LEFT)
Output: *tocca*(LEFT)

```
1: if tocca(FRONT) then
2:     destra
3: else
4:     avanti
5:     if ¬ tocca(LEFT) then
6:         sinistra
7:         avanti
8:     end if
9: end if
```

12.3 Combinazione di azioni elementari

Combinando le azioni elementari *contatta*, *libera* e *striscia* esaminate nel precedente paragrafo si riesce a risolvere interessanti problemi di movimento, come descritto nei seguenti sottoparagrafi.

Affiancare un ostacolo

In molte situazioni di movimento, per poter evitare, superare o circumnavigare un ostacolo si rende necessaria una azione preparatoria che consiste nell'avvicinarsi all'ostacolo e disporsi affiancati per poter avanzare affiancati, utilizzando uno dei due sensori di contatto laterale. Questa azione è descritta nella figura che segue: quadretto si avvicina all'ostacolo e poi ruota per liberare la testa dalla condizione di contatto frontale, in modo da poter avanzare.

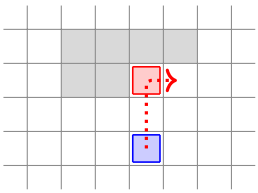


Figura 12.2: Quadretto contatta ed affianca l'ostacolo posto davanti lungo la direzione di avanzamento.

Il procedimento di affiancamento può essere scomposto nelle seguenti due azioni:

```
1: avanza fino ad entrare in contatto con l'ostacolo davanti
2: ruota a destra fino a liberarti dal contatto frontale
```

Si può riconoscere facilmente che queste due azioni corrispondono proprio alle due azioni *contatta* e *libera* precedentemente esaminate; pertanto il procedimento complessivo di affiancamento può essere descritto come riportato nell'algoritmo 7.

Algoritmo 7 - *affianca*

Input: c'è un ostacolo davanti
Output: $tocca(\texttt{LEFT}) \wedge \neg\, tocca(\texttt{FRONT})$

1: *contatta*
2: *libera*

Costeggiare un ostacolo

Il seguente algoritmo permette di costeggiare un ostacolo; viene richiesta solo la precondizione che l'ostacolo si trovi davanti lungo la direzione di avanzamento iniziale. Il tragitto di costeggiamento è descritto nella figura 12.3 dove la linea descrive il tragitto della posizione del baricentro di Quadretto.

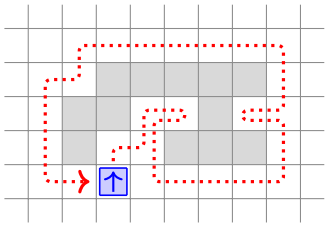


Figura 12.3: Percorso di costeggiamento di un ostacolo.

L'idea fondamentale sulla quale fondare l'algoritmo per costeggiare l'ostacolo è la seguente: *avanza mantenendoti affiancato a sinistra*. Lo schema generale del procedimento è descritto nell'algoritmo 8. Il ciclo infinito comporta che Quadretto costeggia l'ostacolo indefinitamente.

Algoritmo 8 - *costeggia*

1: *contatta*
2: **loop**
3: *striscia*
4: **end loop**

Circumnavigare un ostacolo

Consideriamo la situazione in cui Quadretto si trova a contatto frontale con un muro. Il problema consiste nel far circumnavigare Quadretto attorno al muro, a contatto con il muro sul lato sinistro, fino a tornare al punto di partenza. Si tratta di una variazione del problema di costeggiamento analizzato precedentemente. È sufficiente memorizzare la posizione di partenza e circumnavigare l'ostacolo fino a raggiungere la posizione iniziale. Il procedimento è descritto nell'algoritmo 9.

Algoritmo 9 - *circumnaviga*

Input: l'ostacolo è isolato, Q è a contatto frontale
Output: Q circumnaviga fino a tornare al punto di partenza

```
1: (x_i, y_i) ← (xpos, ypos)      ▷ posizione iniziale
2: libera
3: repeat
4:     striscia
5:     (x_c, y_c) ← (xpos, ypos)      ▷ posizione corrente
6: until (x_i, y_i) = (x_c, y_c)
```

12.4 Raggiungere posizioni obiettivo

Un'ampia classe di problemi rientra nella categoria di procedimenti per gli automi con stato che devono raggiungere uno stato finale obiettivo, come descritto nell'algoritmo 10.

Algoritmo 10 - *raggiungi obiettivo*

```
1: while ¬ raggiunto obiettivo
2:     muoviti per raggiungerlo
3: end while
```

Superare un ostacolo

Consideriamo il robot Quadretto dotato di un sensore di contatto che rileva un ostacolo posto a contatto davanti, lungo la direzione di avanzamento, mediante il controllo *tocca* che ritorna il valore di verità `TRUE` se il robot si trova a contatto frontale con un ostacolo. Con questo elementare sensore istruiremo Quadretto per superare un ostacolo. Il problema è descritto nella figura 12.4.

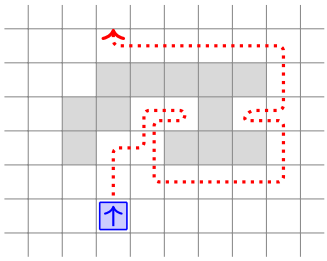


Figura 12.4: Percorso di Quadretto per superare un ostacolo.

Una volta circumnavigata una parte dell'ostacolo, trovandosi dall'altra parte dell'ostacolo, è necessario riprendere il percorso di avanzamento lungo la linea di avanzamento iniziale. Questo condizione si verifica quando Quadretto si trova con la coordinata della linea di avanzamento uguale a quella di partenza e l'altra strettamente maggiore di quella di partenza. L'obiettivo complessivo di superamento dell'ostacolo può essere scomposto in sotto azioni, come descritto nell'algoritmo 11.

Algoritmo 11 - *supera ostacolo* - ver.1

1: avanza fino ad arrivare in contatto con l'ostacolo davanti (*contatta*)
2: *striscia* fino ad arrivare ad una coordinata della linea di avanzamento uguale a quella di partenza e l'altra strettamente più avanzata di quella di partenza (rispetto alla direzione di avanzamento iniziale)
3: ruota a destra

Ad un livello più raffinato l'algoritmo si esprime come descritto nel seguente algoritmo 12.

Algoritmo 12 - *supera ostacolo* - ver.2

1: memorizza lo stato iniziale (x_0, y_0, v_0)
2: superato ← `FALSE`
3: *contatta*
4: **while** ¬ superato
5: *striscia*
6: **if** $\mathcal{C}(x_0, y_0, v_0)$ **then**
7: *destra*
8: superato ← `TRUE`
9: **end if**
10: **end while**

La condizione $\mathcal{C}(x_0, y_0, v_0)$ indicata alla linea 6 dell'algoritmo, indica che Quadretto si trova sulla linea di avanzamento corrispondente allo stato iniziale. Indicando con (x_a, y_a, v_a) lo stato attuale di Quadretto, la condizione $\mathcal{C}$ può essere esplicitata come segue:

$$if(v_0 = \texttt{NORTH} \vee v_0 = \texttt{SOUTH}, x_a = x_0 \wedge y_a \neq y_0, x_a \neq x_0 \wedge y_a = y_0) \wedge (v_a = left(v_0))$$

Chiudere una stanza

Quadretto si trova all'interno di una stanza rettangolare avente entrambe le dimensioni maggiori di 1 ed avente un'apertura unitaria su un lato. Portare Quadretto sull'apertura in modo da chiudere la stanza (figura 12.5).

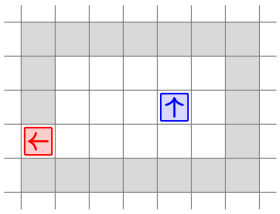


Figura 12.5: Il problema della chiusura della stanza.

Nel problema proposto l'algoritmo generale si esprime come segue:

Algoritmo 13 - *chiudi stanza*

Input: Quadretto si trova nella stanza
Output: Quadretto chiude la stanza

```
1: while ¬ chiusa stanza
2:     muoviti
3: end while
```

La condizione che la stanza sia chiusa si esprime come segue:

$$tocca(\texttt{LEFT}) \wedge tocca(\texttt{RIGHT})$$

Il cuore dell'algoritmo 13 è concentrato nell'azione *muoviti* che viene dettagliata nell'algoritmo 14.

Algoritmo 14 - *muoviti*

```
1: if tocca(FRONT) then
2:     destra
3: else if tocca(LEFT) then
4:     avanza
5:     if ¬ tocca(LEFT) then
6:         sinistra
7:     end if
8: else
9:     avanti
10: end if
```

Innestando l'algoritmo *muoviti* nell'algoritmo *chiudi stanza* si ottiene l'algoritmo complessivo.

Ricercare un oggetto

Consideriamo il problema in cui Quadretto deve esplorare l'ambiente circostante e ricercare un oggetto, all'interno del reticolo, portandosi a contatto con esso. Per risolvere questo problema si possono adottare diverse strategie, sintetizzate come segue:

1. portarsi su un angolo e percorrere un tragitto a serpentina
2. girare a spirale attorno alla posizione di partenza
3. portarsi sul bordo e girare a spirale verso il centro

Di queste diverse strategie la più semplice da realizzarsi è la prima ed è descritta nella figura 12.6.

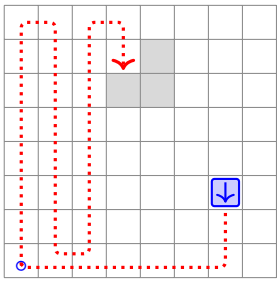


Figura 12.6: Percorso a serpentina alla ricerca di un oggetto.

La struttura generale dell'algoritmo basato su questa strategia può essere espressa come indicato nell'algoritmo 15.

Algoritmo 15 - *ricerca oggetto*

Input: c'è un oggetto nello spazio raggiungibile da Quadretto
Output: Quadretto è a contatto frontale con l'oggetto
1: portati nell'angolo avanti a destra $[P_1]$
2: **while** ¬ tocca ostacolo $[P_2]$
3: avanza di 1 passo su un percorso a serpentina $[P_3]$
4: **end while**

Un primo raffinamento della soluzione dei due sottoproblemi P_1 e P_3 è riportata nei due algoritmi 16 e 17, nei quali emergono i tre sottoproblemi P_4, P_5 e P_6.

Algoritmo 16 - *portati nell'angolo avanti a destra*

1: **for** 2 **times**
2: avanza fino al bordo $[P_4]$
3: *destra*
4: **end for**

Algoritmo 17 - *avanza di* 1 *passo su un percorso a serpentina*

1: *avanza*
2: **if** tocca bordo $[P_5]$ **then**
3: fai una curva di inversione a U $[P_6]$
4: **end if**

A questo livello di dettaglio, nella soluzione dei due sottoproblemi P_4 e P_6, dove viene gestito l'avanzamento di Quadretto, è necessario prendere in considerazione l'evento che si entri in contatto con l'ostacolo ricercato. Globalmente, nella soluzione dei sottoproblemi P_2, P_4, P_5 e P_6 risulta necessario distinguere quando si entra in contatto con il bordo e quando con l'ostacolo. A questo scopo si possono utilizzare i due predicati definiti come segue (con (x, y, v) è indicato lo stato attuale di Quadretto e con `N`, `E`, `S`, `W` sono stati indicati i 4 versi possibili di avanzamento):

$$\begin{aligned} \textit{tocca bordo} &\stackrel{\text{def}}{=} ((x = 1) \wedge (v = \texttt{WEST})) \vee ((x = 8) \wedge (v = \texttt{EAST})) \vee \\ &\quad ((y = 1) \wedge (v = \texttt{SOUTH})) \vee ((y = 8) \wedge (v = \texttt{NORTH})) \\ \textit{tocca ostacolo} &\stackrel{\text{def}}{=} \textit{tocca} \wedge \neg(\textit{tocca bordo}) \end{aligned}$$

Il percorso a serpentina richiede di fare curve di inversione ad U alternativamente a destra ed a sinistra. A questo scopo viene utilizzata una variabile booleana *curva* che ad ogni curva inverte il suo valore. Componendo i diversi spezzoni delle soluzioni dei sotto problemi analizzati e risolti sopra, si giunge alla descrizione completa della soluzione rappresentata dall'algoritmo 18.

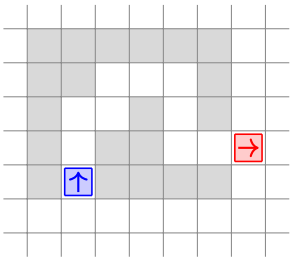
Algoritmo 18 - *ricerca oggetto*

Input: c'è un oggetto nello spazio raggiungibile da Quadretto
Output: Quadretto è a contatto frontale con l'oggetto

```
1:  for 2 times
2:    while ¬(tocca ostacolo) ∧ ¬(tocca bordo) do
3:      avanti
4:    end while
5:    if ¬(tocca ostacolo) then
6:      destra
7:    end if
8:  end for
9:  curva ← TRUE
10: while ¬(tocca ostacolo) do
11:   avanza
12:   if tocca bordo then
13:     for 2 times
14:       if ¬(tocca ostacolo) then
15:         if curva then
16:           destra
17:         else
18:           sinistra
19:         end if
20:       end if
21:       if ¬(tocca ostacolo) then
22:         avanti
23:       end if
24:     end for
25:     curva ← ¬ curva
26:   end if
27: end while
```

Uscire da un tunnel

Quadretto si trova all'interno di un tunnel lineare di larghezza unitaria. Il problema consiste nel farlo uscire dal tunnel.



La condizione di uscita dal tunnel corrisponde ad avere entrambi i lati sinistro e destro liberi da contatto. La strategia di movimento per uscire dal tunnel è

descritta nell'algoritmo 19.

Algoritmo 19 - *uscita dal tunnel*

Input: Quadretto è in un tunnel
Output: Quadretto è uscito dal tunnel ∨ è arrivato in un punto cieco

1: *uscito* ← `FALSE`
2: *bloccato* ← `FALSE`
3: **while** ¬ *uscito* ∧ ¬ *bloccato* **do**
4: a seconda del valore dei sensori di contatto fai un movimento e valuta le condizioni *uscito* e *bloccato*
5: **end while**

Le diverse situazioni che si possono potenzialmente verificare in relazione ai tre possibili contatti frontali e laterali sono 8. Nel caso particolare che Quadretto si trovi all'interno di un tunnel solo il contatto corrispondente al cammino di avanzamento è falso. In base all'unico sensore di contatto avente valore `FALSE` si decide l'azione (una rotazione, a sinistra o destra, oppure un passo in avanti). Tutte le possibili casistiche e la corrispondente azione di movimento da intraprendere sono sintetizzate nel albero di decisione descritto nella figura 12.7 e trova immediata traduzione nell'algoritmo 20.

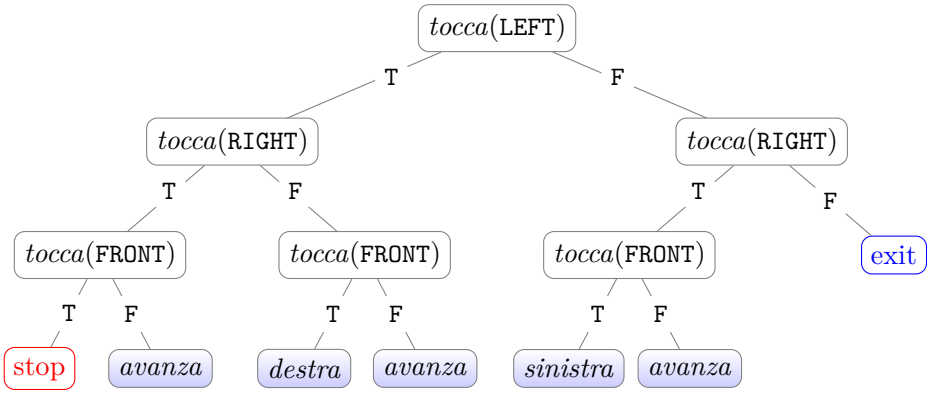


Figura 12.7: Albero di decisione per uscire da un tunnel.

12.5 Uscire da un labirinto

Un *labirinto* è una struttura composta da un intricato insieme di vie; ha un ingresso, una o più uscite (o punti interni), difficili da raggiungere. I labirinti hanno una lunga tradizione storica. Uno dei più famosi è il *labirinto di Cnosso*, risalente all'epoca minoica (dal XVII al XV sec. a.C) e descritto in varie opere mitologiche. La figura 12.8 ne riporta una rappresentazione su un mosaico. Secondo la tradizione il labirinto venne fatto costruire dal re Minosse nell'isola di Creta per rinchiudervi il mostruoso Minotauro. Sempre la tradizione riporta che venne costruito da Dedalo che, a costruzione ultimata, venne fatto

Algoritmo 20 - *esci dal tunnel*

Input: *tocca*(LEFT) $\wedge$ *tocca*(RIGHT)
Output: $\neg$ *tocca*(LEFT) $\wedge \neg$ *tocca*(RIGHT)

```
1:  uscito ← FALSE
2:  bloccato ← FALSE
3:  while ¬ uscito ∧ ¬ bloccato do
4:    if tocca(LEFT) then
5:      if tocca(RIGHT) then
6:        if tocca(FRONT) then
7:          bloccato ← TRUE     ▷ vicolo cieco
8:        else
9:          avanti
10:       end if
11:     else
12:       if tocca(FRONT) then
13:         destra
14:       else
15:         avanti
16:       end if
17:     end if
18:   else
19:     if tocca(RIGHT) then
20:       if tocca(FRONT) then
21:         sinistra
22:       else
23:         avanti
24:       end if
25:     else
26:       uscito ← TRUE     ▷ uscito dal tunnel
27:     end if
28:   end if
29: end while
```

rinchiudere da Minosse, assieme al figlio Icaro, affinchè non potessero rivelare la piantina della costruzione.

Abbandonando la tradizione mitologica, un labirinto è un oggetto matematico che può essere rappresentato mediante un grafo e studiato ed analizzato mediante l'omologa teoria.

Figura 12.8: Minotauro, Teseo e il Labirinto. - Modena, Villa di Via Cadolini, I sec. d. C.

Alcune strategie efficaci per uscire da un labirinto sono note dalla tradizione: l'eroe greco Teseo uscì dal leggendario labirinto di Cnosso usando un gomitolo di filo che aveva srotolato lungo il percorso; Pollicino riuscì ad uscire dal labiritno lasciando delle briciole di pane lungo il percorso all'andata in modo tale da poter tornare sui propri passi. Il problema di come uscire da un labirinto è stato utilizzato anche in vari studi scientifici sul comportamento e l'apprendimento degli animali. Il problema è stato riconsiderato ultimamente in connessione ai robot, tantoché è diventato un classico problema delle gare di robotica.

Il problema di base di un labirinto consiste nel determinare delle strategie che permettano di trovare in modo efficiente la via da percorrere. Se si conosce la pianta del labirinto è possibile adottare un approccio brute-force, percorrendo in modo esaustivo tutti i possibili cammini, trovando quello porta all'uscita. Non conoscendo la pianta bisogna adottare una strategia diversa. Un metodo semplice e sicuro consiste nell'appoggiare la mano destra (o la sinistra) alla parete destra del labirinto (o rispettivamente alla parete sinistra) all'entrata del labirinto, e camminare senza staccare mai la mano dalla parete scelta, fino a raggiungere una delle eventuali altre uscite, o il punto di partenza. La regola della mano (destra o sinistra, equivalentemente) pur non garantendo il percorso più breve, permette di individuare con certezza un'uscita dal labirinto, nell'ipotesi che essa esista e che il percorso che compone il labirinto sia semplicemente connesso. La garanzia di successo di questo procedimento è fornita da alcuni risultati teorici sugli spazi topologici semplicemente connessi caratterizzati dal fatto che per ogni punto si può definire una curva chiusa passante per il punto ed è possibile deformare con continuità la curva all'interno dello spazio, fino a renderla il punto stesso. È possibile dimostrare che ogni spazio semplicemente connesso può essere deformato con continuità fino a diventare un cerchio: il labirinto è quindi topologicamente equivalente a una stanza circolare con una o più porte. È quindi evidente che seguendo la parete interna della stanza si giungerà inevitabilmente a una porta.

Alla fine del XIX secolo i matematici francesi G. Tarry e M. Trémaux idearono un algoritmo in grado di risolvere qualsiasi labirinto. L'algoritmo è descritto a seguire in due equivalenti formulazioni, note anche come *regola della mano destra*.

Algoritmo 21 - metodo di Tremaux

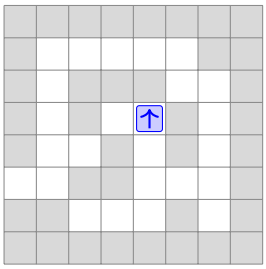
1: Quando ti trovi a un nodo prendi il ramo più a destra. Se arrivi a un vicolo cieco, ritorna sui tuoi passi fino all'ultimo nodo e prendi il ramo più a destra tra quelli ancora inesplorati.

Algoritmo 22 - metodo della mano destra

1: toccare con la mano destra il muro alla propria destra per tutto il labirinto, senza mai saltare un ramo situato alla propria destra.

Nei due algoritmi 21 e 22 si può equivalentemente usare la mano sinistra al posto della mano destra, ottenendo lo stesso risultato.

Adattiamo ora l'algoritmo 22 al caso in cui, partendo da una posizione interna al reticolo, deve uscire dal labirinto avanzando fino a raggiungere una casella sul bordo. Un esempio di situazione è descritta nella figura che segue.



L'algoritmo si scrive:

Algoritmo 23 - *esci dal labirinto*

Input: (Quadretto tocca il labirinto) $\wedge$ (il labirinto è connesso)
Output: Quadretto è a contatto frontale con l'oggetto
1: **while** $\neg$ (*sul bordo*)
2: *striscia*
3: **end while**

Il predicato *sul bordo* può essere definito come segue, dove con (x, y) si denotano le coordinate correnti di Quadretto:

$$\textit{sul bordo} \stackrel{\text{def}}{=} (x = 1) \vee (x = 8) \vee (y = 1) \vee (y = 8)$$

La condizione che il labirinto sia connesso è essenziale: se si togliesse, ad esempio, la casella di coordinate $(3, 3)$ Quadretto continuerebbe a girare attorno allo spezzone centrale di labirinto, senza mai trovare la strada di uscita.

Osservazione. Nei robot reali si devono gestire ulteriori caratteristiche dei sensori:

- numero dei sensori (1, 2, array)
- distanza fra i sensori (se più di 1)
- posizionamento dei sensori (davanti, laterali, ...)
- sensibilità dei sensori di colore (bianco/nero, scala di grigi, colori)
- cono di rilevamento dei sensori di distanza (angolo ed apotema)

ESERCIZI

12.1 Muovere Quadretto dalla posizione corrente alla posizione $P = (x_1, y_1)$ passando per un fissato punto $Q = (x_2, y_2)$.

12.2 Muovere Quadretto alle posizioni $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ per il percoso più breve (è indifferente l'ordine con il quale si raggiungono le due posizioni).

12.3 Scrivere un programma per ottenere la transizione dallo stato attuale ad un determinato stato (x, y, v).

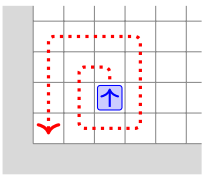
12.4 Analizzare tutte le possibili situazioni in cui si può trovare Quadretto nell'algoritmo 1 e verificare che in ogni caso il robot riesce a togliersi dall'ostacolo.

12.5 Realizzare un sensore virtuale *distanza* che ritorna la distanza dall'ostacolo posto davanti a Quadretto.

12.6 Realizzare un sensore virtuale *contatto* che ritorna `TRUE` se e solo se Quadretto è a contatto con un ostacolo con uno dei suoi 4 lati.

12.7 Realizzare un sensore virtuale di distanza che rileva la distanza dall'ostacolo posto davanti. Suggerimento: avanzare fino ad arrivare a contatto con il muro, contando i passi, e poi ritornare nella situazione di partenza.

12.8 Muovere Quadretto lungo un percorso a spirale come descritto nella figura che segue, arrestando quando si arriva a contatto con il bordo.



12.9 Muovere Quadretto in un angolo.

12.10 Muovere Quadretto nell'angolo più vicino.

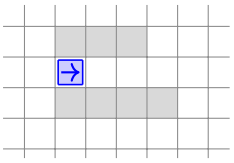
12.11 Muovere Quadretto a contatto frontale con il muro (muro esterno o interno).

12.12 Muovere Quadretto a contatto frontale con il muro più vicino (muro esterno o interno), usando il sensore di distanza.

12.13 Stabilire l'effetto dell'algoritmo 6 (*striscia*) nel caso in cui non sia vera la precondizione *tocca*(`LEFT`).

12.14 Stabilire l'effetto dell'algoritmo 8 (*costeggia*) se non ci sono ostacoli o gli ostacoli sono attaccati al muro perimetrale.

12.15 Stabilire l'effetto dell'algoritmo 20 (*esci dal tunnel*) nella situazione descritta nella seguente figura:



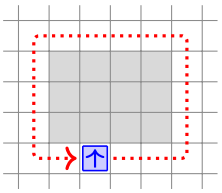
12.16 Adattare l'algoritmo 20 (*esci dal tunnel*) per gestire la situazione in cui Quadretto si trovi inizialmente di traverso all'interno del tunnel e fare in modo che cerchi l'uscita a ritroso nel caso in cui il tunnel sia chiuso ad una estremità, fermandosi nel caso in cui entrambe le estremità del tunnel siano chiuse.

12.17 Contare quante sono all'interno del reticolo di Quadretto le caselle di un dato colore.

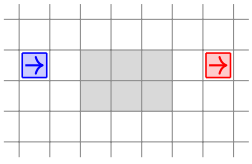
12.18 Muovere Quadretto indefinitamente avanti-indietro fra due ostacoli posti sulla direzione di movimento (uno da una parte ed uno dall'altra rispetto al verso di avanzamento).

12.19 Costruire un recinto di muro di dimensioni 3×3 collocando Quadretto all'interno.

12.20 Quadretto si trova a contatto con un muro rettangolare. Far circumnavigare Quadretto attorno al muro, a contatto con il muro sul lato sinistro, fino a tornare al punto di partenza.



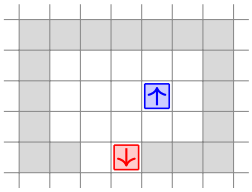
12.21 Quadretto si trova davanti ad un muro rettangolare. Portare Quadretto dall'altra parte del muro, in una posizione speculare al muro rispetto alla posizione di partenza.



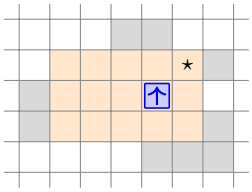
12.22 Quadretto si trova all'interno di un recinto rettangolare. Determinare il perimetro (lunghezza del perimetro interno) e l'area (numero di celle all'interno del recinto).

12.23 Quadretto si trova all'interno di una stanza rettangolare di dimensioni $m \times n$ unità ed avente su un lato un'apertura di 1 o più unità. Far uscire Qua-

dretto dalla stanza, portandolo sul bordo. Analizzare e risolvere il problema nel caso in cui non si conoscano le dimensioni della stanza.



12.24 Determinare l'area di una stanza rettangolare disarrocata, avente almento un elemento non angolare di muro su ciascun lato. In pratica si tratta di determinare il più grande rettangolo libero contenente la posizione attuale di Quadretto. Attenzione alla posizione iniziale critica denotata con $\star$ nella figura.



12.25 Quadretto si trova all'interno di una stanza rettangolare chiusa su tutti quattro i lati. All'interno della stanza è presente un ostacolo di forma generica, non appoggiato alle pareti. Individuare l'ostacolo, portando Quadretto a contatto con l'ostacolo. Analizzare e risolvere la situazione nel caso in cui la stanza sia completamente sgombra da ostacoli, fermando Quadretto nel momento in cui si sia riconosciuto che nella stanza non è presente alcun ostacolo. Generalizzare il problema ai casi in cui ci possano essere più oggetti all'interno della stanza ed al caso in cui gli oggetti possano essere appoggiati alle pareti.

12.26 Quadretto si trova davanti ad un muro (di forma irregolare) circumnavigabile. Determinare il perimetro (lunghezza del perimetro esterno del muro) e l'area (numero di caselle componenti il muro).

12.27 Nell'ambiente di Quadretto ci sono alcune caselle colorate con diversi colori (non ci sono caselle-muro); contare le caselle di un dato colore.

12.28 Nell'ambiente di Quadretto c'è un muro rettangolare, staccato dal muro perimetrale; determinare l'area del muro.

12.29 Nell'ambiente di Quadretto c'è un muro di forma generica, staccato dal muro perimetrale; determinare il perimetro del muro.

12.30 Nell'ambiente di Quadretto ci sono alcune caselle-muro unitarie, staccate dal muro perimetrale e staccate fra di loro (non si toccano neanche sugli spigoli); determinare il numero di caselle-muro presenti nell'ambiente.

12.31 Nell'ambiente ci sono alcune caselle-muro; portare Quadretto (ovunque si trovi inizialmente) ad una data casella (evitando i muri presenti nell'ambiente).