SOLUZIONE DI PROBLEMI CON PASCAL

EDIZIONE ITALIANA

Kenneth L. Bowles

GRUPPO EDITORIALE JACKSON

SOLUZIONE DI PROBLEMI CON PASCAL

di
Kenneth
L. Bowles

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

Il Gruppo Editoriale Jackson ringrazia per il prezioso lavoro svolto nella stesura dell'edizione italiana la signora Francesca di Fiore, e l'Ing. Roberto Pancaldi.

Stampato in Italia da:
S.p.A. Alberto Matarelli - Milano - Stabilimento Grafico

Fotocomposizione:
CorpoNove s.n.c. — Bergamo — via Borfuro 14/c — Tel. 22.33.63-22.33.65

PREFAZIONE

Questo libro è indirizzato sia a studenti dei primi anni di università che frequentano corsi introduttivi sulla soluzione di problemi mediante l'uso di elaboratori, sia a persone che si preparano singolarmente. Una versione precedente del libro è stata più volte usata per l'insegnamento in corsi propedeutici all'università di S. Diego in California (UCSD). Questa prefazione è rivolta agli insegnanti, o a qualsiasi persona abbastanza informata nel campo dell'elaborazione dati, per metterla in grado di consigliare i futuri studenti.

UCSD ha organizzato 10 settimane di lezioni in un trimestre e questi corsi sono stati terminati da circa il 55% degli studenti. Abbiamo impostato il corso usando il Sistema Personalizzato di Istruzione di Keller (PSI), benché l'organizzazione del libro non ne richieda l'uso. Il metodo PSI facilita leggermente l'assimilazione del materiale da parte dello studente rispetto alla normale lezione/interrogazione. PSI permette il raggiungimento di determinati livelli a seconda del numero di capitoli completati. Di norma tutti gli studenti che finiscono il corso alla UCSD completano i primi 10 capitoli con i relativi esercizi. Relativamente ad un corso semestrale, 15 capitoli dovrebbero fornire una buona quantità di materiale. Per un corso trimestrale non ci si aspetti di completare più dei primi 12 capitoli a meno di finire il corso con impegni fuori orario.

Mentre la maggior parte dei testi introduttivi è orientata a studenti che correggeranno i loro programmi su una gran varietà di elaboratori e sistemi operativi, questo libro, almeno nel prossimo futuro, si rivolgerà invece a studenti che non hanno una profonda preparazione matematica: uno dei nostri obiettivi principali è cioé di raggiungere la vasta massa di studenti usciti dalle superiori.

In generale abbiamo notato che questi studenti hanno quasi la stessa capacità nello studio della programmazione e nella soluzione di problemi mediante elaboratore, di quelli che hanno una più forte preparazione matematica. La metodologia base di programmazione non cambia sostanzialmente passando dal campo scientifico o di ingegneria a quello artistico, umanistico o degli affari. Abbiamo trovato che è possibile motivare ed insegnare agli studenti in questo campo, usando esempi di problemi con un orientamento non numerico.

Le precedenti versioni di questo libro hanno raggiunto lo scopo con lo studio della manipolazione di stringhe, dove possibile. In questa versione per aumentare l'aspetto motivazionale, abbiamo spesso fatto uso di grafici prendendo spunto dal metodo ideato da Syemour Papert al MIT denominato "Turtle Graphics".

Per raggiungere quegli studenti che non hanno mai programmato usando esempi basati su grafici e su manipolazione di stringhe, si richiede che l'implementazione di Pascal includa procedure e funzioni interne, che permettano di lavorare con quel genere di dati. I compilatori Pascal standard, – intendendo per standard le implementazioni che aderiscono strettamente alle definizioni contenute in "Pascal – Manuale e

standard del linguaggio" di Jensen e Wirth (Jackson 1980) — non contengono le funzioni e procedure interne necessarie.

Alla UCSD abbiamo implementato un sistema mono-utente completo basato sul Pascal, e questo sistema ha delle estensioni rispetto allo standard per quanto riguarda la gestione grafici e stringhe. Per tutti gli altri aspetti aderisce completamente agli standard. Questo sistema è basato su un interprete ed è destinato ad un uso interattivo su una gran varietà di microelaboratori: microelaboratori che attualmente stanno avendo un consumo quasi di massa. Si assume che il microelaboratore abbia la possibilità di gestire grafici, anche solo di tipo semplice come quelli ottenuti con sistemi "mappati a bit" su ricevitori televisivi normali. Questo sistema software è attualmente disponibile alla UCSD per utilizzo su piccoli elaboratori basati su LSI-11 della DEC (o qualsiasi altro "mini" della famiglia PDP-11), su Zilog Z80, su 8080, su 6800 e 6502. Sarà ben accetta da parte dell'autore qualsiasi richiesta di copie di questo sistema per i microelaboratori già previsti.

Naturalmente le estensioni a Pascal necessarie per far girare questo sistema in un ambiente time-sharing o comunque su grossi elaboratori, non sono di eccessiva difficoltà. Nel limite del possibile le estensioni sono state fatte in modo da evitare grandi cambiamenti al compilatore. L'autore ed i suoi collaboratori saranno lieti di cooperare, nei limiti delle risorse disponibili, con altri gruppi che desiderino implementare queste variazioni di Pascal su altri sistemi operativi.

Il corso basato su questo libro, alla UCSD, è sostenuto da: corsi sulla gestione del software, manuali di autoistruzione, ed altro materiale stampato. Saremo felici di dividere questo materiale con altre università.

La mia gratitudine per l'assistenza prestata e l'influenza esercitata va ai circa 50 laureati e studenti dell'ultimo anno dell'UCSD.

È loro la maggior parte del lavoro che ha permesso di rendere operativo il sistema software sottostante e ai loro sforzi e suggerimenti si deve il grosso successo del corso PSI tra gli studenti che vi hanno partecipato. Un riconoscimento particolare è dovuto a Mark Overgaard, che mi ha assistito fin dall'inizio, e a Robert Hofkin, Richard Kaufmann, Keith Shillington, Roger Sumner e John Van Zandt, ognuno dei quali ha contribuito in modo sostanziale. Shawn Fanning ha curato la preparazione del Glossario e dell'Indice; la maggior parte dei disegni sono stati curati da Dale Ander. Nessuna colpa potrà essere fatta loro per gli errori che inevitabilmente si troveranno in questa edizione preliminare e dei quali io solo mi ritengo responsabile. Anche gli errori di redazione e battitura sono miei soltanto dal momento che ho usato un programma per l'elaborazione dei testi che gira sotto lo stesso pacchetto su cui è basato il materiale d'insegnamento. Come spesso accade il tempo previsto per il lavoro è stato un po' troppo corto.

SOMMARIO

CAPITOLO 0

INTRODUZIONE

L'obiettivo principale di questo libro è di insegnarvi un approccio disciplinato alla soluzione di problemi usando un elaboratore. Come secondo ed inseparabile obiettivo, voi dovete imparare a scrivere programmi. Ci cureremo poco di descrivere gli elaboratori, o di esaminare i numerosissimi usi che se ne fa.

Il materiale è stato scelto per essere comprensibile a tutti gli studenti usciti dalle superiori; sostanzialmente non sono necessarie basi di matematica se non alcune nozioni di algebra. A dispetto del nostro approccio non numerico, i metodi insegnati sono gli stessi di quelli tradizionalmente usati in problemi basati sulla matematica. Il testo dovrebbe servire, nello stesso modo, a studenti di scienze, lettere, arti o ingegneria per prepararli all'uso dell'elaboratore nei campi che hanno scelto. Coloro che non useranno mai più un elaboratore dovrebbero trarre beneficio nell'usare le stesse metodologie di soluzione di problemi in altri contesti.

Questo capitolo descrive l'ambiente nel quale si ritiene che voi, come studenti, stiate lavorando. Le basi fondamentali del nostro approccio sono largamente applicabili all'uso dell'elaboratore nelle scienze, negli affari, industria, arte, comunicazioni e parecchi altri campi. Per ragioni pratiche, nessun testo può sostenere questo approccio senza trattare parecchi dettagli che non possono essere applicati senza modifiche nell'ambiente che incontrerete dopo il completamento del lavoro con questo libro. Il campo dell'elaborazione dati cambia così rapidamente che tutti gli studenti possono essere certi che si troveranno a lavorare in un ambiente molto diverso da quello in cui hanno imparato ad usare l'elaboratore. Noi abbiamo visto che parecchi studenti che completano questo testo non hanno difficoltà alcuna a passare ad altri linguaggi di programmazione o a metodi molto diversi di interazione con l'elaboratore.

1. Esempi di problemi

La soluzione di problemi non è una scienza esatta, per cui richiede una pratica che

può essere acquisita solo esercitandosi su molti problemi. La maggior parte degli "esercizi" riportati in questo libro necessita per la sua soluzione dell'uso di un elaboratore. I "problemi" alla fine di molti capitoli sono invece risolvibili con carta e penna. Con questo libro cerchiamo di fornire un approccio alla soluzione di problemi che per altri si è rivelato il migliore.

Ciononostante se non farete voi stessi uno sforzo per risolvere con l'elaboratore gran parte degli esercizi presentati, non avrete compreso il messaggio principale oggetto di questo libro.

Il materiale di molti degli esercizi e problemi contenuti nella prima parte è basato sull'elaborazione di grafici e di "stringhe" di testo italiano. Entrambi sono frequentemente usati in applicazioni su elaboratori e richiedono l'uso di gran parte delle tecniche base applicabili anche alla risoluzione di problemi matematici.

I disegni ed i testi sono familiari a tutti gli studenti. Gran parte degli studenti che lavorano su elaboratori e che lo trovano divertente, hanno avuto al liceo delle spiacevoli esperienze con la matematica. La nostra scelta degli esempi evita qualsiasi bisogno di ricorrere alla matematica.

Il libro inizia con la presentazione del repertorio di strumenti base necessari per la soluzione di problemi con l'elaboratore.

Nei primi esempi vi vengono dati molti e dettagliati suggerimenti ed il vostro obiettivo sarà quello di acquistare confidenza, attraverso la pratica, con gli strumenti presentati. Nei capitoli successivi i suggerimenti diminuiranno progressivamente e gli esercizi diverranno progressivamente più complessi.

Alla fine dovreste essere in grado di sintetizzare la soluzione completa con solo le specifiche di ciò che deve essere fatto.

Non appena i problemi diverranno un po' più complessi, vi risulterà più facile dividere ogni problema in sottoproblemi, ciascuno dei quali potrà essere risolto indipendentemente. Questo tipo di approccio del tipo "dividi et impera" è un tema che riapparirà spesso in molti e diversi contesti in questo libro. In certi casi vi sarà utile suddividere ogni componente del problema principale in modo da permettere la riduzione a passi di dimensioni più maneggevoli.

La soluzione di problemi con l'elaboratore richiede la trasposizione in termini astratti di un metodo concettuale e strutturale. Alcuni studenti sembrano considerare il bisogno di usare termini astratti come l'equivalente di lavorare con la matematica, e perciò ugualmente difficile. Può essere di aiuto considerare le astrazioni necessarie in informatica come affini a quelle necessarie nelle conversazioni quotidiane.

Nelle conversazioni usiamo costantemente dei termini astratti riferentesi ad un concetto complesso senza descrivere il concetto stesso al nostro interlocutore. Per esempio, se decidete di andare al cinema con un amico non c'è bisogno di spiegargli

il funzionamento di un proiettore e di uno schermo. In questo caso la parola "cinema" fa riferimento ad un bagaglio di informazioni di cui il vostro ascoltatore è dotato. Nello stesso modo la soluzione di problemi con elaboratore richiede l'uso di termini inventati da applicare a componenti indipendenti del problema.

Mentre si è concentrati su un determinato aspetto del problema può essere di aiuto il riferirsi agli altri aspetti dello stesso problema usando un nome specifico piuttosto che preoccuparsi dei dettagli.à

2. Algoritmi, Dati e Programmi

Un *"Algoritmo"* è costituito da un sequenza di *azioni* attraverso cui si passa per portare a termine un qualche lavoro. I *"Dati"* sono le informazioni sulle quali *"opera"* un algoritmo, vale a dire le informazioni che sono usate dalla sequenza di azioni per raggiungere il risultato voluto. Un algoritmo è simile ad una ricetta di cucina nel quale gli ingredienti sono i vari dati.

Diversamente da molte ricette, ha spesso senso usare lo stesso algoritmo per operare su, o *"elaborare"*, dati diversi aventi le stesse caratteristiche.

Un *"Programma"* è una dichiarazione astratta di un algoritmo, ed una descrizione dei dati che debbono essere trattati dall'algoritmo. Solitamente si dà per inteso che il programma deve essere espresso in termini adatti per l'interpretazione fatta da un elaboratore senza intervento umano. Un *"linguaggio di programmazione"*, come un "linguaggio naturale" come l'italiano, è un insieme di termini e di convenzioni sul come legare tra di loro i termini stessi allo scopo di comunicare i propri pensieri. Nel caso di un linguaggio di programmazione, si vuole solitamente comunicare i dettagli di un programma all'elaboratore. Molto spesso, un importante obiettivo secondario è di comunicare con altre persone, dal momento che molti linguaggi di programmazione permettono di esprimere in modo molto più preciso e conciso un algoritmo di quanto non facciano i linguaggi naturali.

Per i nostri scopi nello studio dei primi capitoli del libro, vi sarà sufficiente avere una idea grossolana della distinzione tra algoritmo e programma. Partendo dal capitolo 6, vedremo che è spesso utile, per comprensione e chiarezza, descrivere un algoritmo per mezzo di diagrammi comprensibile per le persone ma non per l'elaboratore.

3. La scelta di PASCAL come nostro linguaggio di programmazione

I linguaggi di programmazione attualmente più usati sono: BASIC, COBOL e FOR-

TRAN. Meno popolari, seppur largamente usati, sono: APL, PL/1, ed in minor grado ALGOL. Perchè dunque abbiamo scelto di usare PASCAL in un libro di tipo introduttivo?

PASCAL è stato progettato per ovviare a molti degli inconvenienti riscontrati nell'uso dei suddetti linguaggi verso la fine degli anni Sessanta. Venne introdotto da Niklaus Wirth dell'Università di Zurigo come base per l'insegnamento di un approccio sistematico alla soluzione di problemi ed alla programmazione di elaboratori.

PASCAL ha inoltre delle ottime possibilità per quanto riguarda la manipolazione di dati complessi, possibilità che ne hanno progressivamente incrementato l'uso nella programmazione di vasti progetti in campo industriale.

Verso la metà del 1977, PASCAL era usato nell'insegnamento in circa 400 fra scuole ed università ed il suo uso si andava diffondendo rapidamente. Una delle grandi università statali dove è stato usato per circa 4 anni, riferisce che la popolarità di PASCAL ha quasi raggiunto quella di FORTRAN. Recentemente è stato reso noto che PASCAL è disponibile per l'uso su almeno 50 differenti tipi di elaboratori di diversa fabbricazione.

PASCAL è uno dei migliori linguaggi a larga diffusione per un insegnamento di tipo introduttivo di quei concetti conosciuti in campo industriale sotto il nome di *"programmazione strutturata"*.

La Programmazione Strutturata è un metodo ideato per facilitare il programmatore nella ricerca e correzione degli errori di logica contenuti nei programmi. Inoltre, è un metodo ideato per permettere al programma di produrre risultati corretti malgrado un minimo di errori da parte del programmatore. Le maggiori società di informatica analizzano la produttività dei propri programmatori e insistono sull'uso dell'approccio disciplinato della Programmazione Strutturata. Questo tipo di approccio può essere usato con uno qualsiasi dei linguaggi precedentemente citati, è però consigliabile evitare l'uso di questi linguaggi in situazioni che sono ormai provate essere fonti di errore.

Abbiamo constatato che gli studenti che imparano a programmare usando PASCAL hanno poche difficoltà a passare all'uso di FORTRAN. Nel fare ciò sono naturalmente portati a servirsi di un approccio strutturato. Il passaggio all'uso di COBOL necessita di alcune settimane di preparazione, questo perché COBOL ha delle regole più complicate da affrontare prima che si possano scrivere dei programmi corretti. Il passaggio a BASIC può essere ottenuto in poche ore dando la dimostrazione che questo linguaggio manca di tutte le possibilità fornite da PASCAL.

In conclusione la via migliore è quella di imparare a programmare usando PASCAL e di passare poi ad altri linguaggi qualora ve ne siano le ragioni. Una di queste ragioni potrebbe essere la necessità di comunicare sulle proprie attività di programmazione

con persone che fanno largo uso di un altro linguaggio. Lo studio della Programmazione Strutturata partendo con uno dei linguaggi sopra citati, comporta un'esperienza simile a quella di imparare a camminare in un'area accidentata dove un piccolo inciampo porterebbe ad una caduta in un burrone.

La versione PASCAL usata in questo libro include tutte le caratteristiche del linguaggio come era stato originalmente progettato dal Professor Wirth; l'"originale" non prevedeva però tutte quelle possibilità che permettono ai neofiti di lavorare con grafici e stringhe che costituiscono l'oggetto principale di questo libro. Ecco perché abbiamo cercato di estendere PASCAL. Quando raggiungerete il livello in cui vorrete usare PASCAL su un elaboratore diverso da quello utilizzato nello studio con questo libro, vi sarà molto utile rivedere l'Appendice A che riporta un elenco delle differenze fra la nostra versione estesa e quella originale di Wirth.

4. Strumenti — micro, mini e maxi elaboratori

L'industria degli elaboratori definisce le piccole macchine mono-utenti e poco costose *"micro elaboratori"*. Una macchina abbastanza grande da poter essere divisa fra diverse persone che lavorano contemporaneamente su dei programmi di modeste dimensioni, e che costa normalmente da 50 a 100.000 milioni è chiamata *"mini elaboratore"*. Le macchine più grandi usate da più persone e con programmi più grossi saranno chiamate in questo libro *"maxi elaboratori"*. Il gergo industriale è meno standardizzato per quanto riguarda i maxi elaboratori di quanto lo sia per i mini e micro.

Le attuali tendenze dell'industria degli elaboratori fanno supporre con sempre maggiori probabilità, che userete un micro elaboratore sia durante lo studio con questo libro che successivamente. Fino a poco tempo fa, la maggior parte delle scuole ed università usava dei maxi o mini elaboratori, *utilizzati* da diverse persone contemporaneamente. La crescita nell'uso dei micro elaboratori è stata estremamente rapida verso la metà del 1977, e se ne prevede un ulteriore aumento. È ormai evidente che verso la metà del 1980 la maggior parte degli utenti userà dei micro piuttosto che dei mini o maxi elaboratori.

La ragione di questa rapida crescita è che le nuove tecniche di produzione su vasta scala dei componenti elettronici, stanno riducendo sensibilmente il costo dei micro elaboratori tanto da permettere lo sviluppo di un mercato di massa.

Le estensioni a PASCAL necessarie nello studio di questo libro, possono essere fatte, in teoria, su ogni elaboratore che disponga di un compilatore PASCAL. L'installazione di queste estensioni a macchine di una certa dimensione richiede un certo sforzo. Presso la UCSD è stato possibile fornire l'attuale generazione di micro elaboratori del PASCAL completo, con sforzo minimo. A meno che non stiate usando una

versione di PASCAL derivata da quella sviluppata dalla UCSD, dovrete fare attenzione alle differenze fra la versione che state usando e quella riportata in questo libro.

5. Dispositivi per la visualizzazione di grafici

La maggior parte delle illustrazioni qui riportate è una copia dei disegni eseguiti da un micro elaboratore con programmi PASCAL. Esiste una grande varietà di dispositivi utilizzabili per la stesura di grafici attraverso elaboratori. Una delle ragioni principali di questa grande disponibilità deriva dal fatto che le immagini di alto livello sono ottenibili esclusivamente con l'uso di dispositivi abbastanza costosi, mentre è possibile ottenere delle approssimazioni di queste immagini con un semplice apparecchio televisivo e con della elettronica aggiuntiva, poco costosa. Fra questi estremi esiste una grande possibilità di scelta di dispositivi di qualità sufficiente per un uso didattico, e che costano da 1 a 3 milioni. Oltre ai dispositivi usati per *"visualizzare"* un'immagine su un video, simile allo schermo di un televisore, ne esistono altri che registrano le immagini in forma permanente su carta (*"hard-copy"*).

Quasi tutte le illustrazioni riportate in questo libro sono state registrate su carta usando due dei dispositivi di registrazione "hard-copy" prodotti dalla Tektronix Corporation. Uno di questi disegna linee con penna ed inchiostro. L'altro registra su carta l'immagine apparente sullo schermo di un dispositivo progettato per l'interazione con elaboratori. Molti dei terminali video della Tektronix sono usati da scuole per l'insegnamento dell'informatica. Dobbiamo alla Tektronix il prestito di questi strumenti che ci sono serviti durante la preparazione di questo libro.

A seconda del micro elaboratore che userete studiando con questo libro, l'aspetto dei grafici potrà differire leggermente da quello riportato nel libro stesso, la figura 0-1 ne mostra la ragione. Molti dispositivi non usano lo schermo come un "continuum" ma, piuttosto, creano l'apparenza di una linea tracciando una sequenza di punti ad intervalli regolari. Di solito i punti possono essere riportati solo nei punti di intersezione delle linee di un immaginario foglio di carta per grafici. La figura 0-1 mostra i punti, le linee da essi raffigurate, e la quadrettatura definente il foglio per grafici. Risulta meno costoso produrre un dispositivo che riporta l'immagine nei punti di intersezione di uno che la riporti esattamente nei punti da voi desiderati. Il risultato è una linea che, in alcuni casi, ha l'aspetto di una scala. Meno raffinato sarà l'immaginario foglio per grafici e più appariscente sarà l'aspetto a scala della linea tracciata dal vostro elaboratore. Con un po' d'immaginazione dovreste riuscire ad ignorare l'aspetto a zigzag ed a vedere la linea tracciata come se fosse continua.

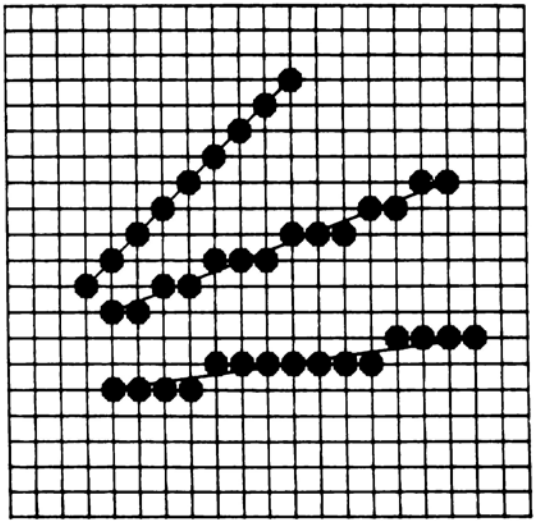


Figura 0-1

6. Impostazione del libro

Sembra non esistere un ordine ideale con cui poter presentare il repertorio di strumenti base che bisogna possedere per poter scrivere programmi. È difficile capire la ragione dell'uso di un approccio strutturato alla soluzione di problemi fino a quando non si ha imparato a scrivere dei programmi semplici. L'approccio usato in questo libro è quello di partire con degli esempi molto semplici usando una selezione di strumenti di base. Nei capitoli successivi gli esempi diverranno progressivamente più difficili affinchè possiate esercitarvi a sintetizzare soluzioni complete di problemi. Per ogni capitolo successivo si presume che abbiate compreso e studiato il materiale oggetto dei capitoli precedenti. Questa è l'unica ragione per cui si consiglia di procedere nello studio del libro seguendone l'ordine di presentazione invece che di seguire un ordine di preferenza personale.

Per molti studenti uno degli aspetti più fastidiosi nell'apprendere l'uso di elaboratori è rappresentato dalla necessità di conoscere dettagli sull'ambiente di lavoro che non sono generalmente riportati nei libri di testo. Il capitolo 1 di questo libro, approfondisce questi argomenti più di quanto non si faccia normalmente, presupponendo però che userete una variante del sistema software PASCAL della UCSD. Se al contrario non userete questo sistema, vi sarà disponibile una appendice relativa al sistema da voi usato, in sostituzione del capitolo 1.

I capitoli dal 2 al 6 presentano degli strumenti di base per la programmazione e la stesura di algoritmi. I capitoli dal 7 al 10 aggiungono nuovi strumenti per poter lavorare con dati trasmessi all'elaboratore da dispositivi esterni, e per poter elaborare dei dati complessi. Il capitolo 11 riguarda la controversa istruzione GOTO. I capitoli dal 12 al 15 riportano problemi complessi del tipo frequentemente incontrato da ogni programmatore. Le appendici alla fine del libro servono per consultazioni.

7. Gergo usato in informatica

Nell'ambiente dell'informatica, a molti dei concetti diventati ormai familiari vengono dati dei nomi o delle abbreviazioni comunemente riconosciute. Spesso il nome o la parola usata si rifà, nel caso migliore, a parole che nel linguaggio quotidiano servono a descrivere un insieme di concetti leggermente differenti; alcune volte è presa, tale e quale, dall'inglese. I termini usati dai tecnici per comunicare fra di loro permettono una sintesi maggiore. Qui ci si richiama all'esempio "del cinema", precedentemente citato.

In questo libro cercheremo di evidenziare i termini specialistici di questo gergo, mettendoli fra virgolette e sottolineandoli quando appariranno per la prima volta nel testo. Qualora, in seguito, aveste dimenticato il significato di un termine potrete consultare il glossario nell'appendice B. Detto glossario riporta una breve definizione o descrizione della parola, ed anche un riferimento al capitolo e alla sezione dove questa appare per la prima volta. Una volta introdotti useremo questi termini liberamente. Alcuni studenti contestano questo modo di procedere perché richiede loro l'apprendimento, in breve tempo, di molte parole estranee; ciò può essere fastidioso ma è necessario. Senza l'uso di questi termini il libro risulterebbe molto più lungo e difficile da leggere.

8. Gli obiettivi

Ogni capitolo comincia con una breve esposizione degli obiettivi che dovreste raggiungere prima di passare al capitolo successivo. Sono molti gli studenti che si perdono in dettagli senza riuscire a capire i punti principali di un libro o di un corso di studi. Generalmente un autore delinea le intestazioni dei capitoli e delle sezioni in modo da definire il quadro generale del materiale che verrà presentato. Gli obiettivi differiscono leggermente dalle intestazioni in quanto sintetizzano nozioni che potreste avere sotto controllo solo unificando tutte le intestazioni delle sezioni di un capitolo.

Molti degli studenti che hanno usato una precedente versione di questo testo, han-

no trovato gli "obiettivi" molto utili. Vi suggeriamo di leggere dapprima gli obiettivi, poi il capitolo come un tutto unico e di risolvere quindi gli esempi posti alla fine di ogni capitolo. Poi, prima di proseguire con il successivo, rileggete di nuovo gli obiettivi del capitolo appena finito, questo dovrebbe aiutarvi a rilevare i punti principali di tutto il capitolo.

9. Come studiare

La soluzione di problemi con gli elaboratori richiede sia creatività che buona volontà per seguire un certo numero di regole ben precise e rigide. Alcuni studenti trovano difficile abituarsi all'uso di regole ben precise, credono che l'elaboratore dovrebbe, in qualche modo, capire quello che è il loro pensiero anche deviando un po' dalle regole. Se la pensate in questo modo, convincetevi che l'elaboratore non possiede nessuna intelligenza. Tutto ciò che può fare è di seguire una serie logica di passi (un algoritmo) dopo essere stato precedentemente programmato per farlo. Procedendo con il corso, potrete capire come sia difficile prevedere tutti i modi possibili con cui un utente può chiedere all'elaboratore di eseguire anche il più semplice dei compiti. La dimensione e complessità del programma necessario per far fronte alla variabilità umana sono allo stadio attuale, troppo grandi. Fino a quando non avremo trovato il modo di rendere l'elaboratore più intelligente con meno sforzi di programmazione, dovremo affrontare la necessità di essere precisi se si vuole operare proficuamente.

Una conseguenza della necessità di essere precisi è che questo testo deve essere scritto in modo da concentrare in relativamente poche pagine una grande quantità di informazioni. Non potrete leggere del materiale di questo tipo come se si trattasse di un romanzo, affrontabile con delle tecniche di lettura rapida ed aspettandovi nello stesso tempo di trarne un qualsiasi beneficio.

In alcuni casi la mancanza di osservanza delle regole è prontamente svelata da un programma (chiamato *"compilatore"*) che traduce il linguaggio di programmazione in una forma comprensibile alla macchina stessa. In altri casi questa non osservanza è manifestata da risultati sbagliati ed inattesi durante lo svolgimento del programma. Gli errori svelati dal compilatore sono, in molti casi, facili da correggere in quanto il compilatore visualizza dei messaggi che permettono di risalire allo sbaglio. Gli errori di logica sono più difficili da trovare, e spesso richiedono un approccio sistematico per trovarne la soluzione. Sviluppi di questo approccio sistematico dovrebbero costituire uno dei vostri obiettivi nello studio di questo libro.

Psicologi che hanno compiuto studi nell'ambito della programmazione con elaboratori, hanno suggerito diversi tipi di approcci allo sviluppo dei programmi, che fun-

zionano bene per molte persone. Ma un determinato approccio può non andar bene a tutti. Alcuni dei seguenti suggerimenti risulteranno più comprensibili rileggendoli dopo aver completato alcuni dei capitoli successivi.

9a. Soluzione di gruppo di esercizi

Spiegate il vostro progetto ad un compagno prima di provarlo direttamente sull'elaboratore. In caso di errori è molto più probabile che li possiate trovare voi stessi durante la spiegazione che non il vostro amico anche se conosce perfettamente la materia. Spesso troverete gli errori più velocemente in questo modo che non controllando i risultati direttamente sull'elaboratore. Ricordate che l'elaboratore fa ciò che voi gli avete *chiesto* di fare e non necessariamente ciò che era nelle vostre *intenzioni*.

9b. Documenti preparatori

Fate una descrizione scritta degli obiettivi che il programmma deve raggiungere. Disegnate dei grafici o delle tabelle da allegare a questo *"documento"* e che contribuiscano alla spiegazione. Anche se il documento consistesse di soli appunti, promemoria, è molto importante fare lo sforzo di scrivere queste note allo scopo di "vedere" subito se la logica di un programma è corretta.

9c. Controllate più volte il vostro lavoro

Scrivete la soluzione iniziale di un problema di programmazione poi mettetela da parte per alcuni giorni. Prima di passare la soluzione sull'elaboratore rivedete il vostro scritto e cercate di capirne la logica. Il processo di rivedere ciò che avete fatto in precedenza vi permette spesso di trovare quelle inconsistenze logiche facilmente correggibili con carta e penna. In alcuni casi l'uso diretto dell'elaboratore, non permetterebbe la scoperta dell'errore se non dopo un lasso di tempo maggiore.

9d. Sperimentare

Siate sempre disponibili a *sperimentare* (con semplici programmi illustrativi) sull'elaboratore, qualora le descrizioni scritte non vi siano del tutto chiare. Se le descrizioni scritte riportassero tutte le possibili idee errate di tutti i potenziali lettori, i mate-

riali di tipo descrittivo risulterebbero troppo vasti per essere letti dalla maggior parte delle persone. È possibile trovare alcuni esempi semplici che verifichino la vostra comprensione del funzionamento di un punto particolare considerato dal manuale. Provare alcuni esempi di questo tipo vi fa risparmiare tempo, è inoltre più produttivo del cercare di indovinare il funzionamento delle cose per poi dimenticarle in un programma più complicato.

9e. Progetto modulare

Usate un approccio modulare nella stesura di algoritmi e di programmi. Questo è un tema ricorrente in tutto il libro. Assicuratevi che ogni blocco sia corretto prima di unirlo agli altri nella creazione di una struttura completa. A questo scopo è utile la creazione di speciali dati di verifica da usarsi esclusivamente per esaminare un singolo componente. Uno sforzo in questa verifica anticipata vi può far risparmiare degli sforzi maggiori nel controllare il programma completo.

9f. Analizzate — Non congetturate

Non cercate di risolvere le difficoltà contenute in un programma facendo delle *congetture* a caso. Questo viene da noi chiamato "l'approccio del tiro a segno", ed è uno dei modi più comuni con il quale gli studenti sprecano gran parte del loro tempo. *Analizzate* l'esecuzione del vostro programma per verificare che produca o meno il risultato desiderato. Se introducete un cambiamento nel programma, abbiate una ragione logica per farlo. Controllate che il risultato ottenuto dopo il cambiamento si accordi con il risultato da voi atteso.

CAPITOLO 1

PREPARAZIONE

1. Obiettivi

Lo scopo principale di questo capitolo è di "introdurvi" all'elaboratore, al modo con cui lo usiamo ed ai metodi che utilizzeremo per descriverne l'uso.

1a. Lancio del programma TARTARUGA per disegnare figure sullo schermo dell'elaboratore.

1b. Apprendimento dell'uso dell'Editor per leggere e modificare i programmi GRAFICO 1 e STRINGA 1.

1c. Compilazione e svolgimento dei programmi campione, con ritorno al punto 1b fino a quando non si raggiunge un'idea di ciò che essi compiono e su come si possono modificare per ottenere risultati differenti.

1d. Studio dell'uso delle carte sintattiche come metodo più veloce per capire le regole di costruzione di programmi usando il linguaggio di programmazione PASCAL. Studio della costruzione di un <identificatore>, e di un <programma> completo costituito da diverse <istruzioni>.

1e. Introduzione intenzionale di alcuni errori di sintassi nel programma, compilazione ed osservazione del modo in cui il compilatore tenta di informarvi su ciò che è stato sbagliato.

1f. Lancio di uno dei programmi da voi verificato sotto controllo del programma correttore (Debugger). Esecuzione di una istruzione alla volta ed osservazione di come l'azione svolta si riferisce alla parte del programma in esecuzione.

2. Comandi all'elaboratore

La struttura dell'elaboratore che potete vedere e toccare, chiamata in gergo "hard-

ware", può eseguire delle operazioni primitive che modificano in qualche modo i dati immagazzinati nella sua memoria. Ciò che permette all'elaboratore di portare a termine delle serie di azioni complesse è la sua capacità di immagazzinare delle sequenze di istruzioni, ognuna delle quali dice all'elaboratore di compiere determinate azioni primitive. Queste sequenze di istruzioni immagazzinate costituiscono i *"programmi"*, e sono immessi in memoria così come lo sono i dati. In questo libro presteremo poca attenzione alle istruzioni primitive che l'hardware stesso comprende. Ci interesseremo invece di ciò che riguarda l'utilizzo di alcuni dei programmi che sono già stati approntati per il vostro uso. Questi programmi fanno apparire l'elaboratore come capace di eseguire complesse e sofisticate istruzioni. Comunque anche questi vasti programmi possono essere cambiati abbastanza facilmente, quando se ne conoscano le modalità, per questo sono chiamati *"software"*. La maggior parte delle persone che usa elaboratori, non deve generalmente preoccuparsi del funzionamento dell'hardware, che è l'oggetto di lavoro per pochi specialisti.

Il software che userete lavorando con questo libro è stato progettato in modo tale che voi possiate "conversare" con l'elaboratore. Come primo passo nella comprensione di quanto appena affermato, dovreste ricevere una breve dimostrazione su come rendere operativo il *"sistema"* software, da parte di qualcuno che lo sappia già usare. Useremo molto spesso la parola "sistema" per definire l'insieme dei programmi che costituiscono il software. Nella maggior parte dei casi, userete degli elaboratori i quali richiedono che il software sia caricato in memoria da un supporto removibile come per esempio un disco morbido. Il metodo di immissione in memoria varierà da un elaboratore all'altro. Per questo, nel caso in cui non trovaste nessuno che vi dia una prima dimostrazione, dovrete ricorrere ad un piccolo manuale di istruzioni relativo al vostro elaboratore.

Appena il sistema software entra in funzione, appariranno sullo schermo (o sul terminale a tastiera) abbinato all'elaboratore, qualcosa di simile alle seguenti linee:

```
Command: E(dit), R(un), F(ile), C(ompile), X(ecute)

U.C.S.D. PASCAL SYSTEM 11.3B
```

La seconda di queste linee semplicemente vi informa che il sistema è in funzione e sta aspettando istruzioni su cosa fare. La prima linea chiamata *"linea di suggerimento"* vi comunica quale parte del software è attualmente in funzione ("Command:" in questo caso), e quali istruzioni potete dare al sistema partendo da questo punto. L'invio di un ordine si fa premendo il tasto, facente parte di una tastiera simile a quella di una macchina da scrivere, corrispondente alla lettera iniziale dell'ordine voluto. Per esempio, per mettere in esecuzione un programma MIOPROG (useremo spesso i termini "eseguire" e "girare" con lo stesso significato) dovete premere il tasto "X". Il sistema vi risponde facendo apparire un messaggio richiedente il nome del programma che deve essere eseguito. Dopo aver battuto sulla tastiera il nome del program-

ma voluto, completate la vostra risposta premendo il tasto di Ritorno (RET). Se il programma indicato è disponibile nella libreria su dischi connessi all'elaboratore, questo programma verrà messo in esecuzione, cioè comincerà a girare.

Ritorneremo successivamente su ciascuno degli ordini che appaiono nella linea di suggerimento Command, sopra riportata. Ciò che è importante capire a questo punto è che l'elaboratore non possiede un'intelligenza propria. Potete fargli eseguire determinate azioni dicendogli di cominciare a far girare un dato programma o parte di esso. Nel sistema software di tipo conversazionale potrete far questo selezionando uno dei tanti comandi che sono stati programmati nel sistema. Per comunicare al sistema quale comando avete selezionato, premete il tasto corrispondente al comando desiderato.

3. Disegno di semplici figure con comandi

Se l'elaboratore da voi usato può disegnare linee, sarà disponibile in libreria un programma chiamato TURTLE (tartaruga). (Qualora il vostro elaboratore non disegni linee, studiate il testo così com'è e provate a risolvere gli esercizi usando il programma tipo descritto nella sezione 11 di questo capitolo). Per lanciare questo programma partendo dal livello di "Command:", premete il tasto X (di eX(ecute), la "E" è riservata per E(dit)), poi battete:

TURTLE

seguito dal tasto di ritorno RET(urn). Il programma sostituirà la linea di suggerimento "Command:" con la propria e visualizzerà un segno simile ad una freccia, vicino al centro dello schermo.

Questo segno, un esempio del quale potete vedere al punto (a) della figura 1-1, è chiamato TARTARUGA perché può essere usato quasi allo stesso modo della tartaruga meccanica controllata da un elaboratore, fabbricata parecchi anni fa da un gruppo di ricerca del Massachusetts Institute of Technology. Su alcuni schermi vi potrà risultare difficile discernere la direzione verso cui punta la tartaruga. In questi casi ricordatevi che quando il programma ha appena iniziato a girare essa è orizzontale e rivolta verso il lato destro del video.

Il programma TARTARUGA risponde a dei comandi che permettono di disegnare sullo schermo utilizzando un'appropriata sequenza dei comandi stessi. Il metodo da impiegare per dire al programma quale comando usare differisce da quello descritto nella sezione precedente che si riferisce al livello "Command:", e da quello descritto in altre parti del sistema software principale. Il sistema a comandi risponde alla pressione di un solo tasto. Il programma TARTARUGA richiede la battitura per intero del

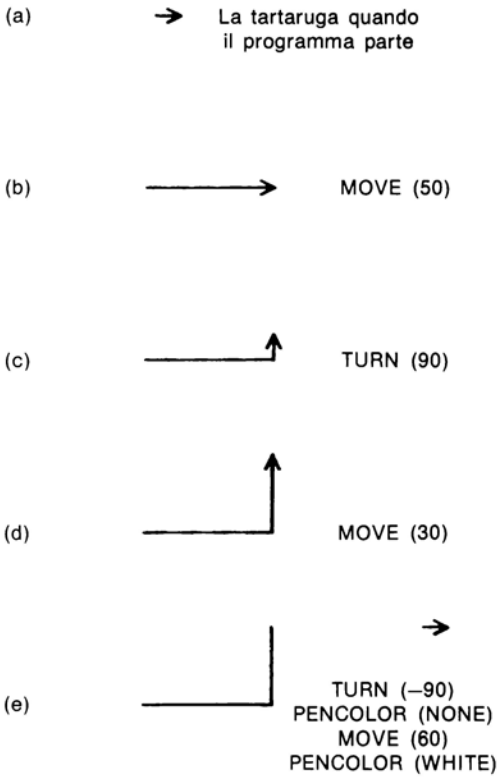


Figura 1-1

nome del comando. Il programma è stato progettato in questo modo così da permettervi di acquisire la pratica e la comprensione necessarie per scrivere programmi usando il linguaggio PASCAL. Come esempio provate il seguente comando sul vostro elaboratore (dopo l'avvio del programma TARTARUGA con il comando "X"):

MOVE (50) <RET>

dove <RET> indica che dovete premere il tasto di ritorno. Il risultato che dovreste ottenere è indicato al punto (b) della figura 1-1. In successivi riferimenti a MOVE e ad altri comandi Tartaruga, ometteremo di ricordarvi l'uso del tasto <RET> per evitare di imbrattare le illustrazioni con troppe indicazioni.

Se fate un errore di battitura, il programma visualizzerà un messaggio indicante che non ha capito il comando da voi inviato mediante la battitura dei tasti. Ciò dimo-

stra che il programma TARTARUGA vi permette di ripetere l'operazione senza nessun problema. Il sistema infatti non trasmette il vostro messaggio di comando al programma fino a quando non premete il tasto < RET >, inoltre potete eliminare le lettere battute per errore partendo dall'ultima, con l'uso dei tasti < Backspace > o < Rubout >. Il tasto < Backspace > cancella una lettera alla volta; < Rubout >, alle volte indicato come < DELete >, cancella l'intero comando.

Il numero 50 comunica al programma la lunghezza della linea che dovrà apparire sullo schermo. Più alto è il numero, più lunga sarà la linea. La lunghezza esatta della linea che otterrete usando il numero 50, dipende dalle caratteristiche del vostro video, dipendenti dalla marca dell'elaboratore usato. Alcune prove dovrebbero indicarvi se dovete alzare o abbassare i numeri suggeriti per ottenere linee di lunghezza ragionevole. Il programma TARTARUGA richiede che il numero indicante la lunghezza della linea, venga messo fra parentesi tonde.

In qualsiasi punto dello schermo si trovi, potete comandare alla Tartaruga di cambiare direzione usando il comando TURN, come dalla figura 1-1, punto (c). Il successivo comando MOVE gli farà disegnare una linea nella nuova direzione, punto (d). Il numero usato con il comando TURN è misurato in *gradi* e si riferisce all'angolo fra la vecchia direzione e la nuova, in senso antiorario. Se desiderate girare la tartaruga di 45° verso destra, cioè in senso orario, allora usate TURN(—45). Entrambi i comandi TURN(180) o TURN(—180) obbligano la Tartaruga ad invertire completamente la direzione. TURN(270) è l'equivalente di TURN(—90). Non c'è nessuna utilità nell'uso di numeri superiori a 359 in quanto TURN(360) fa fare alla Tartaruga un giro completo, e la fa ritornare nella stessa direzione da cui era partita. Se avete dimenticato, o non avete mai studiato, gli angoli, lasciate che sia la Tartaruga stessa ad insegnarveli facendo delle prove con il tasto TURN ed assegnando dei numeri varianti da —180 a +180 gradi.

Altri due comandi Tartaruga vi saranno utili sperimentando con questo programma. Provate ad immaginare la Tartaruga meccanica in possesso di una penna a sfera con cui disegna su della carta posta sul pavimento e sul quale le è stato ordinato di muoversi. Sul video che state probabilmente usando, la tartaruga disegnerà una linea bianca sullo sfondo nero. Qualora desideriate spostare la tartaruga in un altro punto dello schermo senza disegnare alcuna linea, usate il comando

```
PENCOLOR(NOME)
```

seguito dagli ordini appropriati: MOVE e TURN. Potrete ottenere di nuovo la linea con i comandi Move usando:

```
PENCOLOR(WHITE)
```

(anche se il vostro video disegna delle linee verdi!). Su alcuni schermi è possibile cancellare le linee già fatte usando:

```
PENCOLOR(BLACK)
```

Il secondo comando aggiuntivo è semplicemente:

```
CLEARSCREEN
```

che cancella qualsiasi disegno dall'inizio del programma, permettendovi di ricominciare da capo.

ESERCIZIO 1.1:
Riferendovi alla figura 1.2, usate il programma TARTARUGA ed i comandi abbinati per disegnare le figure qui riportate. Non abbiate timore di sperimentare con la Tartaruga fino a quando non raggiungerete l'effetto desiderato.

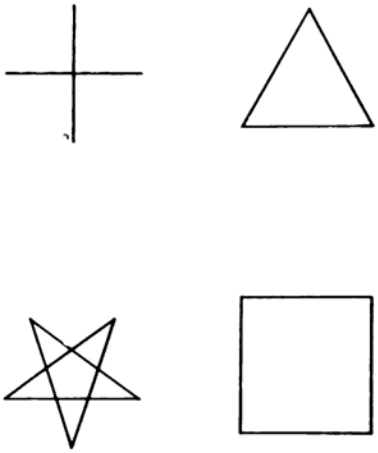


Figura 1-2

4. Un programma PASCAL con l'uso della Tartaruga

Invece di usare i comandi Tartaruga in modo convenzionale, come nella sezione precedente, possiamo scrivere e conservare un programma che esegua una sequen-

za di ordini Tartaruga. Un esempio di questo tipo di programma è riprodotto più sotto come "PROGRAM GRAFICO 1" (GRAPH 1), di cui ne dovrebbe esistere copia nell'insieme dei programma di cui verrete forniti assieme al sistema software. Come primo passo provate ad eseguire questo programma usando l'ordine "X" al livello di "Command:". Quando il comando "X" vi richiede il nome di un programma inserite:

GRAPH1 <RET>

Il programma dovrebbe visualizzare uno dei disegni riportati in figura 1-2. Non dovreste avere difficoltà nell'associare i comandi successivi alla figura che viene visualizzata dal programma che sta girando.

```
 1: PROGRAM GRAPH1;
 2: BEGIN
 3:    MOVE(100);
 4:    TURN(120);
 5:    MOVE(100);
 6:    TURN(120);
 7:    MOVE(100);
 8:    READLN  (*impedisce la cancellazione di quanto visualizzato*)
 9:            (*fino alla battitura del tasto <RET>*)
10: END.
```

Il primo "comando" eseguito dal programma è MOVE in linea 3. I successivi, nelle linee 4, 5, 6, 7 e 8, sono poi eseguiti in ordine. Quando questi comandi sono usati in un programma, ognuno viene definito come un'*"Istruzione eseguibile"*. L'istruzione READLN (read a line, leggi una linea) è qui usata per arrestare il programma al punto 8 fino a quando non premerete il tasto <RET>, significante che una linea è finita. Questo è necessario per mantenere sullo schermo la figura visualizzata fino a quando non sarete pronti a terminare il programma e a tornare al livello di "Command:". Per evitare confusione con comandi successivi, il sistema libera automaticamente lo schermo alla fine di ogni programma.

In questo programma ogni istruzione è separata dalla successiva da un punto e virgola ";". Il punto e virgola produce lo stesso effetto ottenuto dall'uso del tasto <RET> quando queste istruzioni rappresentavano dei comandi inviati al programma TARTARUGA.

I numeri che appaiono alla sinistra di ogni linea non fanno parte del programma, essi sono introdotti nell'illustrazione dei programmi come mezzo di riferimento a punti specifici dei programmi stessi.

La nota riportata fra i due asterischi "(*" e "*)" nelle linee 8 e 9 del programma

GRAPH1, costituisce un *"commento"* all'istruzione, e non è parte eseguibile del programma.

5. Modificare un programma con l'Editor

Usate ora la "E" del comando E(dit) al livello "Command:" che farà partire il programma editor fornito con il sistema software. L'editor vi richiederà il nome del *"flusso"* che intendete usare. In risposta inserite "GRAPH1" seguito, come sempre, da <RET>. Dopo i pochi istanti necessari affinchè venga immesso in memoria il testo relativo al programma GRAPH1, l'editor visualizzerà la sua linea di suggerimento seguita da tante linee del GRAPH1 quante ce ne possono stare sul vostro video.

L'editor utilizza un segnaposto chiamato *"cursore"*, simile per certi aspetti alla Tartaruga. Determinati comandi vi permetteranno di dirigere il cursore verso qualsiasi punto del testo del programma. Nella maggior parte dei casi la vostra tastiera è provvista di quattro frecce direzionali: destra, sinistra, alto, basso. In assenza di queste frecce cercate i comandi equivalenti descritti nelle istruzioni di cui è corredato il vostro elaboratore. Per abbassare il cursore di una linea, premete la freccia rivolta verso il basso; per spostarlo di un posto verso destra, premete la freccia rivolta verso destra e così di seguito. Se il cursore si trovasse a fianco della prima o dell'ultima linea attualmente sullo schermo, l'uso della freccia diretta verso il basso o l'alto, rispettivamente, sposterà la parte di testo vista attraverso il video in modo tale da mantenere il cursore sullo schermo.

Alcune tastiere permettono di spostare il cursore con rapidità, semplicemente tenendo abbassata la freccia appropriata tanto a lungo quanto vi pare necessario. (Altre tastiere possiedono un apposito tasto <REPEAT> che si deve tenere premuto assieme al tasto che deve essere ripetuto). Qualora la vostra tastiera sia sprovvista di questi mezzi, potrete battere il numero corrispondente al numero di volte che uno stesso comando deve essere ripetuto, seguito dalla battitura del comando stesso.

Esistono altri due comandi che servono a modificare un programma, cioè a *"Editare"* il testo PASCAL del programma. La pressione del tasto "I", di I(nsert), vi permette di aggiungere delle nuove linee al programma, oppure di inserire dei caratteri nelle linee già esistenti. I caratteri da inserire iniziano immediatamente *prima* del punto in cui era situato il cursore al momento della pressione del comando "I". La parte di programma il cui carattere iniziale è indicato dal cursore viene spostata sullo schermo per darvi lo spazio necessario all'inserimento di nuovi caratteri. Quando avete terminato la battitura delle nuove informazioni, potete togliere il controllo dal comando I(nsert) mantenendo abbassato il tasto <Control> e premendo il tasto "C". Questo per far sì che le informazioni inserite vengano considerate come parte del testo

del programma. Se, dopo avere inserito diversi caratteri, decidete di non conservarne il risultato, potete ritornare al testo iniziale, prima di I(nsert), premendo il tasto <ESCape>.

L'altro comando importante per la modifica di un programma è "D" di D(elete). Dopo l'inserimento del comando D(elete), i normali tasti di posizionamento del cursore saranno utilizzabili. La parte di testo da cancellare si trova fra la posizione del cursore al momento dell'inserimento di D(elete) e la posizione che raggiunge appena prima che il comando sia completato con il tasto <RETurn>. Spostando il cursore sotto il controllo di D(elete), i caratteri che devono essere rimossi dal testo verranno eliminati dallo schermo. Come si è visto precedentemente per il comando I(nsert), potete usare <ESCape> per completare il comando D(elete) in modo tale da ritornare al testo esistente prima dell'inserimento di D(elete).

In entrambi i comandi, I(nsert) e D(elete), potete cancellare un carattere alla volta usando il tasto <Backspace>, annullando l'effetto dell'azione precedente.

> ESERCIZIO 1.2:
> Usate l'Editor per modificare il programma GRAPH1 così che possa disegnare un quadrato, una croce, una stella o un'altra figura diversa da quelle riportate in figura 1.2.

6. Lancio del programma modificato

Per far girare (effettivamente eseguire) un programma da voi scritto o modificato con l'Editor, dovete prima ritornare al livello "Command:". Per far ciò usate la "Q" del comando Q(uit) dell'Editor. Dopo poco tempo la linea di suggerimento "Command:" dovrebbe riapparire. Ora premete la "R" del tasto R(un). Pochi secondi dopo sullo schermo apparirà il messaggio "compiling...".

Per poter capire le azioni successive, dovete sapere che il programma PASCAL da voi modificato con l'Editor deve essere tradotto in una forma comprensibile all'hardware dell'elaboratore. Il software comprende anche un programma di traduzione chiamato in gergo "*compilatore*" tradotto dal compilatore PASCAL in una forma eseguibile. Nell'eseguire il comando R(un), il sistema controlla dapprima se avete appena modificato il vostro programma. Se così fosse richiederà l'intervento del compilatore che traduce il programma PASCAL in forma eseguibile. Per tenervi informati su ciò che sta succedendo, il compilatore visualizza dei messaggi mostranti la progressione in atto. Per un tipo di programma breve come GRAPH1, l'intero processo di compilazione dovrebbe essere compiuto in 2 o 3 secondi. La maggior parte di questo tempo è usata per la lettura del compilatore nella memoria, dalla libreria su disco.

Se il compilatore non trova nessun errore nel vostro programma PASCAL editato, il

sistema comincerà ad eseguirlo non appena il compilatore avrà terminato il suo lavoro. Sarete avvisati di ciò quando il messaggio "running..." apparirà sul video. Le azioni successive dipendono dall'impostazione del programma.

Purtroppo può succedere che si facciano delle modifiche di edizione al programma che il compilatore non può capire. Riparleremo di come poter stabilire se le proprie modifiche sono esatte partendo dalla sezione 8. Nonostante che il compilatore sia un programma complesso e sofisticato, non è molto intelligente e non è in grado di capire ciò che voi *intendevate* fare anche solo in presenza di un semplice errore che passerebbe inosservato a chiunque. Quando il compilatore trova un errore non può continuare nel processo di traduzione fino a quando l'errore stesso non viene corretto. Apparirà allora un messaggio descrivente la classe dell'errore trovato. Inoltre, il software vi rimanda automaticamente all'Editor e visualizza quella parte del programma in cui è stato trovato l'errore; il cursore vi indicherà il punto esatto. Successivamente, lavorando su programmi più vasti, vi potrà succedere di preparare un programma contenente parecchi errori con l'incombenza di passare attraverso una sequenza di E(dit)—R(un), seguita da ritorno automatico all'Editor. Esiste un metodo per utilizzatori esperti di elaboratori, che permette la scoperta di tutti gli errori contenuti in un programma con un unico uso del comando R(un).

ESERCIZIO 1.3:
Modificate il programma GRAPH1 in modo da disegnare delle figure differenti, come nell'esercizio 1.2, poi lanciate (R(un)) il programma per controllarne i risultati. Ripetete questo procedimento per altre diverse figure.

7. Libreria e flusso-lavoro

Il sistema software dispone di metodi per conservare e recuperare programmi su dischi magnetici collegati all'elaboratore. (In alternativa ai dischi vi capiterà di usare nastri magnetici o cassette). Le informazioni memorizzate su dischi sono raggruppate in unità chiamate "*flussi*". Quando l'Editor, vi chiede quale flusso volete usare o quando il comando "X" richiede il nome di un programma, la risposta da voi data provoca il caricamento del flusso desiderato da disco a memoria.

Quando modificate un determinato flusso con l'Editor, viene fatta una copia temporanea del programma che viene conservata sul disco in un flusso speciale chiamato "*flusso-lavoro*". Se, quando mettete in funzione l'Editor con il comando "E", sul disco esiste un precedente flusso-lavoro, l'Editor visualizzerà la prima parte di questo flusso-lavoro invece di richiedervi il nome del flusso che desiderate usare.

Ora che conoscete l'esistenza ed il significato di un flusso-lavoro, vi è possibile ca-

pire la distinzione fra i comandi R(un) e X(ecute). R(un) è progettato per lavorare esclusivamente con il flusso-lavoro, e presume che vogliate l'esecuzione del programma contenuto nel flusso-lavoro stesso. Se, quando usate il comando R(un), sul disco non è presente nessun flusso-lavoro, apparirà un messaggio di errore. X(ecute) vi richiede il nome del programma che deve essere eseguito e presuppone che il programma richiesto sia già stato compilato correttamente, e che sia conservato nella libreria sotto un nome appropriato. Un programma compilato correttamente ed inserito nel flusso-lavoro, può essere più volte eseguito usando R(un), senza ricorrere ogni volta all'uso del compilatore.

Allo scopo di conservare nella libreria un programma del flusso-lavoro, sotto un nome specifico, dovete usare la "F" del comando F(ile) al livello "Command:". Dopo poco tempo la linea di suggerimento del comando (F(ile) apparirà sullo schermo. A questo punto premete la "S" di S(ave) che vi richiederà il nome da assegnare al programma.

Per avere un elenco dei flussi già conservati in libreria, usate il comando L(ist) al livello di "File:". Noterete che la maggior parte dei flussi hanno suffisso "TEXT" o "CODE". I flussi con suffisso "TEXT" hanno una forma tale per cui possono essere letti o modificati usando l'Editor; quelli con suffisso "CODE" sono il risultato della traduzione, fatta dal compilatore, dei corrispondenti flussi con suffisso "TEXT" in una forma comprensibile dell'hardware.

Il livello "File:" prevede numerosi altri comandi che permettono di lavorare proficuamente con i flussi su memoria di massa (dischi...). R(emove) permette l'eliminazione di un determinato flusso ottenendo così spazio disponibile per altri usi. N(ew) crea un nuovo flusso-lavoro con cui potete iniziare un nuovo programma usando l'Editor. (G(et) sostituisce l'attuale flusso-lavoro con copia di un altro flusso già in memoria. C(hange) vi permette di cambiare il nome di un flusso già inserito nella libreria. La maggior parte di questi comandi vi richiede quale risposta l'inserimento del nome del flusso desiderato, quando è necessario. Se un comando avesse l'effetto di eliminare l'attuale flusso-lavoro, ne verrete informati e vi sarà richiesto se avete l'intenzione di continuare con il comando, oppure no. Rispondete con Y per Y(es), se desiderate continuare. Qualsiasi altra risposta sarà considerata come "No".

ESERCIZIO 1.4:
Dopo aver verificato sul video che il programma del vostro flusso-lavoro produce i risultati desiderati, come nell'esercizio 1.3, salvate in libreria il suddetto programma sotto un nome da voi inventato. In seguito verificate nella direttrice che ciò sia stato compiuto. Ora create un nuovo flusso-lavoro, editate e lanciate un nuovo e semplice programma. Fatto questo ritornate al livello di "File:", togliete il nuovo programma dal disco e verificate che R(un) al livello di "Command:" è rifiutato.

8. Carte sintattiche

Fin qui tutto è dipeso dalla vostra capacità di deduzione su come si possano scrivere dei programmi privi di errori, facendo riferimento al programma tipo GRAPH1. Molto presto ne saprete abbastanza su come scrivere dei programmi PASCAL, che la semplice deduzione dagli esempi non sarà più sufficiente. In possesso anche di pochi degli strumenti di programmazione PASCAL, sarete in grado di comporre dei nuovi programmi abbastanza facilmente. Il numero di possibili programmi senza errori è così alto da rendere necessario l'uso di un metodo più preciso del semplice uso di esempi, per spiegare come si possa costruire un programma "legale" o "corretto". A questo scopo useremo delle "*carte sintattiche*".

L'Italiano, come tutte le altre lingue naturali, è descritto da regole di grammatica che stabiliscono i modi di costruzione di frasi "corrette". Queste regole definenti l'ordine con cui differenti classi di parole (nomi, verbi, aggettivi, avverbi,...) possono essere usate, così come le norme di punteggiatura, sono chiamate regole "sintattiche". Allo stesso modo le regole che descrivono come si possano scrivere programmi per elaboratori sono dette regole sintattiche. In Italiano potete contravvenire a queste regole e ciononostante essere compresi. Gli elaboratori, a parte rare eccezioni, non sono abbastanza intelligenti o abbastanza flessibili per capire delle istruzioni che abbiano violato le regole di sintassi. Chi desidera usare gli elaboratori, deve perciò imparare ad esprimere i propri pensieri in modo conforme alle regole di sintassi del linguaggio di programmazione usato.

Uno dei vantaggi del linguaggio di programmazione PASCAL rispetto ad altri, è che le regole sintattiche PASCAL possono essere descritte in modo chiaro e semplice per mezzo di diagrammi sintattici. In questo capitolo cominceremo con pochi esempi che ci aiuteranno a descrivere dei programmi tipo. Alcuni di questi esempi rappresentano delle versioni semplificate dei diagrammi riportati integralmente nell'ultima appendice di questo libro.

In un programma PASCAL potete assegnare dei nomi diversi alle varie celle della memoria dell'elaboratore. Mentre l'elaboratore fa riferimento a queste celle perlopiù numerandole, per noi è molto più facile assegnar loro un nome che serva anche da riferimento all'oggetto della cella stessa. Questi nomi sono sia un aiuto mnemonico che di comprensione del funzionamento di un programma. Uno dei compiti principali del compilatore è quello di tenere un dizionario di questi nomi e di tradurli poi nel numero corrispondente quando produce un flusso eseguibile ".CODE". Questi nomi sono detti in gergo "*identificatori*". Le regole sintattiche stabiliscono che un identificatore debba essere costruito come segue:

1) Cominciare con qualsiasi lettera compresa fra "A" e "Z".
2) Far seguire alla prima qualsiasi sequenza di lettere (A...Z) e/o cifre (0...9).

Quelli che seguono sono alcuni esempi di sequenze di caratteri che potete usare come identificatori in un programma PASCAL:

X
ABC
CORTO
PRIMOIDENTIFICATORE
P27
L914PDK

Questi di seguito sono esempi di sequenze di caratteri, non accettabili come identificatori in quanto non sono conformi alle regole sintattiche:

3RD
(714)452-4050
UNITA-1
I 12.34
DUE PAROLE
Minuscola

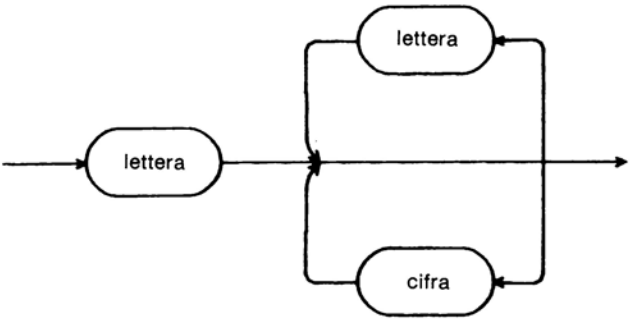


Figura 1-3: Diagramma sintattico di <identificatore>

La figura 1-3 mostra, sotto forma di diagramma, le regole sintattiche per la formazione di identificatori. Ciò che non risulta dal diagramma è che solamente i primi 8 caratteri di un identificatore sono significativi per il compilatore PASCAL che voi userete. Potete utilizzare identificatori con più di 8 caratteri, ma solamente i primi 8 verranno presi in considerazione dal compilatore per distinguere gli identificatori uno dall'altro. Ecco perchè il compilatore giudicherà identici i due seguenti identificatori, nonostante che a noi appaiano differenti:

MARIAGIOVANNA MARIAGIOCONDA

In questo libro faremo spesso riferimento ad un oggetto descritto dalle regole sintattiche mettendolo fra parentesi angolari, per esempio <identificatore>. Quando trovate un qualsiasi riferimento scritto fra parentesi angolari, come sopra, queste ultime dovrebbero costituire un indizio a cercare la descrizione dell'oggetto del riferimento nelle carte sintattiche. All'interno dei quadri delle carte sintattiche, ometteremo generalmente le parentesi angolari, ma continueremo ad usare dei caratteri minuscoli per voci che sono definite da altri diagrammi. Una parola riportata nei diagrammi in CARATTERI MAIUSCOLI è una "*Parola riservata*" (Reserved word), avente per il compilatore un significato speciale. Una parola riservata dovrebbe apparire nel programma con la stessa ortografia e nella stessa relativa posizione con cui appare nel diagramma sintattico. I caratteri di punteggiatura con speciali significati sintattici, appaiono nei diagrammi inseriti in cerchi.

9. Sintassi per <programma> e <blocco>

Fate ora riferimento alla figura 1-4 che illustra le sintassi semplificate per <programma> e <blocco>. Confrontate il programma GRAPH1 con questo diagramma ed assicuratevi di capirne le correlazioni. (Ripetiamo ancora che i numeri relativi alle linee compresi nei programmi tipo di questo testo, non fanno parte dei programmi stessi, e non vengono quindi considerati dalla sintassi. Vengono usati solamente a scopo di riferimento, nell'ambito del testo scritto, per specifiche voci del programma). Nella discussione seguente, i numeri assegnati alle linee si riferiscono al programma GRAPH1.

Il diagramma dice che ogni programma dovrebbe iniziare con una linea:

PROGRAM <identificatore>;

come in linea 1. L'<identificatore> riferisce il nome del programma al compilatore, rappresenta inoltre un riferimento al testo del programma stesso, ma non stabilisce automaticamente il nome del flusso di tipo ".TEXT". Potete usare qualsiasi <identificatore> lecito come nome nel programma.

Il diagramma stabilisce che un programma completo è costituito da una linea PROGRAM completata da un punto e virgola (";") e seguita da un <blocco> seguito a sua volta da un punto ("."). Il punto è essenziale. Come potrete ben presto notare, un programma può contenere la parola END molte volte. Il punto comunica al compilatore che l'ultimo END è stato raggiunto, cioè che è la fine del programma.

Per conoscere il significato di <blocco>, guardate la parte (b) del diagramma in figura 1-4. Potete notare che un <blocco> inizia con la parola riservata BEGIN e ter-

mina con l'altra parola riservata END. Fra queste due parole riservate ci potrà essere un numero qualsiasi di <istruzioni> separate fra loro da un punto e virgola.

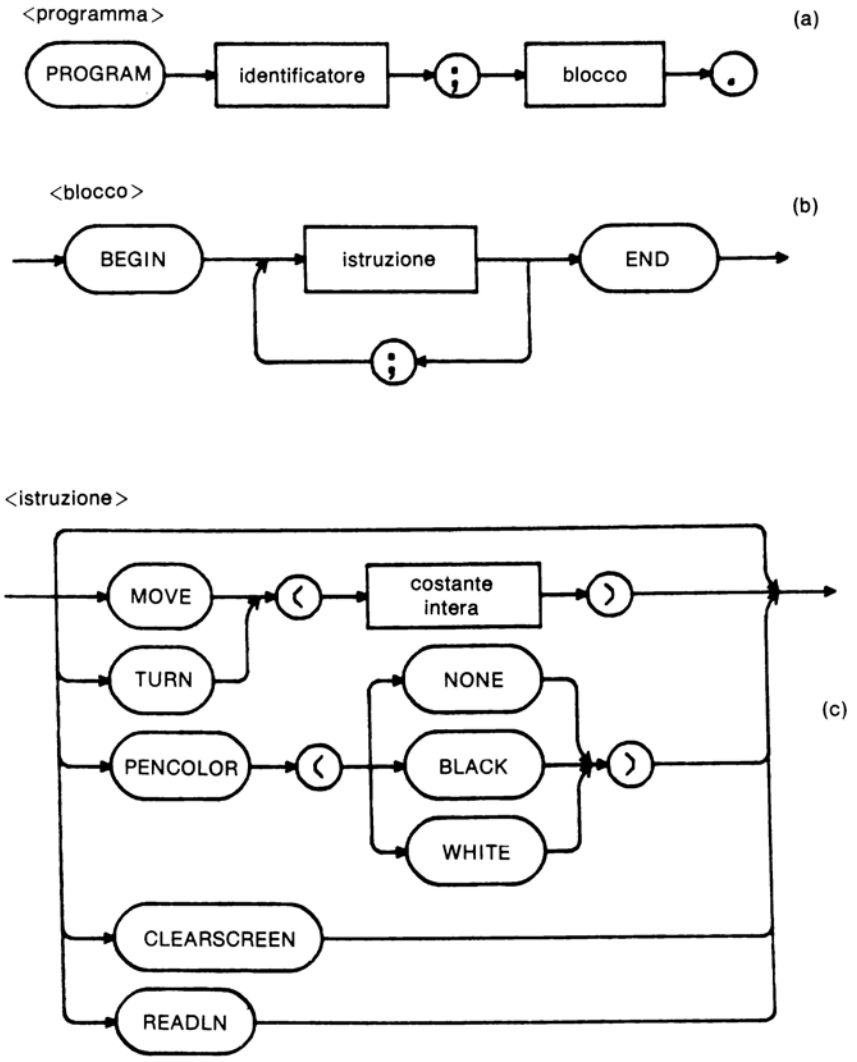


Figura 1-4.

ESERCIZIO 1.5:
Provate a modificare alcune delle versioni fatte partendo dal programma GRAPH1, introducendo di proposito degli errori di sintassi. Osservate cosa viene riportato dal compilatore sull'errore trovato per mezzo del comando R(un) a livello di "Command:". Ecco alcuni esempi di errori: omettete una delle parentesi da istruzioni quali MOVE o PENCOLOR; omettete il punto e virgola che separa due istruzioni fra di loro; sbagliate l'ortografia di una delle parole riservate; omettete BEGIN; omettete il punto che segue END; e così di seguito. Dovreste notare che il messaggio di errore visualizzato dal compilatore non sempre corrisponde al motivo per cui l'errore dovrebbe essere stato trovato. Esistono migliaia di differenti modi di introduzione di errori in un programma, ma c'è spazio solamente per circa 300 messaggi di errori. Il programmatore che ha scritto il compilatore non era in grado di prevedere tutte le possibili combinazioni di circostanze che possono portare a violazioni specifiche delle regole sintattiche che a voi possono apparire rilevanti.

10. Il correttore

Quando in un programma viene introdotto un errore che provoca dei risultati sbagliati, si dice in gergo che è stato introdotto un "*baco*" (bug). La rimozione dei bachi da un programma è detta "*debugging*". In quanto non esiste alcun programma in grado di determinare se il vostro programma ha eseguito ciò che era nelle vostre intenzioni (allo stesso modo con cui il compilatore trova gli errori di sintassi e ve li comunica) siete voi che dovete stabilire in altro modo, se un programma lavora correttamente. Come potrete ben presto constatare, esistono modalità di esecuzione diverse per ciascun programma tanto da non permettere il controllo della correttezza delle operazioni per ogni esecuzione. È comunque possibile verificare separatamente piccole parti di programma, e trattare poi queste parti quali blocchi elementari per ottenere un risultato più complesso. Se ciascun blocco funziona correttamente, così dovrebbe fare l'intero programma, a condizione che ogni blocco elementare sia stato progettato correttamente in relazione agli altri.

Il sistema software che userete abbinato a questo libro, include un programma di utilità chiamato "Debugger" che vi aiuterà nella verifica delle operazioni di un programma e nel trovare i bachi. In questa sezione ci interesseremo di uno solo degli aspetti del debugger, gli altri verranno introdotti in seguito.

Quando siete al livello "Command:" potete far girare un programma sotto il controllo del debugger usando la "D" di D(ebug) invece di R(un). Al posto di "run-

ning...", apparirà la linea di suggerimento del debugger con alcune informazioni addizionali descriventi il programma. Proseguite ora nell'esecuzione usando la "J" del comando J(ump) del debugger. Ogni volta che J(ump) viene usato, verrà eseguita un'istruzione del programma, che verrà visualizzata sullo schermo. Nella maggior parte dei casi potrete anche osservare l'azione connessa all'esecuzione dell'istruzione. Per capire il funzionamento di un programma è spesso utile eseguirlo con questo metodo del "*passo-passo*".

Per i semplici programmi discussi in questo capitolo, potete seguire il metodo di esecuzione passo-passo usando degli appunti scritti contenenti l'intero programma. Non è possibile visualizzare il programma completo quando il debugger è inserito, in quanto lo spazio è necessario per altre informazioni. Quando passerete a dei programmi più grossi, vi sarà utile avere a disposizione un "*listato*" del programma con cui state lavorando. Potrete così fare delle note in margine, che vi saranno utili dopo essere tornati all'Editor. Dovrebbe generalmente essere disponibile una stampante adatta alla stesura di queste copie. I modi d'uso di queste stampanti variano molto rispetto al tipo di elaboratore usato. Per questa ragione dovrete essere istruiti da qualcuno che sa come usare la stampante a vostra disposizione.

> ESERCIZIO 1.6:
> Lanciate sotto il controllo del debugger uno dei programmi con il quale avete lavorato in precedenza, usando il comando D(ebug). Siate certi di poter identificare l'azione compiuta dopo l'esecuzione dell'istruzione visualizzata. (Generalmente entrambe, la precedente e successiva istruzione vengono visualizzate.)

11. Programma tipo con l'uso di <stringhe>

Se il vostro elaboratore non prevede il disegno di linee, il seguente programma tipo vi aiuterà a capire molti dei punti che verranno presentati in questo capitolo. Questo programma mostra delle "*stringhe*" di caratteri come se ne trovano nei normali testi italiani. Attraverso tutto questo libro lavoreremo con grafici e stringhe in quanto entrambi sono importanti nell'uso pratico degli elaboratori. Lavorando con entrambi questi tipi di informazioni, così come con i numeri, partendo dal capitolo 5, avrete una comprensione più completa della soluzione di problemi con l'elaboratore di quanto non sia possibile con l'uso singolo di ognuno di questi tipi.

```
1: PROGRAM STRINGA 1;
2: BEGIN
3:    WRITE('CI');
4:    WRITE(' ', 'SONO');
```

```
5:    WRITELN; (*va all'inizio della prossima linea*)
6:    WRITE('CI SONO');
7:    WRITELN('QUESTA È UNA DIMOSTRAZIONE')
8:    WRITELN('DELL'ESECUZIONE DEL PROGRAMMA')
9: END
```

Potete trovare le sintassi semplificate delle istruzioni WRITE e WRITELN, usate in questo programma nella figura 1-5. Queste istruzioni fanno sì che i dati messi fra parentesi appaiano quali messaggi sul vostro video. La combinazione delle linee 3 o 4 produce lo stesso effetto della linea 6, infatti visualizzano:

CI SONO

Il motivo per cui in questo esempio li presentiamo separatamente è quello di mostrare che messaggi successivi inviati allo schermo non WRITE, seguono semplicemente i messaggi precedentemente inviati. In tutti i casi dove viene inviato un messaggio (linee 3..8, eccetto in linea 5), i dati sono costituiti da una o più <costante stringa >. La sintassi della <costante stringa > è anch'essa mostrata nella figura 1-5. Una <costante stringa > consiste di qualsiasi sequenza di caratteri delimitati da un apostrofo posto alla sinistra e alla destra della sequenza stessa. Questi apostrofi "*delimitano*" la <costante stringa > e sono detti "*delimitatori*". Notate che gli apostrofi non vengono visualizzati da WRITE o WRITELN, e che non sono considerati parte dei dati racchiusi nella <stringa >. Notate ancora che la <stringa > ' ' racchiude uno

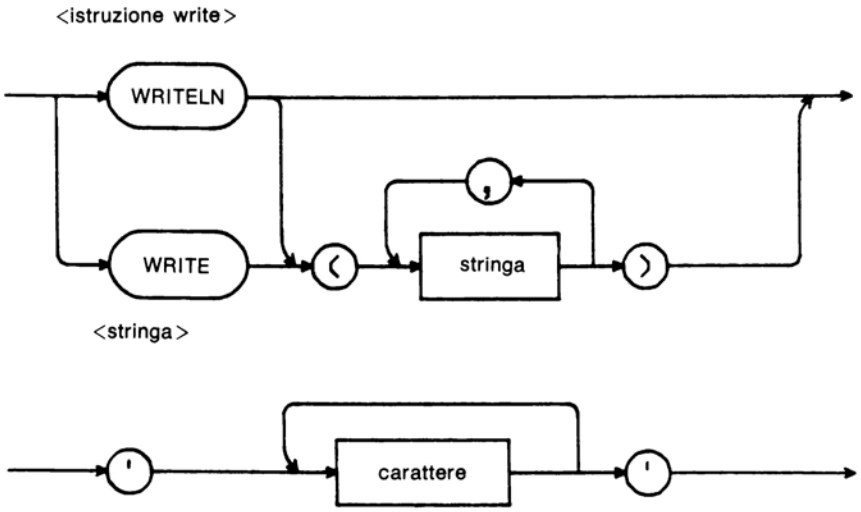


Figura 1-5: Sintassi semplificata per WRITE e WRITELN prima di figura 6-13.

spazio vuoto. Se volete la visualizzazione del carattere relativo a uno spazio vuoto, dovete richiederla all'elaboratore.

Prima di continuare, provate a pensare a come far apparire un messaggio contenente un solo apostrofo, cioè il simbolo di un apostrofo vero e proprio (').

L'istruzione WRITELN è simile a WRITE, ad eccezione che, seguendo WRITELN, l'istruzione WRITE successiva comincerà a sistemare i caratteri all'inizio della linea seguente. In questo modo il programma STRINGA1 dovrebbe visualizzare il seguente messaggio completo:

```
CI SONO
CI SONO QUESTA È UNA DIMOSTRAZIONE
DELL'ESECUZIONE DEL PROGRAMMA
```

La risposta al quesito su come visualizzare un solo apostrofo, è di usare due apostrofi in successione per *ogni* apostrofo che volete visualizzare. Per esempio WRITE ('''') visualizzerà un solo apostrofo. I due apostrofi centrali corrispondono al carattere richiesto. Il primo e l'ultimo sono i regolari delimitatori.

ESERCIZIO 1.7:
Modificate e lanciate il programma STRINGA1 per fargli visualizzare ciò che segue:

```
CI SONO AMICI
QUESTA È UNA DIMOSTRAZIONE
DELL'ESECUZIONE DI UN PROGRAMMA "PASCAL"
```

CAPITOLO 2

PROCEDURE E VARIABILI

1. Obiettivi

In questo capitolo imparerete come suddividere un programma in semplici unità di azione chiamate "*procedure*" e come attribuire dei nomi, scelti da voi, alle locazioni di memoria, dove i dati vengono depositati: le "*variabili*".

1a. Definizione ed uso delle procedure che disegnino delle figure semplici, quali: triangoli, quadrati, stelle, etc. Disegno di figure complesse richiamando queste stesse procedure più volte.

1b. Uso di parametri per controllare le dimensioni delle figure disegnate dalle vostre procedure, e per controllare il punto sullo schermo dove esse sono state disegnate.

1c. Definizione ed uso di variabili del tipo: CHAR, INTEGER, e STRING per conservare e riutilizzare più tardi diversi dati.

1d. Apprendimento dell'uso di procedure e funzioni interne, progettate per introdurre dei cambiamenti nelle stringhe.

1e. Uso di espressioni aritmetiche semplici per calcolare nuovi valori interi combinando variabili intere e costanti.

1f. Prendere familiarità con le regole sintattiche PASCAL localizzando errori in diversi esempi di programmi.

2. Premessa

Avete già visto abbastanza sulla programmazione per capire che scrivere un pro-

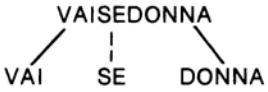
gramma che disegni delle figure complesse può risultare alquanto tedioso. PASCAL ha delle possibilità, come del resto le hanno la maggior parte dei linguaggi di programmazione, che vi permettono di suddividere un programma in piccole unità più maneggevoli, e delle possibilità per ripetere più volte le medesime azioni. Questo ed il prossimo capitolo presentano le principali caratteristiche PASCAL per raggiungere i suddetti scopi.

Per capire la necessità della suddivisione di un programma in piccole parti, provate a ricordare le due stringhe seguenti, composte da 10 lettere ciascuna:

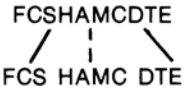
VAISEDONNA FCSHAMCDTE

Coprite questa pagina dopo esservi concentrati su queste due stringhe e verificate per quanto tempo potete ricordare ognuna di esse. È probabile che non abbiate avuto molte difficoltà nel ricordare la prima stringa, ma a meno che non siate un mago con memoria fotografica, avrete avuto bisogno di un maggior sforzo per ricordare la seconda.

Se vi interessate di psicologia non c'è dubbio che conosciate delle spiegazioni per questa differenza: da un lato la memoria a breve termine in contrapposizione a quella a lungo termine; dall'altro il fatto che un certo numero di oggetti è direttamente collegabile, nella nostra memoria, ad altri oggetti. Una spiegazione semplice può essere la seguente: grazie al modo con cui avete imparato a leggere, potete suddividere automaticamente la prima stringa in tre stringhe più brevi e familiari.



Ciascuna delle stringhe più piccole, rappresentano delle parole a voi più familiari. Per quanto riguarda l'altra stringa, non è possibile trovare delle sotto-stringhe che siano di alcun aiuto mnemonico. Naturalmente è possibile suddividerla:



Ma questi nuovi gruppi di lettere non sono di alcuna utilità nel ricordare la stringa intera in quanto nessuno di essi ha per noi un qualsiasi significato.

Questo per arrivare al punto: un programma dovrebbe essere suddiviso in piccoli gruppi di istruzioni (chiamati alle volte "moduli" o "sotto-programmi"), ognuno dei quali sia considerabile come esecutore di una singola azione. All'interno di questi

gruppi devono essere eseguite diverse istruzioni al fine di espletare l'azione desiderata. Comunque una volta che il gruppo è stato scritto, potete pensarlo come un tutto unico e dimenticare il fatto che consiste di diverse istruzioni indipendenti. Qando scrivete un programma con un numero di linee maggiore di quante ce ne possano stare su una pagina, da 25 a 50 linee circa, diventa complicato considerarlo come un'unità semplice. A questo punto nasce la necessità di dividere il programma in moduli separati.

3. Procedure

Una "*procedura*" è in effetti un piccolo programma all'interno di un altro programma. Considerate i quattro quadrati oggetto della figura 2-1. Prima di procedere immaginate come costruire un programma che tracci questi quattro quadrati usando i metodi introdotti nel capitolo 1. Dovreste accorgervi che dopo il completamento del primo quadrato le istruzioni scritte per disegnarlo si ripeteranno quando dovrete eseguire il secondo o terzo quadrato.

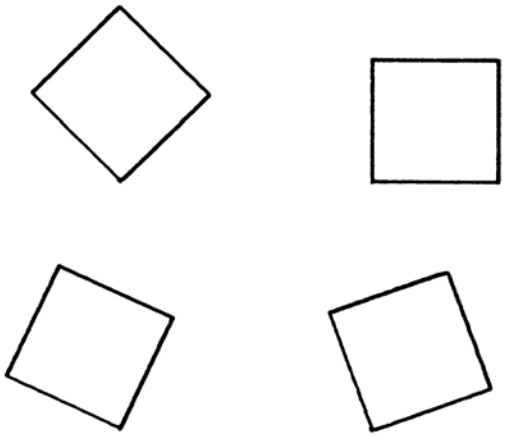


Figura 2-1

Riferitevi ora al programma tipo QUADRATI che esegue la stessa azione, infatti questo è il programma che ha tracciato la figura 2-1. Le linee dal 3 al 15 costituiscono la procedura UNQUADRATO. Come potete notare se queste linee fossero state usate quali basi di un programma completo, sostituendo la parola PROGRAM a PROCEDURE, questo programma avrebbe disegnato un quadrato sullo schermo. (Se non

avete ancora capito perchè un quadrato dovrebbe essere tracciato, è il momento di fare un piccolo esperimento per osservare cosa succede).

Considerate ora le linee dal 17 al 31, costituenti il programma principale QUADRATI. In linea 20 la tartaruga viene spostata di 150 unità verso destra e di 150 unità verso l'alto, partendo dal centro del video. MOVETO è simile a MOVE in quanto sposta la tartaruga in un nuovo punto del video. Al comando MOVETO devono essere attribuiti due numeri: il primo misura lo spostamento della tartaruga verso destra, il secondo lo spostamento verso l'alto, sempre partendo dal centro dello schermo. Se il primo numero fosse negativo, come in linea 22, allora la tartaruga si sposterebbe a sinistra, cioè all'opposto. Se il secondo numero fosse negativo, come in linea 28, allora la tartaruga si sposterebbe in basso, sotto al centro del video. Mentre MOVE provoca lo spostamento della tartaruga di determinate unità partendo dalla sua attuale posizione, MOVETO la sposta ad un'altra posizione partendo dal centro dello schermo. (Qualora questa spiegazione non vi fosse chiara, fate delle prove con il comando MOVETO usando il programma TURTLE come dal capitolo 1, sezione 3).

Dopo aver spostato la tartaruga in alto nella parte destra dello schermo in linea 20, "*chiamiamo*" ora la procedura UNQUADRATO, in linea 21. La linea 21 indica all'elaboratore di trasferire l'ordine di esecuzione alla linea 5 che è la prima istruzione "*eseguibile*" della procedura. Con "eseguibile" intendiamo dire che è la prima linea in cui ci sia un'azione diretta. L'elaborazione continua fino alla linea 14, cioè fino a quando END è raggiunto in linea 15. Le istruzioni a partire dalla linea 5 fino alla 14 provocano il tracciamento del quadrato nella parte alta e destra dello schermo, come mostrato in figura 2-1.

```
 1: PROGRAM QUADRATI;
 2:
 3: PROCEDURE UNQUADRATO;
 4: BEGIN
 5:    PENCOLOR(WHITE);
 6:    MOVE(150);
 7:    TURN(90);
 8:    MOVE(150);
 9:    TURN(90);
10:    MOVE(150);
11:    TURN(90);
12:    MOVE(150);
13:    TURN(90);
14:    PENCOLOR(NONE);
15: END (*UNQUADRATO*);
16:
17: BEGIN (*PROGRAMMA PRINCIPALE*)
```

```
18: (*NOTA: I VALORI DI MOVE SONO PER IL TEKTRONIX 4006*)
19: (* CAMBIANO PER ALTRI TERMINALI*)
20:    MOVETO(150,150);
21:    UNQUADRATO;
22:    MOVETO(-150,150);
23:    TURN(45);
24:    UNQUADRATO;
25:    MOVETO(-150,-150);
26:    TURN(20);
27:    UNQUADRATO;
28:    MOVETO(150,-150);
29:    TURN(-45);
30:    UNQUADRATO;
31: END.
```

Avendo raggiunto la linea 15 la procedura UNQUADRATO è terminata, così come sarebbe terminata nel caso in cui si fosse trattato di un programma completo. In questo caso, comunque, il controllo del programma ritorna al punto immediatamente seguente a quello in cui la procedura UNQUADRATO è stata chiamata, cioè alla linea 22. La linea 21 indica all'elaboratore di elaborare la procedura fino al termine della stessa. Avendo finito, l'elaborazione continua con l'istruzione successiva, come consueto.

Con il raggiungimento della linea 22, la tartaruga viene spostata nella parte sinistra in alto. In linea 23 viene girata di 45 gradi verso sinistra in preparazione di ciò che segue. Segue un'altra chiamata di UNQUADRATO che provoca il disegno di un altro quadrato. Ancora una volta la linea 24 trasferisce il controllo del programma alla linea 5. Questa volta, al termine della procedura, il programma continua dal punto successivo alla linea 24, linea da cui è partita la chiamata della procedura.

Notate che nell'elaborare il programma fin qui, siamo passati due volte attraverso le istruzioni contenute nella procedura UNQUADRATO. Nel programma completo questo avverrà per quattro volte, una volta per ogni quadrato tracciato nella figura 2-1. Notate inoltre che nel programma PASCAL la procedura appare *prima* della parte principale del programma. Vi potrà essere di aiuto sapere che il programma PASCAL è tradotto in "linguaggio macchina" dal compilatore leggendolo con il nostro stesso ordine, cioè partendo dall'alto e lavorando poi su ogni linea successiva. Dunque, il programma comincia a elaborare dopo la linea BEGIN corrispondente al <blocco > principale del programma, cioè, in questo esempio, partendo dalla linea 17. È necessario che quando il compilatore raggiunge la linea 21 sappia tutto sulla procedura UNQUADRATO cosicché sia possibile continuare nella elaborazione della procedura stessa. Questa è una delle ragioni per cui, in PASCAL, una procedura appare prima del programma principale. Si dice che la procedura è "*dichiarata*" attra-

verso le linee comprese fra 3 e 15. Ciò significa che descriviamo in anticipo le azioni che saranno svolte dalla procedura qualora essa venga chiamata dal programma principale.

Questo è il primo punto in cui è importante che voi capiate la distinzione fra lo stadio dove il programma viene compilato e quello successivo in cui viene elaborato (cioè "lanciato" o "eseguito"). Ricordate che prima il compilatore deve tradurre le istruzioni PASCAL in forma comprensibile all'elaboratore stesso. Il compilatore ha esigenza di mantenere traccia delle varie parti del programma, e questo spiega la ragione dell'esistenza di norme relative all'ordine in cui queste parti vengono scritté Dopo che un programma è stato compilato con successo, può essere eseguito. L'ordine di esecuzione non corrisponde esattamente all'ordine di apparizione delle istruzioni, quando vengano usate delle procedure. Nel capitolo successivo vedremo altri modi di alterazione dell'ordine di esecuzione.

Riassumendo il contenuto di questa sezione, avrete notato che le procedure possono essere usate per raggiungere due obiettivi. Primo obiettivo è quello di isolare dal programma principale i passi necessari per svolgere delle azioni primitive (tracciare un quadrato, in questo esempio). Mentre considerate la sola procedura, potete concentrarvi su come questi passi combacino fra di loro, per poi ignorare questi dettagli mentre lavorate al programma principale. Mentre state lavorando al programma principale è sufficiente ricordare in modo approssimativo come funziona la procedura. Può essere molto utile assegnare un nome alla procedura, cioè un <identificatore> che serva come riferimento diretto per ricordare ciò che la procedura dovrebbe fare quando viene chiamata.

Secondo obiettivo è che la procedura riduce sensibilmente il lavoro di scrittura. Nel programma tipo QUADRATI, la scrittura dei passi necessari per tracciare un quadrato è stata fatta una sola volta, ma è servita in 4 diverse occasioni, ogni volta con la semplice chiamata della procedura con un'istruzione di una linea.

4. Chiamata di una procedura dall'interno di un'altra procedura

Può essere spesso utile poter chiamare una procedura dall'interno di un'altra. La cosa più importante da ricordare in questo caso è che il nome della procedura chiamata deve già essere noto al compilatore nel momento stesso della chiamata. Il programma tipo PROCDEMO vi dovrebbe aiutare a capirne il funzionamento. Ecco qui di seguito ciò che il programma visualizza:

```
INIZIO PROGRAMMA PRINCIPALE
PROC-P
```

```
RITORNO DA P
INIZIO PROC-Q
PROC-P
FINE PROC-Q
FINE PROGRAMMA PRINCIPALE
```

L'obiettivo principale di questo programma è quello di illustrare l'ordine di esecuzione quando una procedura ne chiama un'altra, ed è paragonabile alla chiamata dell'ultima procedura dal programma principale, come nell'esercizio precedente. In questo programma le istruzioni WRITELN procurano i mezzi per "*tracciare*" l'ordine di esecuzione. Qualora foste in dubbio sull'ordine di esecuzione seguito da uno dei vostri programmi, sarebbe una buona idea quella di aggiungere delle istruzioni WRITELN che, visualizzando dei brevi messaggi vi informano sullo svolgimento del programma in punti strategici. Dopo che il programma sarà stato completamente corretto, potrete togliere queste istruzioni di traccia per ottenere il risultato finale.

Notate che la chiamata di P nella seconda linea del programma principale funziona allo stesso modo della chiamata della procedura UNQUADRATO della precedente sezione. Questo spiega la prima linea visualizzata con la legenda "PROC-P". Successivamente viene chiamata la procedura Q che visualizza una linea attestante l'inizio della procedura stessa. In linea 11,P è chiamata di nuovo, dall'interno della procedura Q. Di nuovo P elabora la sua unica linea eseguibile e la legenda "PROC-P" riappare. Questa volta, quando P ha terminato, il controllo torna alla linea 12, la linea successiva a quella in cui P è stata chiamata. Ora viene eseguita l'istruzione in linea 12 con l'annuncio che la procedura Q sta finendo.

```
 1: PROGRAMMA PROCDEMO;
 2:
 3: PROCEDURE P;
 4: BEGIN
 5:    WRITELN('PROC-P');
 6: END (*P*);
 7:
 8: PROCEDURE Q;
 9: BEGIN
10:    WRITELN('INIZIO PROC-Q');
11:    P;
12:    WRITELN('FINE PROC-Q');
13: END (*Q*);
14:
15: BEGIN (*PROGRAMMA PRINCIPALE*)
16:    WRITELN('INIZIO PROGRAMMA PRINCIPALE');
17:    P;
```

```
18:    WRITELN('RITORNO DA P');
19:    Q;
20:    WRITELN('FINE PROGRAMMA PRINCIPALE');
21: END.
```

Infine Q termina lasciando, in linea 20 un'ultima istruzione che deve ancora essere elaborata dal programma principale.

Un punto importante da notare è che l'ordine di apparizione delle procedure P e Q non può essere invertito nel programma PROCDEMO senza apportare degli altri cambiamenti. Qualora fossero invertiti, la chiamata di P, che appare all'interno di Q, avverrebbe prima che il compilatore abbia constatato la dichiarazione di "P" quale nome di un'altra procedura. Tutto ciò provocherebbe confusione nel compilatore ed un messaggio di errore di sintassi verrebbe visualizzato avvisandovi che l'identificatore "P" non è dichiarato.

ESERCIZIO 2.1:
Fate riferimento alla figura 2-2. Scrivete e provate un programma che disegni queste figure. Suggerimento: scrivete dapprima una procedura TRIANGOLO che tracci una delle suddette figure, chiamando la procedura TRIANGOLO per sei volte. Fra ogni chiamata di TRIANGOLO sarà necessario spostare la tartaruga di 60 gradi. Prima di completare il programma è senz'altro utile assicurarsi che la procedura ESAGONO funzioni come previsto! Ora potete disegnare i tre esagoni scrivendo il resto del programma principale, muovendo la tartaruga in diversi punti dello schermo prima di ogni chiamata di ESAGONO.

5. Parametri

Vi sarete forse accorti che le figure disegnate potrebbero essere più interessanti qualora fosse possibile comunicare alla procedura le dimensioni desiderate per ogni figura primitiva. Ciò può essere fatto con un "*parametro*", come viene illustrato nel programma tipo GRAFPROCS. Esistono due tipi di parametri, ma questa è una complicazione che rimandiamo al capitolo 4. GRAFPROCS disegna le figure illustrate nella figura 2-3.

Come primo passo nella comprensione di questo programma, concentratevi sulla procedura QUADRATO, dalla linea 14 alla linea 26, che è molto simile alla procedura UNQUADRATO usata precedentemente nel programma QUADRATI. La differenza è che la lunghezza dei lati del quadrato, specificata in ogni istruzione MOVE (linee 17,

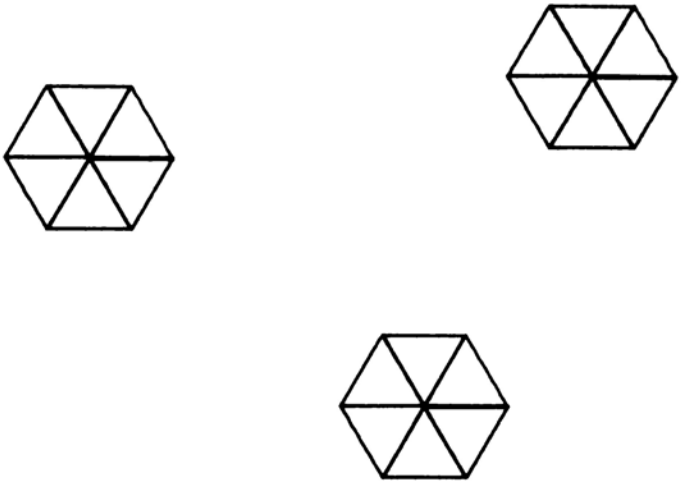


Figura 2-2

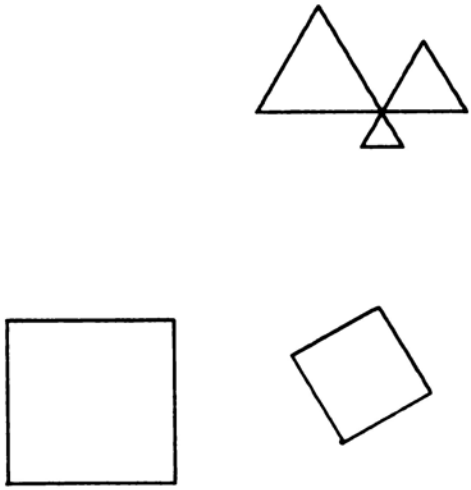


Figura 2-3

19, 21, 23), è data dall'identificatore DIMENSIONE non da un numero. DIMENSIONE si riferisce ad una locazione di memoria dove è stato precedentemente inserito un valore numerico.

Per capire come i valori numerici vengano scritti in DIMENSIONE, notate che DIMENSIONE viene menzionato anche nella prima linea della procedura, la linea 14. L'inserimento della notazione "DIMENSIONE:INTEGER" fra parentesi tonde, comunica al compilatore che l' < identificatore > DIMENSIONE sarà usato come parametro. La dicitura ":INTEGER" dice al compilatore che quel parametro contiene dei valori del "*tipo*" INTEGER. Questo significa che il valore deve essere un numero intero (non una frazione o un decimale). Sui microelaboratori da voi probabilmente usati, un valore intero può essere un qualsiasi numero compreso fra −32767 e +32767, compresi −1, 0, 1, 2, 3,... e così via. Su macchine più grandi il valore intero maggiore e minore potrà includere più cifre.

```
 1: PROGRAMMA GRAFPROCS;
 2:
 3: PROCEDURE TRIANGOLO(DIMENSIONE:INTEGER);
 4: BEGIN
 5:    PENCOLOR(WHITE);
 6:    MOVE(DIMENSIONE);
 7:    TURN(120);
 8:    MOVE(DIMENSIONE);
 9:    TURN(120);
10:    MOVE(DIMENSIONE);
11:    TURN(120);
12:    PENCOLOR(NONE);
13: END (*TRIANGOLO*);
14: PROCEDURE QUADRATO(DIMENSIONE:INTEGER);
15: BEGIN
16:    PENCOLOR(WHITE);
17:    MOVE(DIMENSIONE);
18:    TURN(90);
19:    MOVE(DIMENSIONE);
20:    TURN(90);
21:    MOVE(DIMENSIONE);
22:    TURN(90);
23:    MOVE(DIMENSIONE);
24:    TURN(90);
25:    PENCOLOR(NONE);
26: END (*QUADRATO*);
27: BEGIN (*PROGRAMMA PRINCIPALE*)
28: (*NOTA: I parametri si riferiscono al terminale Tektronix*)
```

```
29: (*4006. Per gli altri saranno necessari cambiamenti*)
30:    MOVETO(200,200);
31:    TRIANGOLO(100);
32:    TURN(120);
33:    TRIANGOLO(150);
34:    TURN(120);
35:    TRIANGOLO(50)
36:    MOVETO(-250,-250);
37:    TURNTO(0);
38:    QUADRATO(200);
39:    MOVETO (150,-200);
40:    TURN(30);
41:    QUADRATO(120);
42: END.
```

Fate ora riferimento alla linea 38 dove la procedura QUADRATO è chiamata per la prima volta dal programma principale. La *"costante intera"*, cioè il numero intero, 200 dice al compilatore di sistemare il programma in modo che il valore 200 sia assegnato a DIMENSIONE quando QUADRATO inizia ad essere eseguita. Questo valore sarà usato invece di DIMENSIONE all'interno della procedura, in ciascuna delle 4 istruzioni MOVE.

Notate ora che la chiamata di QUADRATO in linea 41 usa un valore differente, cioè 120. Questo sarà il nuovo valore che sostituirà DIMENSIONE in ciascuna delle istruzioni MOVE all'interno della procedura, quando questa viene eseguita. Ecco la spiegazione per le differenti dimensioni dei due quadrati disegnati nella figura 2-3. Allo stesso modo viene usato un parametro, chiamato DIMENSIONE, nella procedura TRIANGOLO. TRIANGOLO è chiamato per 3 volte, ogni volta con un parametro di valore differente. Notate infatti, che i tre triangoli tracciati nella illustrazione sono di tre dimensioni diverse.

Abbiamo sin qui usato lo stesso termine "parametro" sia quando una procedura veniva dichiarata, come nelle linee 3 e 14 di GRAFPROCS, sia quando veniva chiamata, linee 31 e 38, ma può essere utile distinguerli fra di loro a seconda dell'uso. Nelle linee 3 e 14, DIMENSIONE rappresenta un *"parametro formale"* poichè il valore che rappresenta è sostituito da un identificatore. Nelle linee 31 e 38 (così come nelle 33, 35 e 41), i valori *"passati"* alla procedura sono chiamati *"parametri effettivi"*, perchè sostituiscono il parametro formale durante l'esecuzione della procedura.

6. Sintassi per procedure

L'illustrazione 2-4 mostra la sintassi di <blocco>, riveduta considerando tutto ciò

che è stato finora fatto sulle procedure. Il riferimento a <identificatore tipo> in <elenco parametri> è sviluppato nella figura 2-5. Definiremo il significato del tipo STRING e tipo CHAR nelle sezioni successive.

Notate che la sintassi di <blocco> richiede che tutte le procedure siano dichiarate prima della parte BEGIN...END, che costituisce la parte principale del <blocco>. Notate inoltre che il riferimento a <blocco>, quale parte della dichiarazione di una procedura significa che è possibile dichiarare una procedura all'interno di un'altra procedura. Quest'ultimo punto verrà approfondito nel capitolo 4.

La sintassi mostra che all'interno di un <elenco parametri> si possono elencare diversi identificatori dello stesso <tipo>, separati da una virgola (","). Ciascun identificatore rappresenta il nome di un parametro. Potremmo avere anche tipi diversi di parametri dichiarati nella stessa linea di *"intestazione"* di una procedura. Svilupperemo quest'idea dopo aver descritto l'uso dei tipi STRING e CHAR nelle prossime sezioni. Notate ancora che la sintassi permette un elenco di parametri *"vuoto"*, mancante addirittura delle parentesi. È questo il caso della procedura usata nel programma QUADRATI, dove non viene usato nessun parametro.

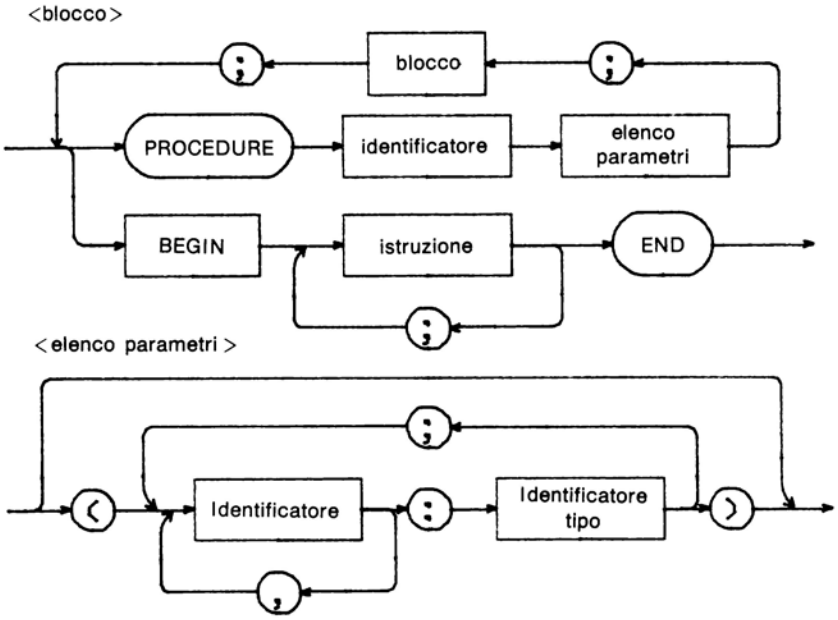
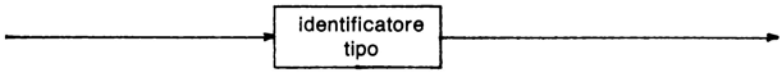


Figura 2-4

<tipo>



<identificatore tipo>

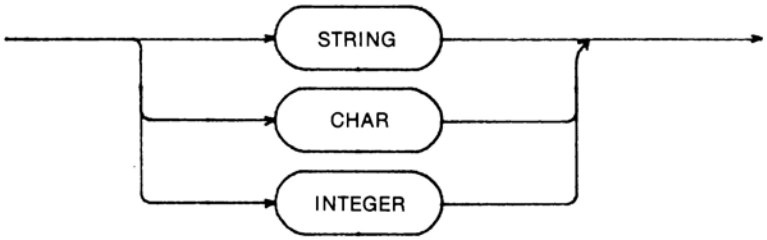


Figura 2-5

ESERCIZIO 2.2:
Verificate la vostra abilità nel trarre delle conclusioni generali dalle carte sintattiche, trasformando il programma dell'esercizio 2.1 nel modo seguente: scrivete il programma in modo che disegni ciascuna delle tre figure esagonali assegnando un valore diverso alla lunghezza del lato di ogni triangolo. Invece di usare le istruzioni MOVE o MOVETO per spostare la tartaruga nel punto di inizio di ciascun esagono, nell'ambito del programma principale, dichiarate ed usate due parametri addizionali all'interno della procedura ESAGONO, che definiscano il punto in cui deve essere tracciato l'esagono. Le relative istruzioni MOVE e MOVETO, dovranno, in questo caso, essere inserite all'interno della procedura invece che nel programma principale. I due parametri definenti il punto di inizio potrebbero essere, sia il raggio che va dal centro del video ad un angolo, sia le distanze verticali ed orizzontali dal centro del video. Il risultato di queste modifiche dovrebbe essere un programma più breve di quello scritto nell'esercizio 2.1, in quanto i vari posizionamenti nel punto di inizio delle figure saranno fatti con l'uso di parametri.

7. Variabili

Una *"variabile"* è il nome di una locazione di memoria dove il valore di un dato (o in alcuni casi valori di un gruppo di dati associati) può essere memorizzato per un uso successivo. Una variabile deve essere dichiarata al fine di poterla associare a qualche < tipo > di dato. Sotto questo aspetto una variabile è simile, concettualmente, ad un parametro, ma differisce da quest'ultimo in quanto non le viene automaticamente assegnato un valore quando una procedura è chiamata. Ogni locazione attribuita ad una variabile semplice, come quelle usate in questa sezione, ha spazio per memorizzare un solo valore alla volta. Potete paragonare una variabile ad una buca delle lettere che possa contenere una sola lettera alla volta.

Per una semplice illustrazione sull'uso di una variabile, considerate il programma STELLE, che visualizza le figure riportate nella figura 2-6. In questo programma la variabile SCALA è dichiarata, in linea 2, essere il tipo INTEGER. Così come per un parametro di tipo INTEGER, ciò significa che SCALA può essere usata per memorizzare un numero intero compreso fra –32767 e +32767, nel caso di un microelaboratore (la scelta può essere maggiore su macchine più grandi).

Nel programma STELLE, alla variabile SCALA è *"assegnato"* il valore 30, in linea 24. Il simbolo (":=") indica che qualsiasi valore che appare alla destra del simbolo deve essere assegnato alla variabile posta alla sinistra del simbolo stesso. La linea 24 è un esempio di *"istruzione di assegnamento"*. Da qui in poi, a meno che non venga assegnato un altro valore in seguito, l'identificatore SCALA avrà valore 30 in qualsiasi punto appaia, per esempio nelle linee 28, 29, 33, 34, 38 e 39. Se un nuovo valore venisse assegnato successivamente, quel valore sostituirebbe quello iniziale di 30, poichè una variabile può contenere un solo valore alla volta.

Ogni volta che un'istruzione di assegnamento viene eseguita, il valore assegnato alla variabile sostituisce quello precedentemente memorizzato in quella variabile, ed il vecchio valore non è più riutilizzabile.

In questo esempio SCALA è usata per specificare di quante unità deve essere lunga ogni linea disegnata dal programma. Il valore 30 usato nel programma stampato in questo libro, è adatto per visualizzare figure sul terminale grafico 4006 della Tektronix. Nel caso in cui state usando dei terminali diversi, a SCALA potrà essere assegnato un valore differente. Per esempio il valore appropriato sul microelaboratore Terak 8510A, sarebbe 10 invece di 30 per ottenere delle figure con le stesse dimensioni di quelle visualizzate sul video Tektronix. Questa differenza è dovuta al fatto che il numero di punti per centimetro (o per pollice) necessario per simulare una linea, come nella figura 0-1, varia da un terminale all'altro.

Se il valore di SCALA fosse interessante in sè stesso, allora potrebbe apparire da

solo quale parametro effettivo, o in altri contesti. Nel programma STELLE usiamo SCALA per cambiare il valore di una *"costante intera"* (cioè un numero intero dato esplicitamente) al fine di controllare la lunghezza della linea specificata da quel numero. Per esempio, in linea 28, MOVE dovrebbe coprire 30 volte 8, cioè 240 unità. L'asterisco ("*") indica che SCALA dovrebbe essere moltiplicato per 8. Questo è un esempio di *"espressione aritmetica"*, un argomento che verrà ampliato in seguito in questo capitolo.

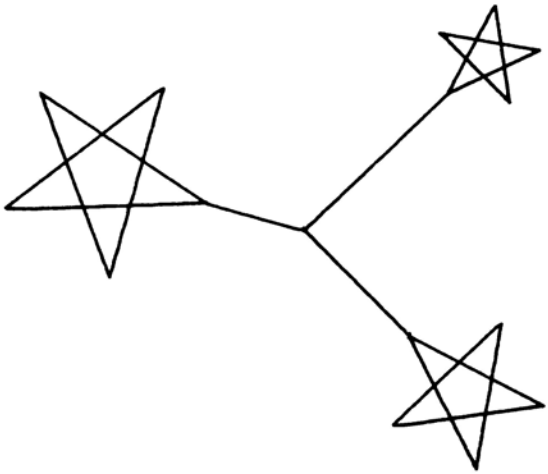


Figura 2-6

```
 1: PROGRAMMA STELLE;
 2: VAR SCALA:INTEGER;
 3:
 4: PROCEDURE STELLA(DIMENSIONE:INTEGER);
 5: BEGIN
 6:    TURN(-18); (*CENTRA LA STELLA SUL RAMO*)
 7:    PENCOLOR(WHITE);
 8:    MOVE(DIMENSIONE);
 9:    TURN(144);
10:    MOVE(DIMENSIONE);
11:    TURN(144);
12:    MOVE(DIMENSIONE);
13:    TURN(144);
14:    MOVE(DIMENSIONE);
15:    TURN(144);
```

```
16:    MOVE(DIMENSIONE);
17:    TURN(144);
18:    PENCOLOR(NONE);
19:    TURN(18);
20:    (*RIPORTA TARTARUGA NELLA DIREZIONE ORIGINALE*)
21: END (*STELLA*);
22:
23: BEGIN (*PROGRAMMA PRINCIPALE*)
24:    SCALA:=30; (*30 PER TEKTRONIX 4006,*)
25:           (*USA 10 PER TERAK, ?? PER ALTRI*)
26:    PENCOLOR(WHITE);
27:    TURN(45);
28:    MOVE(SCALA*8);
29:    STELLA(SCALA*4);
30:    MOVETO(0,0);
31:    TURNTO(165);
32:    PENCOLOR(WHITE);
33:    MOVE(SCALA*4);
34:    STELLA(SCALA*8);
35:    MOVETO(0,0);
36:    TURN(150)
37:    PENCOLOR(WHITE);
38:    MOVE(SCALA*6);
39:    STELLA(SCALA*6);
40: END.
```

8. Sintassi per variabili

Come primo passo nella comprensione della sintassi per variabili, considerate la figura 2-7, illustrante la sintassi ampliata per < blocco >. Notate che un blocco può iniziare con una sequenza di dichiarazioni di variabili introdotte dall'identificatore riservato "VAR", che appare solo una volta. Tutte le variabili comprese in un blocco devono essere dichiarate prima che venga dichiarata la prima procedura del blocco stesso. Come in un elenco di parametri, potete dichiarare che tutti gli identificatori di un elenco separati da virgole devono essere associati ad un unico < tipo >.

Se dichiarate una variabile dopo VAR all'interno di un < blocco > che fa parte della dichiarazione di una procedura, quella variabile è detta essere *"locale"* rispetto a quel blocco e non può essere fatta oggetto di riferimento da parte del programma principale o di qualsiasi altra procedura. Discuteremo più dettagliatamente di questo punto nel capitolo 4. Una variabile dichiarata nel < blocco > del programma principa-

le è detta essere *"globale"* e può costituire oggetto di riferimento da parte sia del programma principale che di qualsiasi procedura. L'unico caso in cui una variabile globale non può essere usata all'interno di una procedura avviene quando una variabile con lo stesso nome è dichiarata come locale in questa procedura.

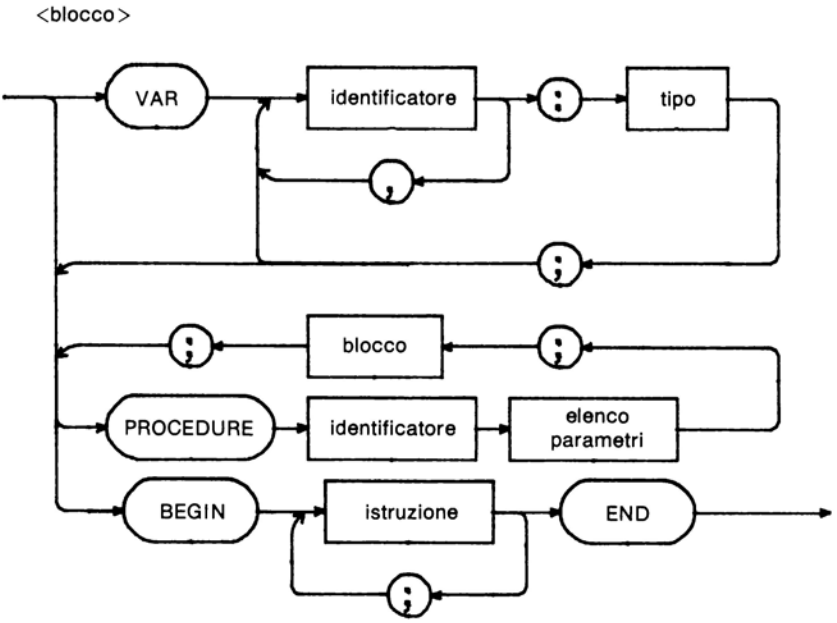
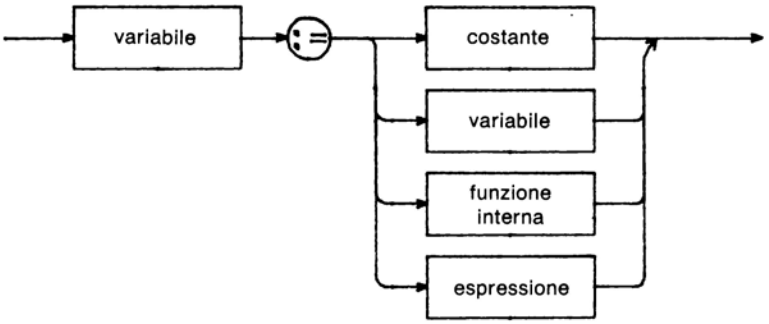


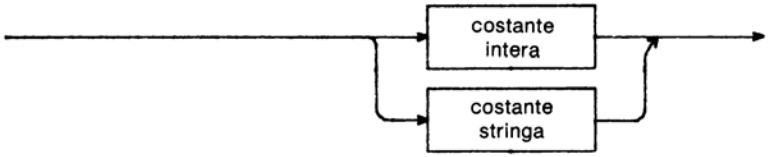
Figura 2-7

La figura 2-8 mostra la sintassi per < istruzione di assegnamento > in forma generale. Ciò che non risulta dal diagramma è che l'entità posta al lato destro dell'*"assegnamento"* (": =") deve essere dello stesso tipo della variabile posta al lato sinistro. Questa restrizione sarà leggermente allentata nel capitolo 5 dove cominceremo ad operare con numeri di due tipi diversi. Provvisoriamente dovrete presumere che ogni oggetto contenuto nella sintassi dell'< istruzione di assegnamento > sia preceduto da *"intero"*. Commenteremo la sintassi per < espressione > nelle successive sezioni.

<istruzione assegnamento>



<costante>



<costante intera>

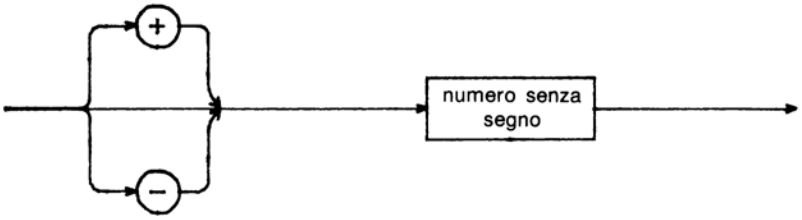


Figura 2-8

9. Operare con variabili STRING

Fin qui il nostro lavoro con stringhe di caratteri si è limitato alla stesura di valori di < costanti stringhe >, come descritto nel capitolo 1. Ora dopo aver introdotto il concetto di < variabile > possiamo iniziare ad operare con variabili che memorizzano sia stringhe intere che un solo carattere. Prima di mostrarvi come si possano usare stringhe per compiere delle interessanti elaborazioni, è necessario preparare il campo mostrando il rapporto esistente fra la variabile di tipo CHAR e quella di tipo STRING. Il programma campione PUNTA fornisce l'esempio da cui possiamo partire. Questo programma dovrebbe visualizzare le seguenti linee:

```
 1: PROGRAMMA PUNTA;
 2: (*puntatore a caratteri specifici in una stringa*)
 3: VAR S:STRING;
 4:     CH:CHAR;
 5:         I:INTEGER;
 6: BEGIN
 7:     S:='ANCHE SE MI VIDE';
 8:     WRITELN(S); (*traccia*)
 9:     CH:=S[7]; (*7mo carattere di S*)
10:     WRITELN(CH);
11:     I:=11;
12:     WRITELN(S[I],S[11]);
13:                 (*sarebbe il caso fossero lo stesso!*)
14:     CH:='X';
15:     S[4]:=CH;
16:     WRITELN(S); (*traccia successiva al cambiamento*)
17:     CH:=S[I-3];
18:     WRITELN('S[',I-3,']=',CH);
19: END.


ANCHE SE MI VIDE
S
II
ANCHE SE MI VIDE
S[8]=E
```

La variabile CH può avere come valore un qualsiasi carattere singolo che possa essere visualizzato. (Una variabile di tipo CHAR può anche contenere un carattere che non può essere visualizzato. A meno che non siate molto coraggiosi o molto intelligenti vi consigliamo di evitare per ora questa situazione). In linea 14, alla variabile CH è assegnato il valore 'X', cioè la singola lettera "X". La costante posta al lato

destro in linea 14 è una costante stringa rappresentata da un unico carattere. A CH deve essere assegnato un valore di < tipo > CHAR.

Alla variabile STRING S è assegnato il valore di una < costante stringa > in linea 7. Una variabile STRING può contenere un valore variabile da 0 a 80 caratteri. Mezzi sono disponibili per cambiare il *"difetto"* (default), cioè il valore presunto del numero massimo di caratteri di una variabile stringa con qualche altro valore. Approfondiremo questo punto in seguito.

Ogni carattere memorizzato in una variabile STRING è, esso stesso una variabile di < tipo > CHAR. Questo ci permette il riferimento ad un carattere specifico nella variabile S, come in linea 9. Il numero che seleziona il carattere desiderato all'interno della variabile STRING, segue l'identificatore della variabile stessa ed è posto fra parentesi quadre. Il primo carattere memorizzato nella STRINGA è 1, cioè nel nostro esempio il valore memorizzato in S[1] è 'A'. Nello stesso modo "S[7]" si riferisce al settimo carattere in S. L'entità messa fra parentesi deve essere del < tipo > INTEGER, non è però necessario che sia una costante intera quale "7". In linea 11 assegnamo alla variabile INTEGER 1 il valore 11. Poi in linea 12 dimostriamo che S[1] e S[11] si riferiscono allo stesso carattere in S, poichè in questa linea il valore di 1 rimane uguale a 11.

Come con le altre variabili, potete usare un carattere memorizzato in una locazione specifica nella variabile STRING, come pure assegnare un nuovo valore a quella locazione. In linea 15, il valore attualmente memorizzato in CH viene assegnato alla locazione 4 in S. L'istruzione WRITELN, usata quale traccia in linea 16, mostra che il quarto carattere è stato cambiato in 'X' da quella azione.

Nelle linee 17 e 18, illustriamo l'uso di un'< espressione aritmetica >, in questo caso "I–3". Qui si ottiene che il valore 3 venga sottratto dal valore di I producendo in questo caso 8, e che il risultante valore intero venga poi usato. In linea 17, questo valore fa riferimento alla locazione 8 nella STRINGA S. In linea 18, il valore dell'espressione è interamente stampato. Notato che due < costanti > quotate sono usate per far sì che la linea visualizzata appaia quale riferimento a "S[8]=".

10. Premesse sulle espressioni aritmetiche

In questo e nei prossimi due capitoli avremo l'opportunità di usare delle < espressioni aritmetiche > semplici. Considerando che le espressioni aritmetiche forniscono dei mezzi di manipolazione di numeri con l'elaboratore, preferiamo rimandare al capitolo 5 delle considerazioni più dettagliate su questo argomento. Le premesse fatte in questa sezione dovrebbero essere sufficienti per permettervi di capire l'uso di semplici espressioni aritmetiche negli esempi incontrati prima del capitolo 5.

Le figure 2-9 e 2-10, illustrano le sintassi semplificate per <espressione aritmetica> e per i suoi componenti. Ignorate il riquadro con l'indicazione <funzione interna intera> fino alla prossima sezione. La sintassi mostra che in ciascuna delle posizioni di una <espressione aritmetica> è possibile usare una costante intera, una variabile intera o un <termine> e ogni posizione è separata dalla successiva da "+" o "–". L'*"operatore"* "+" significa somma delle entità poste ai due lati del simbolo, mentre "–" significa sottrazione della seconda entità dalla prima. In un'<espressione aritmetica> i simboli "+" e "–" sono facoltativi davanti al primo valore. Quando sono presenti implicano che la costante intera 0 sia alla sinistra del simbolo. Perciò "–10" è equivalente a "0–10". Gli spazi fra simboli successivi sono facoltativi, ma non lo sono all'interno di un identificatore o di una costante intera. L'entità chiamata <termine> in sintassi è un elemento da usare abbinato al segno della moltiplicazione "*" (e più tardi con i simboli che significano dividere). La sintassi mostra che un termine è costituito da uno, due o più <fattori>. Se sono presenti più fattori, ognuno deve essere separato dall'altro dal simbolo "*". In tal modo "A*B" rappresenta un <termine> in cui la variabile A deve essere moltiplicata per la variabile B. Questa moltiplicazione è necessaria per l'ottenimento di un valore del <termine>. L'effetto è che nell'espressione:

A*B+C*D–E

il calcolo inizia eseguendo dapprima le moltiplicazioni. Solo dopo aver completato l'operazione di moltiplicazione si può passare all'addizione, seguita dalla sottrazione. L'elaborazione procede da sinistra verso destra, manipolando gli operatori "+" e "–" all'interno di un'<espressione aritmetica> e manipolando i successivi operatori "*" all'interno di un <termine>. Si dice che l'operatore della moltiplicazione "*" ha *"precedenza"* sugli operatori dell'addizione e della sottrazione in quanto la moltiplicazione viene prima. In altre parole, in un'<espressione aritmetica> l'operazione di moltiplicazione deve precedere quella di addizione e sottrazione.

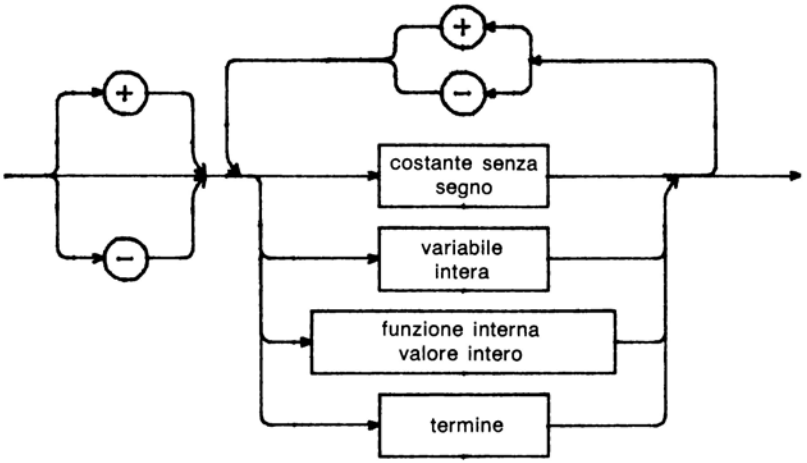
Notate che nella definizione di <fattore> c'è un espediente che vi permette di eseguire l'addizione e la sottrazione prima della moltiplicazione: l'uso di parentesi tonde. Quando racchiudete parte di una espressione fra parentesi tonde, essa verrà risolta prima della parte posta fuori dalle parentesi. Per vederne il funzionamento considerate il seguente esempio:

Se A ha valore 1, B=2 e C=3 allora

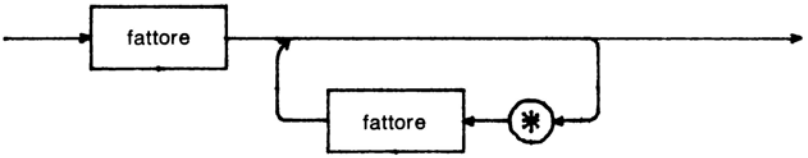
A + B*C ha valore 7
(A + B) *C ha valore 9

Nel primo caso B è moltiplicato per C prima di essere aggiunto ad A, seguendo la

<espressione aritmetica>



<termine>



<fattore>

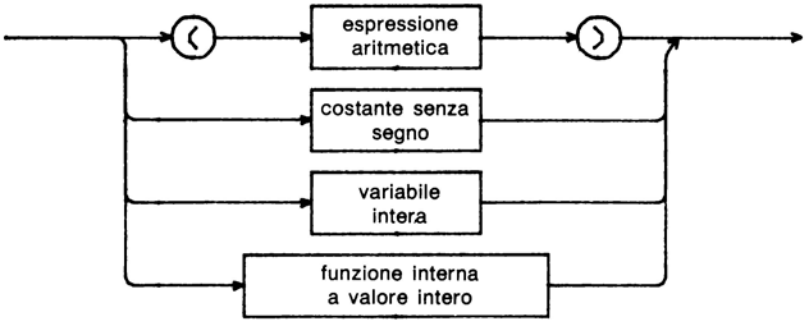


Figura 2-9

<variabile>

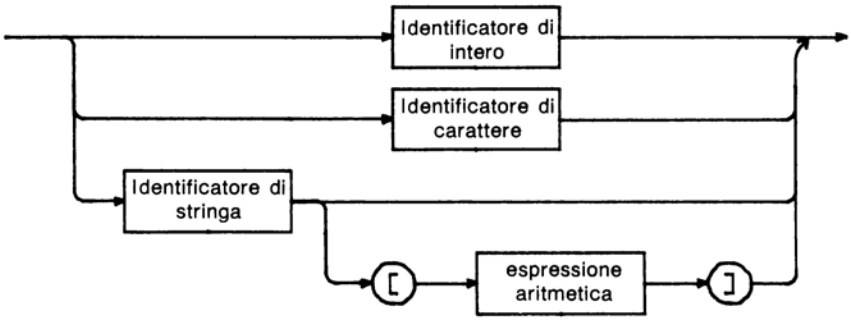


Figura 2-10

regola della precedenza. Nel secondo caso viene prima trovato il valore di A aggiunto a B perchè posti fra parentesi, poi il risultato viene moltiplicato per C.

ESERCIZIO 2.3:
Il disegno riportato nella figura 2-11, può essere visualizzato da un programma principale contenente le seguenti quattro istruzioni e da un altro programma che definisca il valore della dimensione o scala del disegno in unità video.

```
RETT(5,5);
RETT(10,3);
RETT(3,10);
RETT(2,12);
```

RETT è una procedura che disegna un rettangolo la cui altezza è definita dal primo parametro, e la cui lunghezza è data dal secondo parametro.

Scrivete e provate un programma completo, includente la procedura RETT, che disegni questa figura. Assicuratevi che valori diversi del fattore dimensione diano figure di differenti dimensioni, ma aventi lo stesso aspetto. In altre parole, una variabile DIMENSIONE dovrebbe controllare la dimensione di ogni parte dell'intero disegno. Dovrebbe bastare l'alterazione di un'unica istruzione per cambiare la dimensione dell'intera figura.

11. Procedure e funzioni interne per stringhe

Una *"funzione"* (in gergo) è una procedura di tipo speciale. Una funzione viene chiamata collocando in forma di espressione, il suo identificatore più un elenco di parametri effettivi. Quando, procedendo nella risoluzione dell'espressione, si raggiunge l'identificatore della funzione, essa viene messa in esecuzione. Terminata l'esecuzione della funzione, il suo identificatore sarà sostituito da un valore e la risoluzione dell'espressione verrà ripresa. Si dice che una funzione *"restituisce"* un valore da usare nell'espressione. Il < tipo > di valore restituito da una funzione dipende dal modo in cui la funzione è dichiarata e dal modo con cui svolge le sue elaborazioni. Il capitolo 4 include una descrizione del modo con cui potete dichiarare le vostre funzioni e procedure.

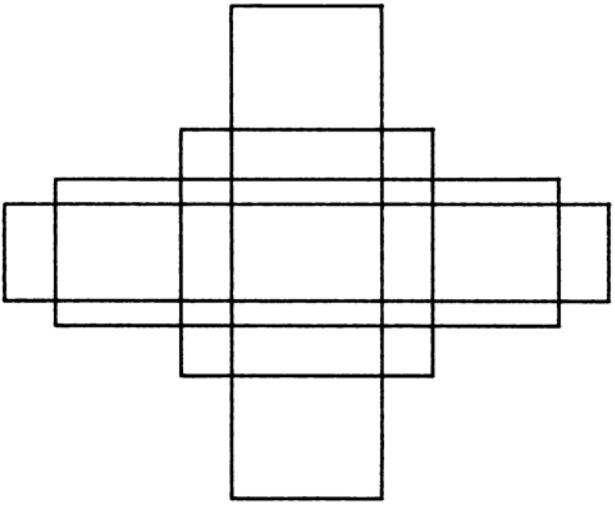
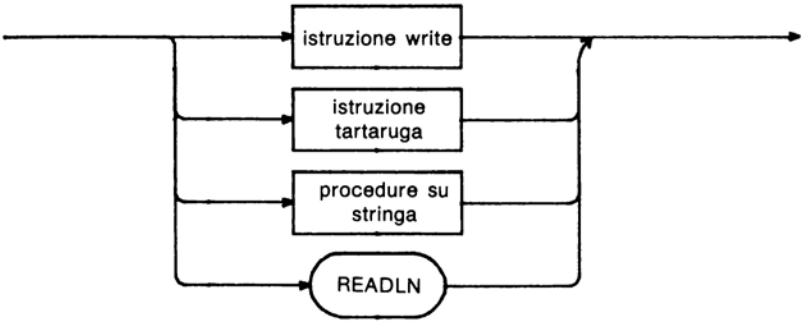


Figura 2-11

Una comodità per i programmatori è rappresentata dalla possibilità fornita dal compilatore PASCAL, di chiamare delle procedure e funzioni progettate come parte del sistema per cui non è necessario dichiararle, sono cioè *"interne"*. Conoscete già le procedure interne MOVE, TURN, MOVETO, PENCOLOR e CLEARSCREEN, ognuna delle quali è stata progettata per lavorare con grafici tartaruga. In questa sezione introduciamo due procedure e quattro funzioni interne, ideate per lavorare con variabili STRING. Troverete le sintassi relative illustrate in figura 2-12, 2-13 e 2-14.

<procedure interne>



<procedure su stringhe>

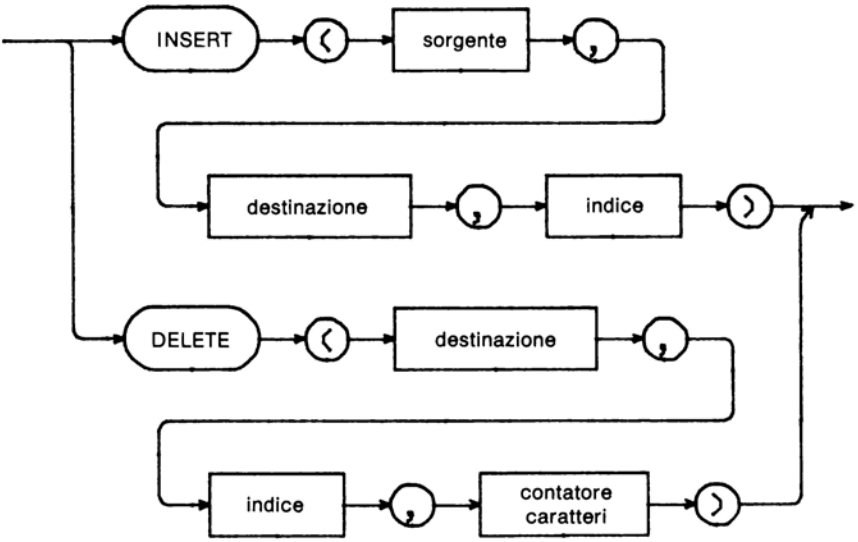
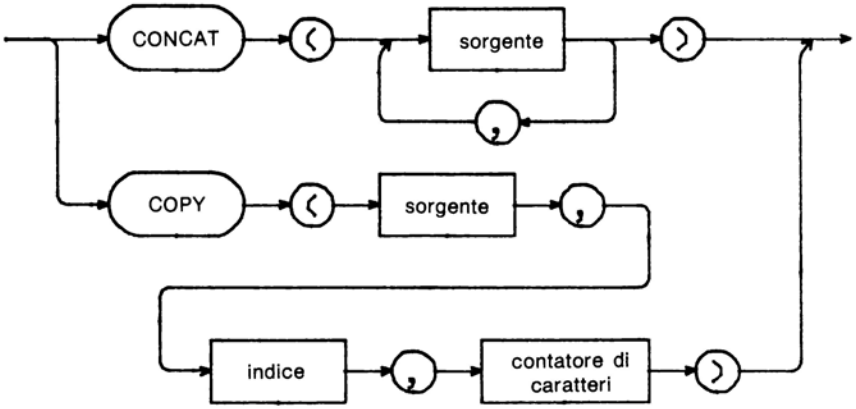


Figura 2-12

<funzioni interne>

parte (a) — con valore stringa



parte (b) — con valore intero

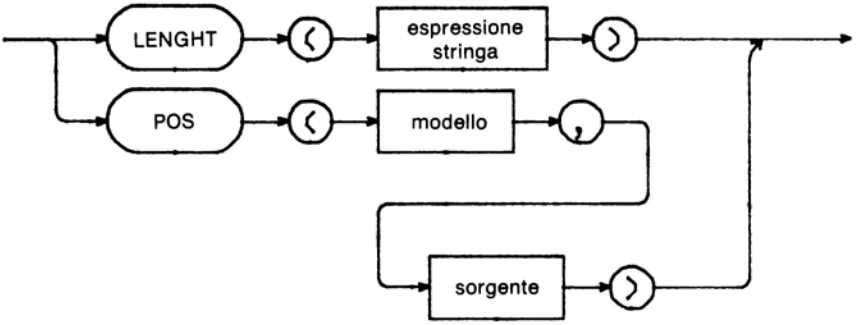


Figura 2-13

Illustriamo prima di tutto il funzionamento di ognuna di queste procedure e funzioni con il programma INTRINSECO. Mostreremo poi altri esempi con dati più interessanti. INTRINSECO visualizza le seguenti linee:

```
LA POSIZIONE DI MI EST:14
LA LUNGHEZZA DELLA CONFIGURAZIONE EST:2
PRIMA DI CANCELLARE:ANCHE SE LEI MI AMA
DOPO AVER CANCELLATO:ANCHE SE LEI AMA
DOPO L'INSERIMENTO:ANCHE SE LEI LO AMA
DOPO L'AGGANCIO:ANCHE SE LEI LO AMA IO NON CI CREDO
PRIMA COPIA:ANCHE SE LEI LO
SECONDA COPIA:AMA IO NON CI CREDO
```

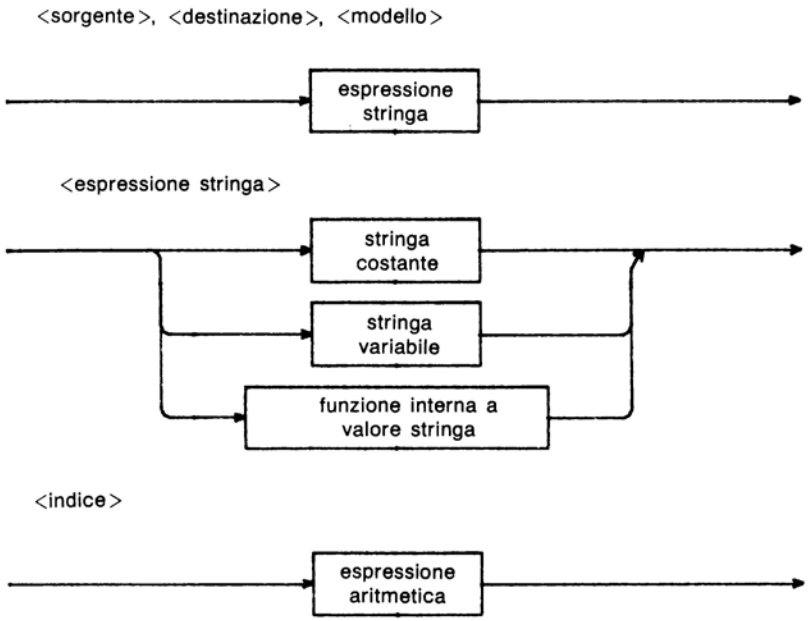


Figura 2-14

In linea 9, la funzione POS cerca una stringa, all'interno della variabile DEST, che si accordi al contenuto della variabile *"configurazione"* ELEMO. Il valore ritornato è un intero che rappresenta la posizione del primo carattere dove viene trovata la stringa uguale a quella cercata. Come potete prontamente verificare contandone i caratteri, "MI" inizia con la "M" in posizione 14 di DEST, come viene visualizzata sul vi-

deo. Se nessuna stringa identica al modello viene trovata, POS ritorna un valore 0 (zero), che non corrisponde a nessuna delle posizioni permesse in una STRING.

In linea 14, la funzione LENGTH ritorna il numero di caratteri memorizzati nella variabile STRING ELEM. Come appare sul video, e come potete verificare, "MI" contiene 2 caratteri.

In linea 21, la procedura DELETE rimuove i caratteri di LE (cioè il valore memorizzato in LE) dalla STRINGA DEST, partendo dalla locazione del carattere NB.

```
 1: PROGRAMMA INTRINSECO;
 2: (*illustra le procedure e funzioni interne*)
 3: VAR SORG,DEST,ELEM,TEMP: STRING;
 4:    LE,NB,NE: INTEGER;
 5: BEGIN
 6:    DEST:='ANCHE SE LEI MI AMA';
 7:    ELEM:='MI';
 8:    (*cerca la Posizione di ELEM in DEST*)
 9:    NB:=POS(ELEM,DEST);
10:    WRITELN('LA POSIZIONE DI ' , ELEM , 'EST:', NB);
11:              (*usato come traccia*)
12:
13:    (*ora determina il numero di caratteri da cancellare*)
14:    LE:=LENGTH(ELEM);
15:    NE:=NB+LE;
16:       (*salva la fine della configurazione in NE per un uso futuro*)
17:    WRITELN('LA LUNGHEZZA DELLA CONFIGURAZIONE EST:',LE);
18:    WRITELN('PRIMA DI CANCELLARE:',DEST);
19:
20:    (*cancella la configurazione prima di sostituire*)
21:    DELETE(DEST,NB,LE);
22:    WRITELN('DOPO AVER CANCELLATO:',DEST);
23:    INSERT('LO', DEST, NB);
24:    WRITELN('DOPO L'INSERIMENTO:',DEST);
25:
26:    (*concatena la nuova stringa alla fine di DEST*)
27:    SORG:='IO NON CI CREDO';
28:    DEST:=CONCAT(DEST,SORG);
29:    WRITELN('DOPO L'AGGANCIO:',DEST);
30:    (*ora copia l'inizio e la fine di DEST*)
31:
32:    TEMP:=COPY(DEST,1,NE-1);
33:    WRITELN('PRIMA COPIA:',TEMP);
```

```
34:    TEMP:=COPY(DEST,NE,LENGTH(DEST)-NE+1);
35:    WRITELN('SECONDA COPIA:',TEMP);
36: END.
```

DELETE riduce il numero di caratteri memorizzati nella < destinazione >. Se provate ad eseguire DELETE con un valore del < contatore caratteri > corrispondente a più caratteri di quanti ne sono attualmente memorizzati nella < destinazione > della variabile STRINGA, il vostro programma terminerà in modo abnorme e verrà visualizzato un messaggio che ne spiega il motivo.

In linea 23, la procedura INSERT apre uno spazio all'interno della variabile < destinazione > DEST partendo dal carattere NB (cioè il valore memorizzato in NB), poi inserisce dentro questo spazio la stringa 'LO'. Come è mostrato dalla sintassi in figura 2-14, la < sorgente > potrebbe essere stata una < costante >, come in questo caso, < una variabile > o potrebbe essere stata una < funzione interna > (una funzione interna che lavora su stringhe) significante CONCAT o COPY. Il risultato dell'uso di INSERT è la memorizzazione di più caratteri nella variabile < destinazione > STRING, come può essere verificato con l'uso di LENGTH.

In linea 28, la funzione CONCAT costruisce dapprima una nuova variabile STRING contenente l'attuale contenuto di DEST seguito da quello di SORG. Successivamente assegna il contenuto di questa nuova variabile, che è nascosto dal sistema, a DEST, sostituendo così il valore originalmente memorizzato in DEST. Come mostrato dalla sintassi, potete *"concatenare"* più di due stringhe fra di loro, con l'uso di CONCAT. Per esempio, se DEST, PRIMA e DOPO sono variabili STRING, allora:

```
PRIMA:='QUESTA ';
DOPO:='META ';
DEST:=CONCAT(PRIMA, 'ERA LA', DOPO);
```

DEST conterrà 'QUESTA ERA LA META'.

In linea 32, la funzione COPY ritorna un valore di < stringa > consistente dei caratteri (NE–1) di DEST partendo dalla locazione 1. Similmente, in linea 34, COPY ritorna tutti i caratteri contenuti in DEST partendo dalla locazione NE fino alla fine. Quale semplice esercizio verificate che l'espressione

```
LENGTH(DEST)-NE+1
```

produce il numero di caratteri attualmente nel valore della stringa visualizzata dopo "SECONDA COPIA".

Notate che la conseguenza dell'uso di CONCAT è che due parole sono visualizzate

senza che vi sia uno spazio di separazione, come era probabilmente inteso, cioè "A-MATO". Il programma gira, ma non produce esattamente il risultato voluto. Siete in grado di rivedere il programma in modo che possa visualizzare "AMA IO"?.

12. Programmi tipo con l'uso di stringhe

Fate ora riferimento al programma TAGLIA che visualizza le seguenti linee:

```
ANCHE SE LEI LO AMA, IO NON CI CREDO
SE LEI LO AMA, IO NON CI CREDO
LEI LO AMA, IO NON CI CREDO
LO AMA, IO NON CI CREDO
AMA, IO NON CI CREDO
IO NON CI CREDO
```

Questo programma si serve della procedura TAGLIAPAROLA per la rimozione di una parola, partendo dall'inizio della variabile stringa SOG ogni volta che TAGLIAPAROLA viene chiamata. Il primo passo da fare è di scorrere la stringa alla ricerca di uno spazio vuoto. In linea 8 tutti i caratteri, lo spazio vuoto incluso, sono cancellati. In linea 9 viene visualizzato il nuovo contenuto di SOG.

ESERCIZIO 2.4:
Riscrivete il programma TAGLIA in modo che visualizzi ogni linea successiva del valore originale di SOG con la rimozione del primo spazio vuoto. Per esempio, dopo la prima chiamata della procedura, SOG dovrebbe contenere:

ANCHE SE LEI LO AMA, IO NON CI CREDO

```
 1: PROGRAMMA TAGLIA;
 2: VAR SOG:STRING;
 3:     NSPAZI: INTEGER;
 4:
 5: PROCEDURE TAGLIAPAROLA;
 6: BEGIN
 7:   NSPAZI:=POS(' ',SOG);
 8:   DELETE(SOG,1,NSPAZI);
 9:   WRITELN(SOG);
10: END (*TAGLIAPAROLA*);
11:
12: BEGIN (*PROGRAMMA PRINCIPALE*)
```

```
13:   SOG:='ANCHE SE LEI LO AMA, IO NON CI CREDO'
14:   WRITELN(SOG);
15:   TAGLIAPAROLA;
16:   TAGLIAPAROLA;
17:   TAGLIAPAROLA;
18:   TAGLIAPAROLA;
19:   TAGLIAPAROLA;
20: END.
```

Ora rivedete il programma in modo da mettere *due spazi* per ognuno di quelli presenti nel programma originale. Così:

ANCHE SE LEI LO AMA, IO NON CI CREDO

Fate ora riferimento al programma CAMBIA; qui la procedura SOST viene usata per sostituire una stringa, il primo parametro, con un'altra stringa, il secondo parametro. Questo programma visualizza le seguenti linee:

SEI TROPPO SAGGIA
SEI TROPPO SERIA

ERI TROPPO SERIA

In linea 7, POS è usata per individuare un equivalente della stringa modello. Se viene trovato, il modello viene cancellato. Notate che il programma terminerà in modo abnorme se il modello non viene trovato. Dobbiamo rimandare al prossimo capitolo per qualsiasi considerazione su cosa fare in queste circostanze. Supponendo che il modello venga trovato e cancellato, la stringa sostitutiva SNUOVO viene inserita al suo posto. Il risultato viene quindi visualizzato.

ESERCIZIO 2.5:
Rivedete il programma CAMBIA in modo che visualizzi dapprima le due seguenti linee:

SEI TROPPO SAGGIA, ERI TROPPO SERIA
VEDO CHE SEI TROPPO SAGGIA PER ME

poi alterate ogni apparizione di "SAGGIA" sostituendolo con 'SERIA' (o 'POSATA' o 'SIGNORA', o qualsiasi altra parola a vostro piacere) e poi visualizzate il nuovo risultato. Per operare questi cambiamenti, il programma dovrebbe usare una procedura simile a SOST. Non risolvete il problema semplicemente visualizzando le due linee da voi assegnate a SOG adattando esplicitamente il suo valore in modo da ottenere il risultato desiderato.

```
 1: PROGRAMMA CAMBIO;
 2: VAR SOG:STRING;
 3:
 4: PROCEDURE SOST(ELEM,SNUOVO:STRING);
 5: VAR NP: INTEGER;
 6: BEGIN
 7:    NP:=POS(ELEM,SOG);
 8:    DELETE(SOG,NP,LENGTH(ELEM) );
 9:    INSERT(SNUOVO,SOG,NP);
10:    WRITELN(SOG)
11: END (*SOST*);
12:
13: BEGIN (*PROGRAMMA PRINCIPALE*)
14:    SOG:='SEI TROPPO SAGGIA';
15:    WRITELN(SOG);
16:    SOST('SAGGIA' , 'SERIA');
17:    WRITELN; (*una linea vuota sullo schermo*)
18:    SOST('SEI', 'ERI');
19: END.
```

Problemi

PROBLEMA 2.1:
Il programma TENTA, riprodotto qui di seguito, dovrebbe visualizzare le 4 linee seguenti:

```
        X
       XXX
      XXXXX
     XXXXXXX
```

```
PROGRAMMA TENTA
VAR: S,XS,X2; STRING;
     N; INTEGER;
BEGIN
  S='          '; (*dieci spazi bianchi*)
  XS:='X';
  X2:='XX';
  S:=CONCAT(S,XS)
  WRITELN(S);
```

```
    XS=CONCAT(X2,XS);
    INSERT(XS,S,N);
    WRITELN(S);
    XS:=CONCAT(XS,XS);
    INSERT(XS,S,N-2)
    WRITELN(S);
    XS:=CONCAT(X2,XS);
    INSERT(XS,S,N-3)
END.
```

Sfortunatamente il programma è stato preparato da uno studente che era un po' in ritardo rispetto alle conoscenze richieste fin qui, e contiene perciò alcuni errori. Come primo passo correggete gli errori sintattici; dovreste essere in grado di trovarne 9 senza ricorrere all'aiuto del compilatore. Nessuno di questi errori presenta delle ambiguità su ciò che il programma dovrebbe fare dopo la loro correzione.

Anche dopo aver corretto gli errori sintattici, il risultato visualizzato non corrisponderà esattamente a ciò che ci si aspettava, come mostrato sopra. Come secondo passo analizzate ciò che effettivamente vien fatto dal programma e scrivete ciò che visualizzerà *senza l'aiuto dell'elaboratore*. Spesso il sistema più veloce per trovare gli errori in un programma è l'uso di questo tipo di approccio.

Infine alterate il programma in modo che produca il risultato desiderato. A questo punto, se ne avete la possibilità, provate la versione riveduta del programma sull'elaboratore. Per un problema con questo livello di difficoltà, dovreste riuscire a raggiungere la correttezza del programma ancora prima di provarlo sull'elaboratore. Ritenete che sarete giudicati come "appena sufficienti" se avete fatto due tentativi, "insufficienti" nel caso di più di due tentativi!

PROBLEMA 2.2:
In alcuni programmi per editare si può visualizzare una linea di testo, che deve essere alterata, su un piccolo schermo paragonabile a quello televisivo, e simile al video da voi usato. Esistono diversi metodi per indicare il punto in cui si vuole operare un cambiamento. Un metodo è quello di dividere una linea nel punto in cui è attualmente localizzato l'indicatore logico del testo. Per esempio:

```
      LA VELOCE VOLPE ROSSA
diventa
      LA VELOCE
                VOLPE ROSSA
```

Il punto in cui la linea salta è quello in cui si vuole operare un cambiamento. Il programma riportato di seguito, serve quale illustrazione. Aggiungete a quelle già fornite, delle istruzioni e delle dichiarazioni PASCAL in modo da completare il programma. Correggete gli eventuali errori sintattici contenuti nella parte data di programma, che dovrebbe stampare la linea divisa come sopra, una volta completato.

```
PROGRAMMA TRASFERIMENTO;
VAR CIMA,FONDO,SPAZI,MOD:STRING;
BEGIN
  SPAZI:='          ';
  CIMA:="LA VELOCE VOLPE ROSSA";
  MOD:='ROSSA';
  NP:=POS(MOD,CIMA);
  FONDO:=COPY(CIMA,NP,LENGTH(TOP)-NP+1);
```

PROBLEMA 2.3:
Tipici nomi italiani sono costruiti nella seguente forma:

<nome> <SP> <cognome>

Per esempio Mario Rossi. Notate che questa è in effetti una semplice regola sintattica. La voce <SP> rappresenta uno spazio bianco. Spesso è necessario ordinare i nominativi mettendo prima il cognome poi il nome, per esempio nel caso di un elenco in ordine alfabetico, si avrà quindi la forma:

<cognome>, <SP> <nome>

come in Rossi, Mario. Il programma sotto iniziato, è stato progettato per riordinare qualsiasi nominativo dalla prima alla seconda forma. L'istruzione assegnante un valore a SOURCE sostituisce la lettura di un qualsiasi nome dalla tastiera o da altri dispositivi esterni. (Descriveremo come fare questo nel capitolo 7). Correggete eventuali errori sintattici e scrivete il resto del programma in modo da lasciare in DEST la forma finale, desiderata del nome. Il programma dovrebbe poi visualizzare il valore lasciato in DEST.

```
PROGRAMMA ULTIMOPRIMO;
VAR SORGENTE,DEST,CONFIG:STRING;
  NL:INTEGER;
```

```
BEGIN
  SORGENTE:='MARIA G. DOLCE';
            (*fatelo funzionare per qualsiasi nome!*)
  CONFIG:='.';
  NL:=POS(CONFIG,SORGENTE);
  DEST:=SORGENTE;
```

PROBLEMA 2.4:
Con parole vostre, rispondete brevemente alle seguenti domande:

Che cosa è una procedura?

Descrivete tre ragioni per cui si usano delle procedure in un programma progettato per uno scopo qualsiasi.

Che cosa è un parametro e come viene usato?

Che cosa è una variabile e come viene usata?

Che cosa è un <tipo>? Nominate tre <tipi> usati in questo capitolo.

A che scopo viene usata un'<espressione aritmetica>?

PROBLEMA 2.5:
Supponete che le seguenti dichiarazioni di variabili facciano parte dell'intestazione di un programma:

```
VAR I,J:INTEGER;
    CH1,CH2:CHAR;
    SA,SB:STRING;
```

Indicate quali delle seguenti istruzioni violano le regole PASCAL e quali sono invece "lecite":

```
I:=1;
J:=I;
J:='3';
I:=LENGTH(SA);
I:='1234';
CH1:=1;
CH1:='Q';
```

```
CH2:=SA[J];
CH2:=27;
CH1:=CH2;
SA:='qualsiasi stringa';
SA:=5;
SB:=CH2;
SA:=I;
SB:=SA;
SB:=CONCAT(SB,SA);
SA:='R';
```

Indicate l'errore per ognuna delle istruzioni da voi definite non lecite.

CAPITOLO 3

CONTROLLO DELL'EVOLUZIONE DEI PROGRAMMI, RIPETIZIONE

1. Obiettivi

In questo capitolo imparerete a scrivere un programma che possa ripetere più volte delle azioni specifiche, e che possa decidere se certe azioni verranno eseguite o meno in dipendenza dei valori di determinati valori dei dati.

1a. Apprendimento dell'uso delle istruzioni WHILE e REPEAT per controllare il numero di volte in cui alcune istruzioni vengono ripetute basandosi sui valori dei dati.

1b. Uso dell'istruzione FOR per controllare la ripetizione di un predeterminato numero di volte, e per semplificare l'assegnamento di nuovi valori a una variabile di controllo che aumenta (o diminuisce) di 1 ad ogni ciclo.

1c. Apprendimento dell'uso dell'istruzione IF, con una o due diramazioni, per decidere se un'istruzione (o un gruppo di istruzioni) sarà eseguita, dipendentemente da specifici valori dei dati.

1d. Uso dell'istruzione composta per permettere il trattamento di gruppi di istruzioni come se fossero una sola, a scopi di controllo.

1e. Uso delle variabili Booleane per la conservazione e riutilizzazione dei valori TRUE o FALSE necessari per controllare le istruzioni IF, WHILE e REPEAT.

1f. Apprendere a leggere semplici diagrammi di flusso che indicano la via per l'esecuzione di sezioni di un programma avente diramazioni o cicli.

1g. Introduzione nei vostri programmi di semplici valori di dati con l'uso delle istruzioni READ e READLN per ottenere informazioni inserite dalla tastiera.

1h. Uso dei diversi nuovi strumenti di programmazione conosciuti in questo capitolo, per costruire dei grafici più complessi, e per eseguire delle operazioni con stringhe, più complesse.

1i. Uso dell'indentazione per facilitare la visualizzazione e spiegazione della struttura di un programma.

1l. Correzione dei programmi nei quali le variabili di controllo non sono correttamente fissate su valori iniziali, o non risultano avere valori corretti quando la ripetizione è terminata.

2. Premessa

In diverse occasioni nel capitolo 2, abbiamo dovuto ripetere la stessa istruzione, o gruppo di istruzioni, parecchie volte. Per esempio la procedura UNQUADRATO è stata chiamata diverse volte nel programma QUADRATI. Lo stesso è avvenuto con TAGLIAPAROLA nel programma TAGLIA. Questi programmi sono molto semplici, vi sarete probabilmente chiesti come si possano scrivere dei programmi nei quali la stessa procedura deve essere chiamata centinaia di volte (o addirittura milioni di volte!). Le istruzioni WHILE, REPEAT e FOR introdotte in questo capitolo sono progettate per fronteggiare queste situazioni con facilità e con una necessità di scrittura minima.

Vi sarete anche chiesti come un programma possa manipolare una situazione speciale che insorge solo occasionalmente durante ripetizioni di calcoli, ma che richiede quando insorge, la risoluzione di speciali calcoli. Per esempio, un programma calendario ideato per visualizzare il mese ed il giorno, basato sul conteggio dei giorni a partire dal 1 gennaio, dovrebbe svolgere un'azione speciale alla fine di febbraio di un anno bisestile. Allo stesso modo, un programma che disegni linee, quale è il programma TARTARUGA da voi già usato, deve magari decidere se disegnare o meno una linea basandosi sull'elaborazione della lunghezza della linea e verificare che essa non superi la possibilità dello schermo. Decisioni simili sono facilmente affrontabili con l'istruzione IF.

Molto spesso esisteranno delle ragioni per cui dovrete controllare non una singola istruzione, ma un intero gruppo di istruzioni, sempre con l'uso di IF, WHILE, REPEAT o FOR. Le regole sintattiche PASCAL vi permettono questo procedimento raggruppando le istruzioni in una *"istruzione composta"*.

Le istruzioni IF, WHILE, REPEAT e FOR dipendono dal fatto che un elaboratore può essere programmato per "saltare" da un punto ad un altro del programma qualora alcune condizioni di prova vengano soddisfatte. Nel metodo di introduzione al linguaggio di programmazione tradizionale, agli studenti viene presentato piuttosto in anticipo l'uso dell'istruzione GOTO che serve per controllare direttamente questi salti del programma. Comunque l'uso dell'istruzione GOTO senza una considerevole dose di precauzione, è stato dimostrato essere la fonte di errori di logica, e la causa della

perdita di tempo nella correzione dei programmi. Le quattro istruzioni precedentemente citate rappresentano altrettanti modi di usare la possibilità del salto dell'hardware in un modo software che elimina molte delle possibilità di errore. Rimanderemo al capitolo 11, la discussione su alcune delle situazioni in cui l'uso di GOTO si dimostra molto vantaggioso. In questi casi potrà addirittura essere usato per evitare di fare errori. Per ora abbiamo combinato il sistema PASCAL in modo tale da non rendervi necessario l'uso di questa particolare istruzione.

3. L'istruzione WHILE

Il programma WHILETRA illustra l'uso dell'istruzione WHILE e produce il disegno mostrato nella figura 3-1. Partendo dal centro dello schermo viene disegnata una linea lunga 10 unità, nella direzione iniziale della tartaruga. Viene poi effettuato uno spostamento di 89 gradi e la variabile DISTANZA è aumentata di 3 unità, da 10 a 13. Un'altra linea lunga il numero di unità contenute in DISTANZA, viene disegnata seguita da un altro spostamento di 89 gradi, e così di seguito. Il disegno si arresta quando il valore di DISTANZA supera le 600 unità, ciò per impedire al disegno di uscire dallo schermo!

L'istruzione WHILE in linea 20 determina se il valore di DISTANZA è inferiore o uguale a 600. In questo caso viene chiamata la procedura PROSSLINEA. Considerato che PROSSLINEA può avere, ed ha in effetti un'influenza su DISTANZA, l'intestazione dell'istruzione WHILE confronta di nuovo DISTANZA con 600. Qualora quest'ultima fosse ancora uguale o inferiore a 600, viene chiamata di nuovo PROSSLINEA. Questo procedimento continua fino a quando DISTANZA diventa maggiore di 600, a quel punto PROSSLINEA non viene chiamata e l'intera istruzione WHILE si conclude.

```
 1: PROGRAMMA WHILETRA;
 2: VAR
 3:    DISTANZA,ANGOLO:INTEGER;
 4:    CAMBIO:INTEGER;
 5:
 6: PROCEDURE PROSSLINEA;
 7: BEGIN
 8:    MOVE(DISTANZA);
 9:    TURN(ANGOLO);
10:    DISTANZA:=DISTANZA+CAMBIO;
11: END (*PROSSLINEA*);
12:
13: BEGIN (*PROGRAMMA PRINCIPALE*)
14:    PENCOLOR(WHITE);
```

```
15:    DISTANZA:=10;
16:    ANGOLO:=89;
17:    CAMBIO:=3;
18: (*DISTANZA & CAMBIO corretti per Tektronix 4006*)
19: (*usa 4 e 1 per Terak 8510A; ?? per altri*)
20:    WHILE DISTANZA<=600 DO PROSSLINEA;
21: END.
```

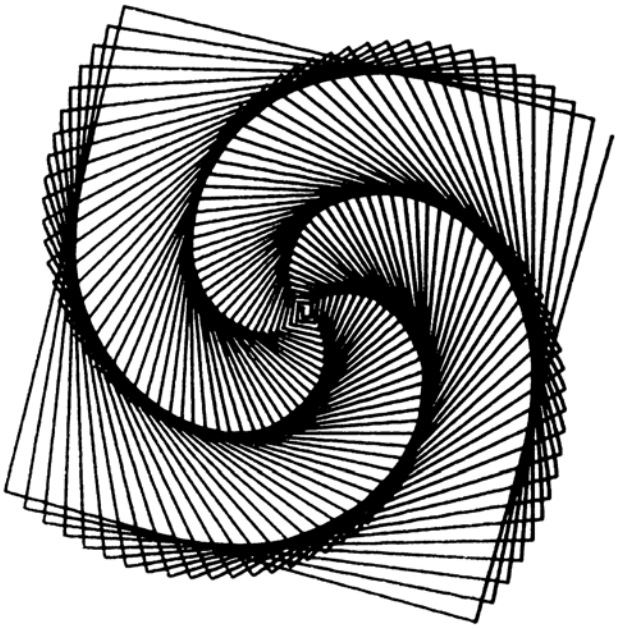


Figura 3-1

```
 1: PROGRAMMA WHILE1;
 2: VAR S:STRING;
 3: BEGIN
 4:    WRITELN('BATTI QUALSIASI STRINGA SEGUITA DA <RET>');
 5:    READLN(S);
 6:    WRITELN(S);
 7:    WHILE LENGTH(S)>0 DO
 8:    BEGIN
 9:       DELETE(S,1,1); (*TOGLIE PRIMO CARATTERE*)
10:       WRITELN(S);
11:    END (*WHILE*);
12: END.
```

Il programma WHILE1 illustra l'uso dell'istruzione WHILE e dell'istruzione composta, inoltre introduce in forma semplice l'istruzione READLN. Questo programma visualizza in linea 4, un messaggio con la richiesta di battitura sulla tastiera di una stringa. Dopo che il tasto di ritorno < RET > è stato premuto, l'istruzione READLN assegna i caratteri introdotti dalla tastiera (e visualizzati sullo schermo) alla variabile STRINGS. L'ultimo carattere assegnato ad S è quello battuto immediatamente prima della pressione del tasto < RET >.

Supponiamo ora che voi inseriate la stringa "VA VIA" in risposta al messaggio di suggerimento visualizzato in linea 4. Il risultato dovrebbe essere la visualizzazione delle seguenti linee:

```
VA VIA
A VIA
VIA
VIA
IA
A
```

Notate che la linea visualizzata sullo schermo mentre voi battevate sulla tastiera, è tuttora visualizzata. Quando l'esecuzione raggiunge la linea 7, il valore di LENGTH(S) viene confrontato con 0 (zero). Se è superiore a zero, verranno eseguite le linee da 8 a 11.

Per meglio comprendere la funzione dell'istruzione WHILE, riferitevi alla figura 3-2 illustrante i diagrammi sintattici e gli schemi a blocchi relativi al programma WHILE1. La parte (a) della figura 3-2 rappresenta uno *"schema a blocchi"*, esprimente in termini generali, come opera l'istruzione WHILE. Un'< espressione Booleana > è caratterizzata dal fatto che può adottare uno solo fra due valori possibili, cioè FALSE o TRUE. Dopo aver introdotto l'istruzione, viene fatta una prova per determinare se l'< espressione Booleana > vale TRUE. Se così fosse l'< istruzione > viene eseguita una volta. Questa < istruzione > può essere una qualsiasi istruzione di quelle già discusse, ed una qualsiasi di quelle introdotte in questo e nei successivi capitoli. Dopo che l'istruzione è stata eseguita, il valore dell'< espressione Booleana > viene di nuovo verificato. L'esecuzione dell'< istruzione > continua fino a quando l'< espressione Booleana > risulta FALSE. A questo punto l'istruzione WHILE si conclude. Le scatole rettangolari dello schema a blocchi contengono delle istruzioni che eseguono una o più azioni. Le scatole rombiche contengono < espressioni Booleane > e sono usate in relazione alle prove di cui si è discusso sopra.

La parte (b) mostra uno schema a blocchi descrivente il programma WHILE1, con un piccolo cambiamento. Nel primo rettangolo viene assegnata a S una costante specifica. Questo valore di S è quello già usato nell'illustrazione precedente. Potete

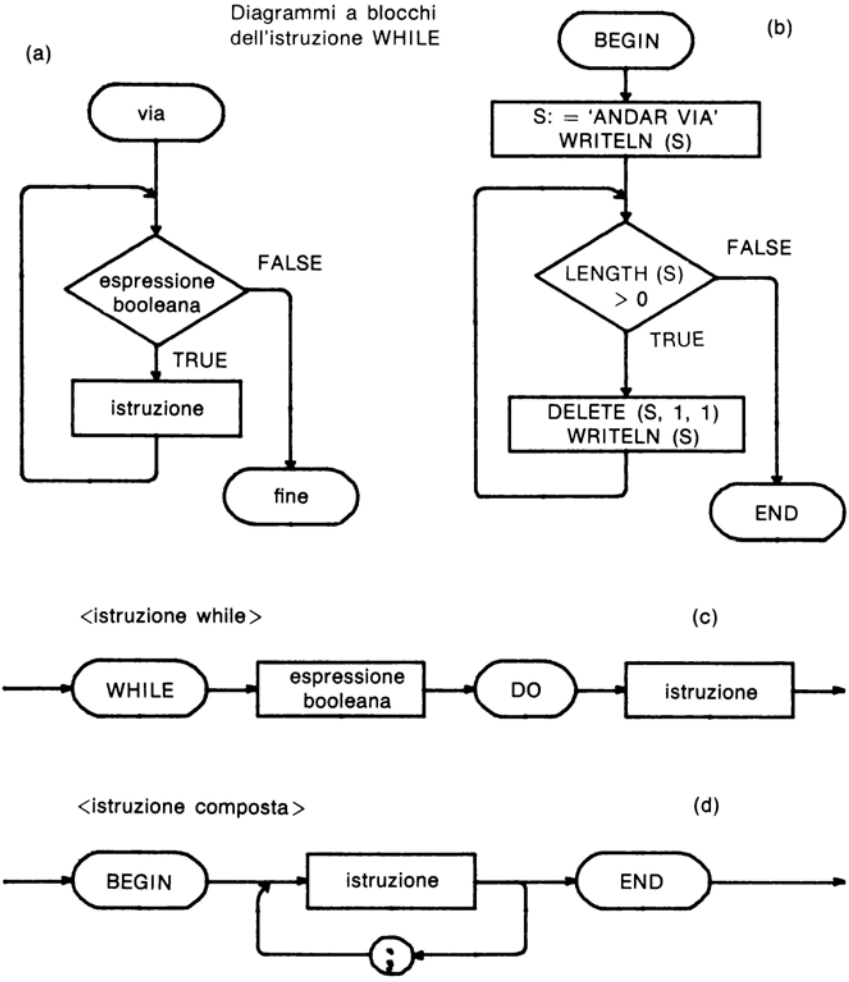

Diagrammi a blocchi
dell'istruzione WHILE
(a)
via
espressione
booleana
FALSE
TRUE
istruzione
fine
(b)
BEGIN
S: = 'ANDAR VIA'
WRITELN (S)
LENGTH (S)
> 0
FALSE
TRUE
DELETE (S, 1, 1)
WRITELN (S)
END
<istruzione while>
(c)
WHILE
espressione
booleana
DO
istruzione
<istruzione composta>
(d)
BEGIN
istruzione
END
;

Figura 3-2

esercitarvi con questo stesso programma usando delle stringhe differenti inserite dalla tastiera.

Entrambi le parti (a) e (b) della figura 3-2 illustrano alcuni termini del gergo comunemente usato dai programmatori. Se seguite la freccia dalla scatola "via" verso il basso fino al rombo ed al rettangolo sottostante, potete notare che la freccia uscente dal rettangolo ritorna verso l'alto e si riunisce a quella iniziale. Questo rappresenta una via di controllo del programma chiamata in gergo *"ciclo"* (loop).

La parte (c) in figura 3-2 illustra la sintassi dell'istruzione WHILE. Oltre a "WHILE" è presente "DO", un identificatore riservato, conosciuto in questo contesto dal compilatore. L'<istruzione> controllata da WHILE è infatti in questo esempio un' *"istruzione composta"*.

La sintassi dell'<istruzione composta> è illustrata nella parte (d) della figura. Come potete notare, la parte eseguibile di un <blocco> è in effetti un'<istruzione composta>. Possiamo usare i simboli BEGIN ed END, anch'essi identificatori riservati, come se contenessero delle parentesi per un gruppo di istruzioni. Se esistessero dei simboli per contenere informazioni, veramente distintivi, PASCAL userebbe probabilmente questi ultimi invece di BEGIN e END. Un'<istruzione composta> può essere usata in qualsiasi punto del programma dove sia lecito usare qualsiasi altra istruzione. L'<istruzione composta> è un modo molto vantaggioso di riunire insieme diverse istruzioni che debbano essere sempre eseguite insieme.

```
 1: PROGRAMMA WHILE2;
 2: VAR S:STRING;
 3:       L,N:INTEGER;
 4: BEGIN
 5:    WRITELN('BATTI QUALSIASI STRINGA SEGUITA DA <RET>');
 6:    READLN(S);
 7:    N:=1; (*forza N all'inizio di S*)
 8:    L:=LENGTH(S);
 9:    WHILE N<L DO
10:       BEGIN
11:          WRITE(S[N], '-');
12:          N:=N+1
13:       END;
14:    WRITELN(S[L]);
15:             (*visualizza l'ultimo carattere senza la lineetta finale*)
16: END.
```

Il programma WHILE2 vi dà un altro esempio dell'uso dell'istruzione WHILE e del-

l'istruzione composta. Analizzate il programma prima di continuare a leggere la descrizione delle azioni che compie.

Di nuovo WHILE2 accetta una stringa inserita mediante la tastiera e la assegna alla variabile S. Supponiamo che in risposta al messaggio di suggerimento voi abbiate risposto con "LINEATUTTO". Il risultato dovrebbe essere la seguente visualizzazione:

```
L-I-N-E-A-T-U-T-T-O
```

Capite perché non c'è nessuna lineetta dopo la "O" finale?

4. L'istruzione IF

Fate ora riferimento al programma IFDEMO1, che è molto simile concettualmente al programma TAGLIA nella sezione 2.12, ma che differisce in molti dettagli. Dovreste confrontare i due programmi prima di passare oltre.

In IFDEMO1, chiamiamo più volte la procedura UNAPAROLA con l'istruzione WHILE. Ad ogni chiamata, la stringa contenuta in S viene ridotta di una parola, supponendo che in S rimanga almeno uno spazio bianco quando UNAPAROLA viene chiamata. Se battiamo sulla tastiera "QUESTO NON È TEMPO DI ROSE", quale risposta al messaggio di suggerimento in linea 16 del programma, il risultato visualizzato dovrebbe essere:

```
QUESTO NON È TEMPO DI ROSE
NON È TEMPO DI ROSE
È TEMPO DI ROSE
TEMPO DI ROSE
DI ROSE
ROSE
```

La linea battuta dalla tastiera rimarrà sullo schermo appena sopra alla prima di queste linee visualizzate. In entrambi i programmi l'eliminazione dell'ultimo spazio dalla STRINGA farà sì che POS abbia valore 0 (zero). In TAGLIA questo impedisce che l'ultima parola della stringa venga cancellata, e successive chiamate di TAGLIAPAROLA daranno come risultato la visualizzazione dell'ultima parola. In IFDEMO1, un valore zero in N è scoperto in linea 11 dall'istruzione IF, causando la cancellazione di tutti i caratteri rimasti in S (cioè LENGTH(S) caratteri). In questo caso la verifica di N >0, in linea 8, sarà FALSE e l'istruzione DELETE in linea 9 non sarà eseguita. In un caso come questo si dice in gergo che la verifica falsa (come in linea 8) causa la *"caduta"* del programma al punto successivo alla fine dell'istruzione controllata da IF (come in linea 11).

```
 1: PROGRAMMA IFDEMO1;
 2: VAR SPAZIO,S:STRING;
 3:    N:INTEGER;
 4:
 5: PROCEDURE UNAPAROLA;
 6: BEGIN
 7:    N:=POS(SPAZIO,S); (*cerca primo spazio in S*)
 8:    IF N >0 (*verifica se lo spazio è stato trovato*)
 9:       THEN DELETE(S,1,N);
10:            (*non cancellare se non è stato trovato lo spazio*);
11:    IF N=0 THEN DELETE(S,1,LENGTH (S) );
12:    WRITELN(S)
13: END (*UNAPAROLA*);
14:
15: BEGIN (*PROGRAMMA PRINCIPALE*)
16:    WRITELN('BATTI QUALSIASI STRINGA SEGUITA DA <RET>');
17:    READLN(S);
18:    SPAZIO:=' ';
19:    WHILE LENGTH(S) >0 DO UNAPAROLA;
20: END.
```

5. Istruzione IF a due diramazioni, sintassi delle istruzioni IF

Avrete probabilmente notato che in IFDEMO1, era necessario verificare il valore di N due volte, dapprima in linea 8 per vedere se il valore era superiore a zero, poi in linea 11 per vedere se era esattamente uguale a zero. (POS non ritorna un valore inferiore a zero in nessuna circostanza). Quanto sopra comporta un lavoro di scrittura maggiore di quanto sarebbe realmente necessario, può inoltre condurre a confusione ed errori in programmi più complessi.

Una via migliore per affrontare questo problema è illustrata nel programma IFDEMO2. Fino alle linee 8 e 9 l'elaborazione è la stessa di quella eseguita in IFDEMO1. In linea 10 l'identificatore riservato ELSE riporta al testo in linea 8, allo stesso modo di THEN, in IFDEMO1. L'istruzione successiva ad ELSE viene eseguita qualora il risultato della prova sia FALSE. La logica del programma è stata leggermente cambiata affinchè venga visualizzato il messaggio "NON PIÙ SPAZI" che verifica il raggiungimento della fine del programma.

La figura 3-3 illustra la sintassi ed i diagrammi di flusso dell'istruzione IF. Come si vede nella parte (a), sia ELSE che la sua associata (cioè la "clausola ELSE") sono opzionali nella sintassi. Gli schemi a blocchi in (b) e (c) mostrano che la differenza

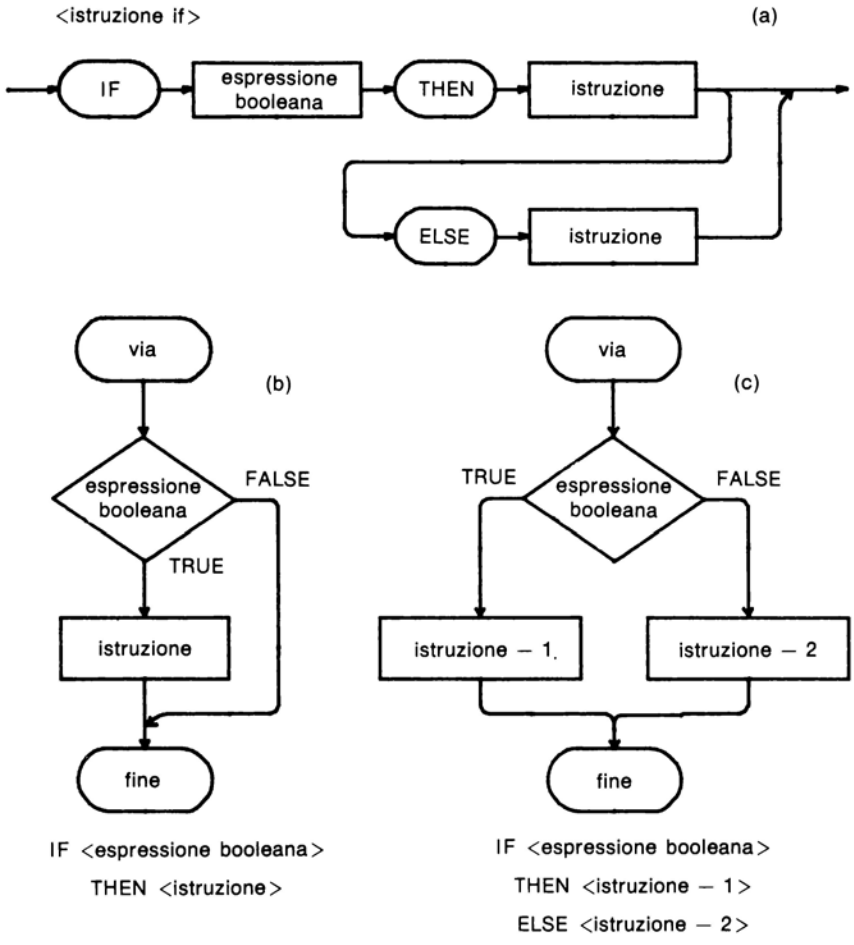

<istruzione if>
(a)
IF
espressione booleana
THEN
istruzione
ELSE
istruzione
via
(b)
espressione booleana
FALSE
TRUE
istruzione
fine
IF <espressione booleana>
THEN <istruzione>
via
(c)
TRUE
espressione booleana
FALSE
istruzione – 1
istruzione – 2
fine
IF <espressione booleana>
THEN <istruzione – 1>
ELSE <istruzione – 2>

Figura 3-3

principale fra le due forme dell'istruzione IF sta nel fatto che nell'IF ad una diramazione (parte (a)) l'istruzione controllata può essere completamente evitata. Nella forma a due diramazioni (parte (b)) una o l'altra delle due istruzioni deve essere eseguita. Avrete frequenti occasioni di usare ambedue le forme procedendo nell'uso di questo libro.

```
 1: PROGRAMMA IFDEMO2;
 2: VAR SPAZIO,S:STRING;
 3:   N:INTEGER;
 4:
 5: PROCEDURE UNAPAROLA;
 6: BEGIN
 7:   N:=POS(SPAZIO,S); (*trova primo spazio in S*)
 8:   IF N>0 (*verifica se lo spazio è stato trovato*)
 9:     THEN DELETE(S,1,N)
10:     ELSE
11:       BEGIN
12:         WRITELN('NON PIÙ SPAZI');
13:         DELETE(S,1,LENGTH(S));
14:       END;
15:   WRITELN(S)
16: END (*UNA PAROLA*);
17:
18: BEGIN (*PROGRAMMA PRINCIPALE*)
19:   WRITELN('BATTI QUALSIASI STRINGA SEGUITA <RET>');
20:   READLN(S);
21:   SPAZIO:=' ';
22:   WHILE LENGTH(S)>0 DO UNAPAROLA;
23: END.
```

Il programma tipo TRACCIANOME dà un esempio dell'uso dell'istruzione IF nel campo dei grafici. Qualora in risposta al messaggio "BATTI QUALSIASI NOME", voi inserite "MARIO ROSSI" seguito da <RET>, sul vostro video apparirà il disegno riportato nella figura 3-4. Il disegno è in stretta dipendenza del nome da voi inserito mediante battitura e, in un certo senso, è una "firma" creata dall'elaboratore. Provate ad inserire il vostro nome per vedere che firma ne risulta. (Nel caso in cui stiate usando il terminale video Tektronix 4006, inserite il valore intero 20 rispondendo al messaggio "DELTA:". Usate 6 al posto di 20 nel caso di un terminale Terak 8510A. Altri valori saranno necessari nel caso in cui stiate usando un terminale video diverso da quelli appena considerati).

TRACCIANOME differisce sotto molti aspetti dai programmi precedenti per quanto

riguarda l'immissione dei dati nel programma dalla tastiera. READLN in linea 8 stabilisce le condizioni per convertire il valore intero da voi inserito, nella forma "binaria" interna usata dall'elaboratore per i valori interi, il compilatore sa che l'identificatore DELTA è stato dichiarato essere di <tipo> INTEGER. Allo stesso modo le istruzioni READ nelle linee 11 e 20 si aspettano un valore unico di <tipo> CHAR. READ è diverso da READLN in quanto il primo non richiede che venga battuto il tasto <RET> prima della fine. TRACCIANOME introduce la funzione interna ODD(X) che ritorna TRUE se il valore del parametro intero (X) è un numero dispari, cioè 1, 3, 5, 7, ...

TRACCIANOME introduce inoltre l'uso delle funzioni di conversione intero-carattere ORD e CHR. ORD(CH) ritorna l'intero equivalente al valore del carattere di CH. CHR(<espressione intera>) ritorna il carattere equivalente al valore dell'espressione. Ognuno dei caratteri visualizzati, e molti di quelli che non vengono visualizzati ma che servono a scopi di controllo, hanno un equivalente valore intero. Questo tipo di funzioni di conversione permettono di calcolare un nuovo valore carattere usando <espressioni aritmetiche>. Il tasto <RET> corrisponde all'intero 13. Per ragioni che spiegheremo nel capitolo 7, il carattere riportato nel programma quando si batte <RET> risulta uguale a <spazio> ed ha un valore intero di 32. Esiste un modo per determinare se è stato battuto il tasto <RET> o lo <spazio> (la barra spaziatrice), ma non parleremo di queste complicazioni fino al capitolo 7.

```
 1: PROGRAMMA TRACCIANOME;
 2: VAR DIMENSIONE,DELTA:INTEGER;
 3:     CH:CHAR;
 4: BEGIN
 5:     WRITELN('TRACCIANOME');
 6:     PENCOLOR(WHITE);
 7:     WRITE('DELTA:');
 8:     READLN(DELTA);
 9:     SIZE:=DELTA;
10:     WRITE('BATTI QUALSIASI NOME:');
11:     READ(CH);
12:     WHILE CH <> CHR(127(*DEL*)) DO
13:       BEGIN
14:         MOVE(DIMENSIONE);
15:         IF ODD(ORD(CH)) THEN
16:           TURN(-90)
17:         ELSE
18:           TURN(90);
19:         DIMENSIONE:=DIMENSIONE+DELTA;
20:           READ(CH);
21:       END;
22: END.
```

Per evitare questa complicazione e per avere un carattere adatto a fermare l'azione di REPEAT, useremo <DEL> (chiamato "RUBOUT" su alcune tastiere) invece di <RET>. <DEL> equivale all'intero 127, ed è usato in linea 12 del programma. Il programma termina quando premete il tasto <DEL> e non quando premete il tasto <RET>. Sia <RET> che <DEL> non appaiono esplicitamente in una <costante stringa> tra virgolette, in quanto sono dei caratteri "non stampabili". È comunque possibile trasmettere al video questi ed altri *"caratteri di controllo"* convertendoli dal valore intero al carattere equivalente per mezzo della funzione CHR.

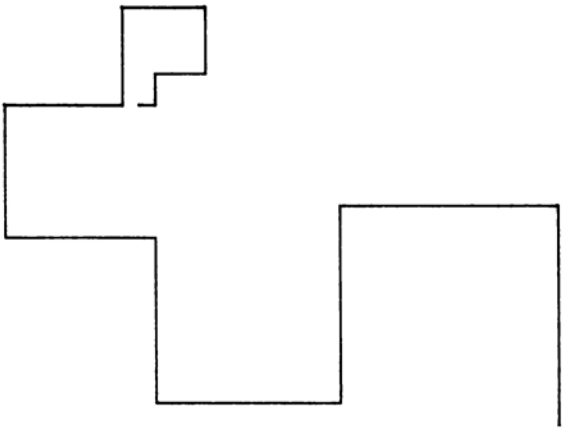


Figura 3-4

ESERCIZIO 3.1:
Il programma TRACCIANOME così com'è stampato sopra, non fornisce alcuna protezione contro la fuoriuscita della tartaruga dallo schermo. Il numero dei caratteri da inserire perché ciò avvenga dipende dal nome battuto.

Modificate TRACCIANOME in modo che termini quando la tartaruga sta per uscire dai limiti dello schermo, ed in modo che conti i caratteri immessi e "visualizzati" come linee fino al termine del programma. La somma dei caratteri dovrebbe essere visualizzata prima del termine del programma, ma dopo il completamento dell'istruzione REPEAT. Invece di immettere dei nomi potete fare un interessante gioco cercando di trovare la parola italiana più lunga e corretta ortograficamente, il cui grafico sia contenibile nello schermo. Ricordate che la tartaruga può girare di 90 gradi sia in senso orario che antiorario. È così possibile conoscere in ogni momento la posizione della tartaruga tenendo i contatori di posizione (nelle variabili intere) per i suoi spo-

stamenti in senso verticale o orizzontale, lontano dal centro del video. Sia il contatore orizzontale che quello verticale cambiano in seguito al tracciamento di una nuova linea, ma non cambiano immediatamente entrambi.

6. Sintassi delle espressioni Booleane

Fate riferimento alla figura 3-5 per la sintassi semplificata riguardante l'uso delle <espressioni Booleane> in PASCAL. "Booleane" è scritto con la maiuscola iniziale in quanto deriva dal nome del famoso matematico George Boole. Come mostra il diagramma, esistono sei differenti simboli che si possono usare nel confronto di due <espressioni semplici> per ottenere un risultato TRUE/FALSE. Entrambe le espressioni semplici comparate devono essere dello stesso <tipo>, cioè, per il momento, INTEGER o CHAR. I simboli sono definiti come segue:

=	uguale
<>	diverso
<	minore di
>	maggiore di
<=	minore o uguale a
>=	maggiore o uguale a

Negli ultimi due di questi simboli è essenziale che il segno di uguale ("=") appaia dopo la parentesi angolare. Nel caso di diverso, l'ordine delle due parentesi non deve essere invertito.

Potete combinare fra loro due o più <espressioni Booleane>, usando i simboli AND e OR. Per esempio, se W=1, X=2, Y=3, Z=4 allora

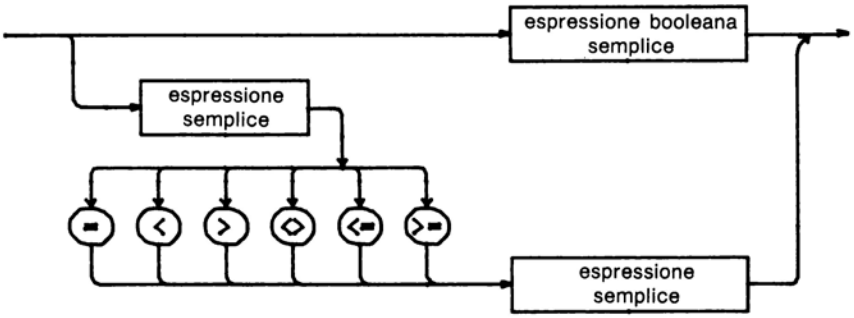
```
(X >0) OR (Y >10)
```

vale TRUE perché X è maggiore di zero, anche se Y non è maggiore di 10. L'espressione risulta essere TRUE quando *uno o l'altro* dei confronti è TRUE, e risulta essere FALSE solo se *entrambi* i confronti sono FALSE. Ancora:

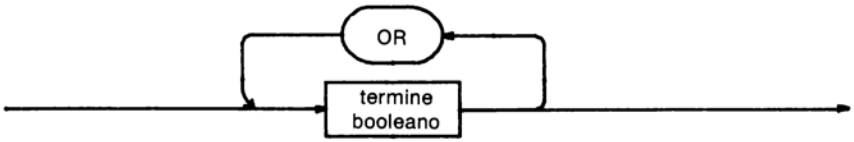
```
( (W–X) <0) AND ( (Z–X) >0)
```

è TRUE perché *entrambi* i confronti sono TRUE. Sarebbe invece valutato FALSE qualora *uno o l'altro* fosse FALSE. La sintassi indica che quando vengono usati AND o OR è necessario racchiudere ogni confronto fra parentesi, in caso contrario non è necessario.

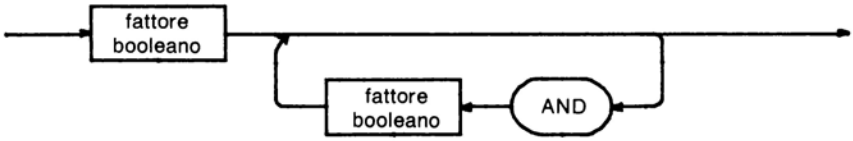
<espressione booleana>



<espressione booleana semplice>



<termine booleano>



<fattore booleano>

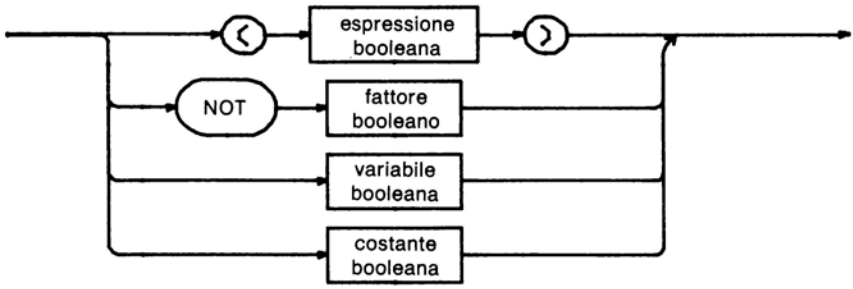


Figura 3-5

Discuteremo ancora della sintassi delle espressioni Booleane alla fine di questo capitolo, in relazione alle variabili Booleane.

ESERCIZIO 3.2:
Modificate il programma TRACCIANOME (quello originale oppure quello da voi ottenuto dall'esercizio 3.1) in modo da tracciare delle linee solo con la battitura di caratteri alfabetici. Per fare ciò dovete sapere che tutti i caratteri maiuscoli dalla 'A' alla 'Z' sono maggiori o uguali ad 'A' ed allo stesso tempo minori o uguali a 'Z'. Analogamente nessun carattere minuscolo è minore di 'a' o maggiore di 'z'.

7. L'istruzione FOR

Le istruzioni WHILE e REPEAT possono essere usate quali basi per qualsiasi tipo di controllo sulle ripetizioni. L'istruzione FOR è stata progettata per semplificare il controllo di un tipo di ripetizione tanto frequente da giustificare l'introduzione in PASCAL di una istruzione particolare. Molto spesso nello scrivere un programma avete necessità di trovare un modo semplice di esecuzione delle seguenti operazioni:

1) Fare in modo che un'<istruzione> venga ripetuta un determinato numero di volte.

2) Fare in modo che una <variabile> INTEGER aumenti (o diminuisca) di 1 ad ogni ripetizione.

Per esempio volete che la variabile K assuma i seguenti valori in progressione uniforme:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Nella manipolazione di STRINGHE può esistere la frequente necessità di verificare se ogni carattere della STRINGA soddisfa determinate condizioni. Un'applicazione potrebbe essere quella di contare il numero totale di vocali ('a', 'e', 'i', 'o', o 'u') contenute in una riga di testo. In questo caso la vostra *"variabile di controllo"* K dovrebbe assumere dei valori uguali a tutti i possibili numeri da 1 fino a LENGTH della variabile STRING.

L'istruzione FOR è illustrata nel programma POLIGONO che vi suggerisce le modalità per disegnare una figura somigliante ad un cerchio, usando unicamente delle linee rette. I disegni ottenuti con questo programma sono mostrati in figura 3-6. La procedura POLI disegna delle linee NLATI di lunghezza LUNG. Dopo ogni linea la tartaruga si sposterà secondo dei gradi ANGOLO. In altri termini, il punto di partenza

```
 1: PROGRAMMA POLIGONO;
 2: VAR SCALA:INTEGER;
 3:
 4: PROCEDURE POLI(NLATI,LUNG,ANGOLO,X,Y:INTEGER);
 5: VAR I:INTEGER;
 6: BEGIN
 7:    MOVETO(X*SCALA,Y*SCALA);
 8:    PENCOLOR(WHITE);
 9:    FOR I:=1 TO NLATI DO
10:    BEGIN
11:       MOVE(LUNG*SCALA);
12:       TURN(ANGOLO);
13:    END;
14:    PENCOLOR(NONE);
15: END (*POLI*);
16:
17: BEGIN (*PROGRAMMA PRINCIPALE*)
18:    SCALA:=9; (*9 per Tektronix 4006*)
19:             (* Usa 3 per Terak; ?? per altri*)
20:    POLI(5,16,72,-30,-30);
21:    POLI(10,8,36,20,-30);
22:    POLI(20,4,18,20,6);
23:    POLI(40,2,9,-30,6);
24: END.
```

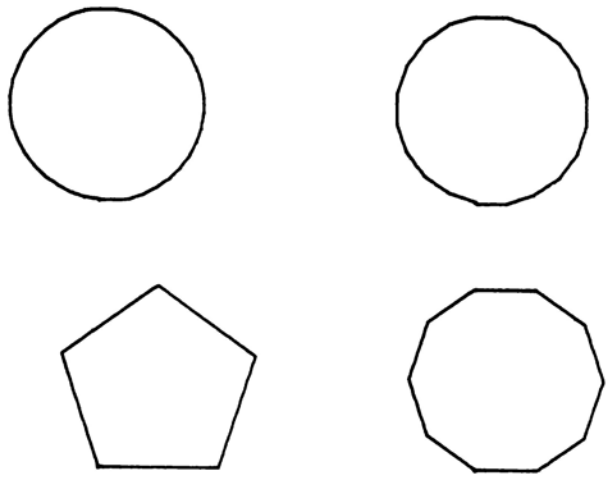


Figura 3-6

```
 1: PROGRAMMA GRAFCARTA;
 2: VAR SCALA,X,Y:INTEGER;
 3:
 4: PROCEDURE LINORI(ASC:INTEGER);
 5: VAR I:INTEGER;
 6: BEGIN
 7:    MOVETO(-100*SCALA,ASC*10*SCALA);
 8:    PENCOLOR(WHITE);
 9:    MOVETO(100*SCALA,ASC*10*SCALA);
10:    PENCOLOR(NONE);
11: END (*LINORI*);
12:
13: PROCEDURE LINORI(ORD:INTEGER);
14: VAR I:INTEGER;
15: BEGIN
16:    MOVETO(ORD*10*SCALA,100*SCALA);
17:    PENCOLOR(WHITE);
18:    MOVETO(ORD*10*SCALA,-100*SCALA);
19:    PENCOLOR(NONE);
20: END (*LINVER*);
21:
22: BEGIN
23:    SCALA:=3; (*3 per Tektronix 4006*)
24:    (*usa 1 per Terak o altri schermi*)
25:    FOR X:=-10 TO 10 DO LINORI(X);
26:    FOR Y:=-10 TO 10 DO LINVER(Y);
27: END.
```

per disegnare ciascun poligono è in (X,Y) relativo all centro dello schermo, dove X rappresenta la distanza orizzontale e Y quella verticale. L'intestazione di FOR si trova in linea 9, mentre l'istruzione controllata da FOR si trova nelle linee da 10 a 13. In questo esempio la variabile semplice I è usata per controllare il numero di ripetizioni occorrenti da 1 fino a NLATI incluso. I, non viene usata all'interno dell'istruzione controllata, nonostante sia possibile usarla in quel caso, e spesso è usata in istruzioni controllate da FOR.

In figura 3-7 sono illustrati la sintassi e lo schema a blocchi per l'istruzione FOR. Come potete vedere dallo schema a blocchi il funzionamento interno di FOR è molto simile a quello di WHILE. L'istruzione FOR ci permette di definire celermente, il valore iniziale della variabile di controllo, cioè di *"inizializzare"* la variabile e di cambiarla di 1 ad ogni ciclo. Quando fra il <valore di partenza> ed il <valore limite> appare "TO", la variabile viene aumentata di 1 ad ogni ciclo. Quando appare "DOWNTO", la variabile viene diminuita di 1 ad ogni ciclo.

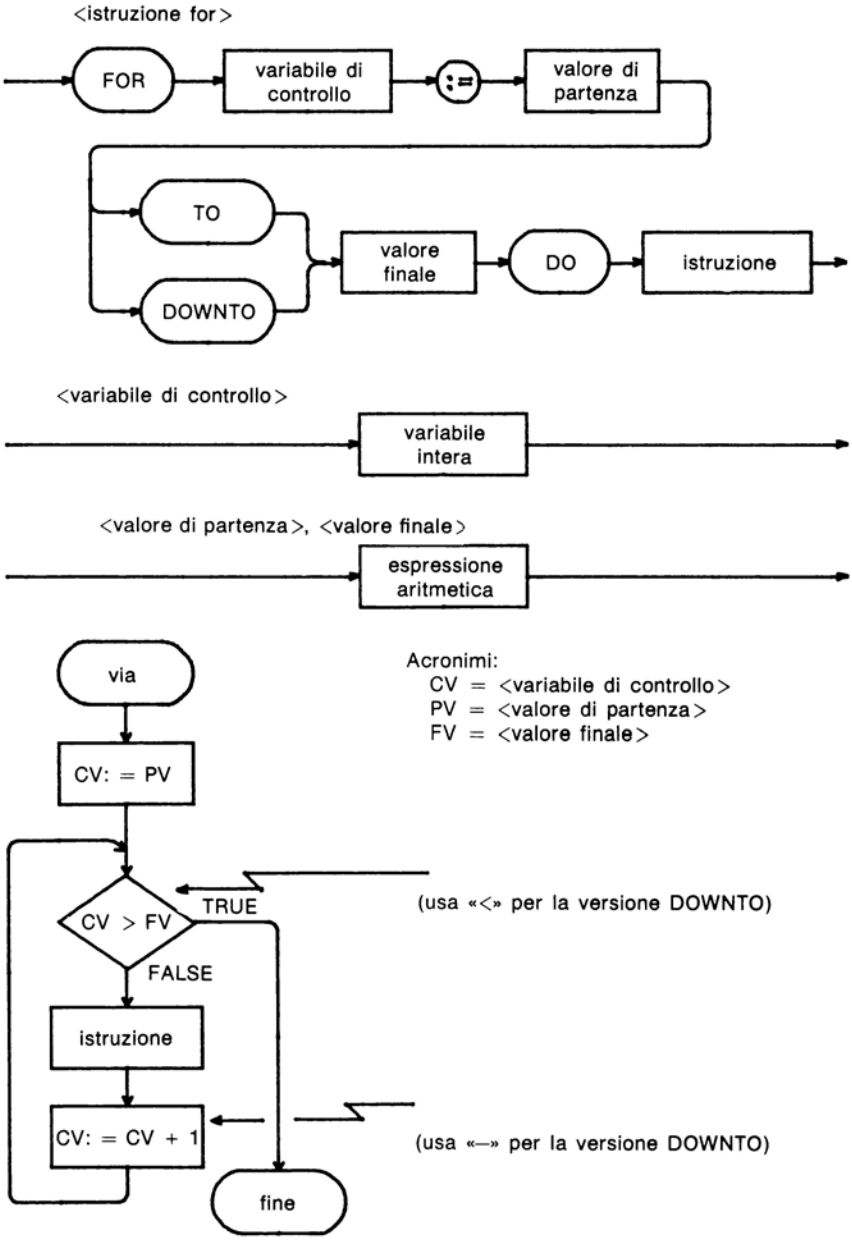

<istruzione for>
FOR
variabile di controllo
:=
valore di partenza
TO
DOWNTO
valore finale
DO
istruzione
<variabile di controllo>
variabile intera
<valore di partenza>, <valore finale>
espressione aritmetica
via
CV: = PV
CV > FV
TRUE
FALSE
istruzione
CV: = CV + 1
fine
Acronimi:
CV = <variabile di controllo>
PV = <valore di partenza>
FV = <valore finale>
(usa «<» per la versione DOWNTO)
(usa «—» per la versione DOWNTO)

Figura 3-7

Il programma GRAFCARTA fornisce un secondo esempio grafico dell'uso dell'istruzione FOR. Il relativo disegno è riportato nella figura 3-8. In questo programma, le variabili X ed Y vengono usate all'interno dell'istruzione controllata, e controllano la posizione delle linee disegnate da ciascuna delle due procedure semplici. Notate inoltre che la variabile di controllo non deve necessariamente cominciare con 1, come avviene per altri linguaggi di programmazione.

Un punto su cui bisogna andar cauti, relativamente all'istruzione FOR, riguarda il valore della <variabile di controllo> dopo che FOR ha completato il suo lavoro. Lo schema a blocchi suggerisce che il valore finale della <variabile di controllo> sarà maggiore (versione TO) o minore (versione DOWNTO) del <valore limite>. Questa azione non è garantita e le regole PASCAL stabiliscono che il valore della variabile di controllo è *"indefinito"* dopo la conclusione di FOR. È una cattiva pratica di programmazione quella di dipendere dal suggerimento dello schema a blocchi per quanto riguarda il valore della variabile di controllo dopo che l'istruzione FOR è terminata. In alcuni sistemi PASCAL, il valore verrà esplicitamente trasformato in valore imprecisato prima che la nuova istruzione possa essere eseguita. Potete usare la stessa variabile, in seguito nel programma, per altri scopi purché la inizializzate con un nuovo valore.

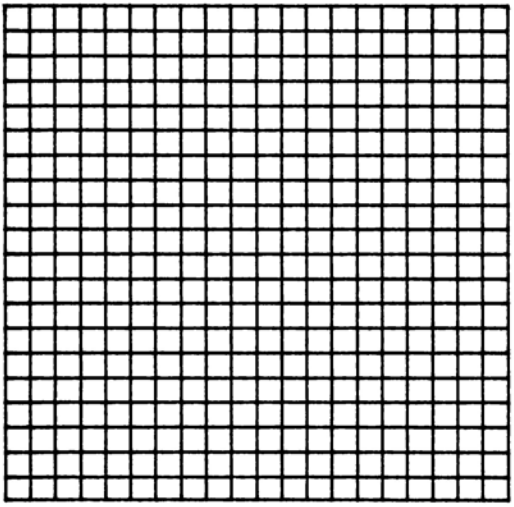


Figura 3-8

```
 1: PROGRAMMA FOR1;
 2: VAR S,W:STRING;
 3:    N:INTEGER;
 4: BEGIN
 5:    W:='HEAP*';
 6:    S:=W;
 7:    FOR N:=1 TO 6 DO
 8:    BEGIN
 9:       WRITELN(S);
10:       S:=CONCAT(S,W);
11:    END;
12:    WRITELN(S);
13: END.
```

```
 1: PROGRAMMA FOR2;
 2: VAR S: STRING;
 3:    K,N: INTEGER;
 4: BEGIN
 5:    WRITELN('BATTI QUALSIASI STRINGA SEGUITA DA <RET>');
 6:    READLN (S);
 7:    N:=LENGTH(S);
 8:    FOR K:=N DOWNTO 1 WRITELN (': K, S[K]);
 9: END.
```

I programmi FOR1 e FOR2 forniscono degli esempi dell'uso dell'istruzione FOR che sfruttano stringhe. FOR1 visualizza le seguenti linee:

```
HEAP*
HEAP*HEAP*
HEAP*HEAP*HEAP*
HEAP*HEAP*HEAP*HEAP*
HEAP*HEAP*HEAP*HEAP*HEAP*
HEAP*HEAP*HEAP*HEAP*HEAP*HEAP*
HEAP*HEAP*HEAP*HEAP*HEAP*HEAP*HEAP*
```

Tutte, esclusa l'ultima linea sono visualizzate dalla linea 9 del programma. L'istruzione CONCAT in linea 10 aggiunge un nuovo "HEAP*" alla fine di S, ad ogni ciclo.

Se, in risposta al messaggio di suggerimento visualizzato in linea 5 di FOR2, inserite la parola "DIAGONAL", il programma visualizzerà (la prima linea è visualizzata quale parte dell'azione di READLN mentre batte i tasti):

```
DIAGONAL
          L
         A
        N
       O
      G
     A
    I
  D
```

Notate che la variabile di controllo K è qui usata all'interno del ciclo controllato dall'istruzione FOR, ma che nessun valore nuovo viene assegnato a K all'interno dell'istruzione controllata. *È essenziale che non assegnate un nuovo valore alla variabile di controllo all'interno dell'istruzione controllata da FOR*, in quanto ciò renderebbe difficile o impossibile prevedere quanti cicli sarebbero eseguiti.

Questo programma illustra un aspetto delle istruzioni WRITE e WRITELN che non abbiamo finora mostrato. La frase ' ':K stabilisce che K colonne saranno completate con caratteri bianchi. Se ad una qualsiasi voce di un elenco che deve essere visualizzato dall'istruzione WRITE o WRITELN, fate seguire un segno di due punti (":") seguito a sua volta da un'<espressione aritmetica>, il numero minimo di colonne da occupare sarà dato dal valore di quell'espressione. Se la quantità che deve essere visualizzata non è uno spazio bianco, e non riempie almeno K colonne, allora delle colonne bianche, extra, saranno inserite a sinistra in modo che la voce completi K colonne. Se la voce riempie più di K colonne, allora questo maggior numero di colonne sarà visualizzato e nessuno spazio bianco sarà inserito a sinistra.

ESERCIZIO 3.3:
Modificate il programma FOR1 in modo che visualizzi le seguenti linee:

```
                                    HEAP*
                                HEAP*HEAP*
                            HEAP*HEAP*HEAP*
                        HEAP*HEAP*HEAP*HEAP*
                    HEAP*HEAP*HEAP*HEAP*HEAP*
HEAP*HEAP*HEAP*HEAP*HEAP*HEAP*
```

Notate che nell'ultima linea vi sono solo 6 ripetizioni di HEAP*.

Fate ora un passo avanti e modificate il programma così da ottenere la stessa visualizzazione, ma senza usare CONCAT, INSERT, COPY o DELETE.
Suggerimento: ciò che vi sarà possibile inserendo una seconda istruzione FOR all'interno di quella già esistente nel programma. Controllate poi WRITE ('HEAP*') con questa istruzione FOR interna. È inteso che questa ultima deve essere una <variabile di controllo> diversa da quella usata per controllare l'istruzione FOR esterna (N nel caso riportato sopra).

Infine, per vedere più chiaramente l'ordine di cambiamento delle due variabili di controllo, aggiungete delle istruzioni WRITE che traccino i valori delle variabili ogni volta che cambiano. La variabile di controllo esterna (N in FOR1) dovrebbe essere mostrata una sola volta per ogni linea visualizzata usando per esempio: WRITE ('N:',N). Qualora l'istruzione FOR interna sia controllata dalla variabile K, allora potete usare WRITE ('K:',K) per ogni giro del ciclo interno. Le suddette tracce dovrebbero essere visualizzate inframmezzate dalle ripetizioni di HEAP*.

8. L'istruzione REPEAT

L'istruzione REPEAT è molto affine all'istruzione WHILE. Fate prima riferimento al programma REPEAT1, quale esempio. Se rispondete al messaggio visualizzato in linea 5 con la stringa "DAI CHE AUMENTI", il programma visualizzerà le linee seguenti:

```
D
DA
DAI
DAI
DAI C
DAI CH
DAI CHE
DAI CHE
DAI CHE A
DAI CHE AU
DAI CHE AUM
DAI CHE AUME
DAI CHE AUMEN
DAI CHE AUMENT
DAI CHE AUMENTI
```

Sintassi e schema a blocchi per l'istruzione REPEAT sono mostrati nella figura 3-9. Notate che l'ordine dell'<istruzione> controllata ed il testo dell'<espressione Booleana> è invertito rispetto all'istruzione WHILE. Mentre WHILE verifica prima di eseguire l'<istruzione> controllata, REPEAT verifica dopo aver eseguito l'istruzione controllata. Ciò significa che REPEAT eseguirà sicuramente l'istruzione controllata almeno una volta. L'istruzione WHILE potrà non eseguire la istruzione controllata se la verifica dà FALSE alla prima prova.

```
 1: PROGRAMMA REPEAT1;
 2: VAR S,SG:STRING;
 3:       L, N: INTEGER;
 4: BEGIN
 5:   WRITELN('BATTI QUALSIASI STRINGA SEGUITA DA <RET>');
 6:   READLN(S);
 7:   N:=1;
 8:   L:=LENGTH(S);
 9:   REPEAT
10:     SG:=COPY(S,1,N);
11:     WRITELN(SG);
12:     N:=N+1
13:   UNTIL N>L
14: END.
```

Notate che l'aspetto dell'istruzione REPEAT è molto simile a quello di un'istruzione composta, eccetto che per l'assenza di BEGIN e END. L'identificatore riservato REPEAT serve sia per comunicare al compilatore il tipo di istruzione che sarà eseguita successivamente, sia come parentesi sinistra del programma nello stesso modo di BEGIN nella istruzione composta. Analogamente UNTIL serve per introdurre l'<espressione Booleana> che verrà verificata, e serve quale sostituto di END che sarebbe altrimenti necessario nell'istruzione composta. Notate che non c'è nessun ostacolo nel controllare una istruzione composta con REPEAT:

```
REPEAT
  BEGIN
    istruzione-1;
    istruzione-2;
  END
UNTIL <espressione Booleana>;
```

In questo caso BEGIN e END sono ridondanti ed inutili, ma non fanno alcun danno.

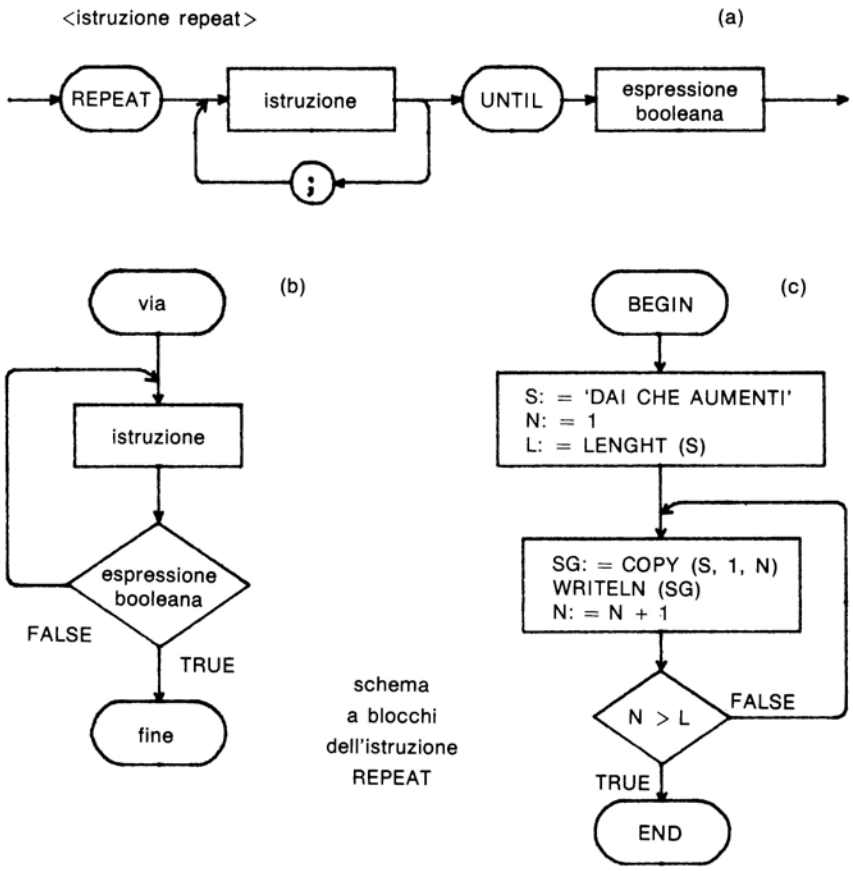

<istruzione repeat>
(a)
REPEAT
istruzione
UNTIL
espressione booleana
;
(b)
via
istruzione
espressione booleana
FALSE
TRUE
fine
schema a blocchi dell'istruzione REPEAT
(c)
BEGIN
S: = 'DAI CHE AUMENTI'
N: = 1
L: = LENGHT (S)
SG: = COPY (S, 1, N)
WRITELN (SG)
N: = N + 1
N > L
FALSE
TRUE
END

Figura 3-9

Generalmente un'istruzione WHILE può essere convertita in un'istruzione REPEAT producente lo stesso effetto se invertite l'<espressione Booleana>. In altri termini, se l'<espressione Booleana> produce TRUE nell'istruzione WHILE, dovrebbe dare FALSE in REPEAT, e viceversa.

Di seguito ci sono due semplici programmi che danno lo stesso risultato:

```
PROGRAMMA WHILEDEMO;
VAR I:INTEGER;
BEGIN
  I:=1;
  WHILE I<=5 DO
    BEGIN
      WRITELN (I);
      I:=I+1;
    END;
END.
```

```
PROGRAMMA REPEATDEMO;
VAR I:INTEGER;
BEGIN
  I:=1;
REPEAT
    WRITELN (I);
    I:=I+1;
  UNTIL I>5;
END.
```

Ciascun programma dovrebbe visualizzare una colonna dei cinque interi compresi fra 1 e 5. Notate che la prova seguente WHILE, cioè I<=5, è l'*"inverso"* o l'opposto della prova che segue UNTIL, cioè I>5.

Quale ultima illustrazione dell'istruzione REPEAT, considerate il programma SPIRALILAT. Questo programma veniva suggerito da Martin Gardner nei suoi articoli "Mathematical Games" sulla Scientific American, novembre 1973. La figura 3-10 mostra alcune delle figure disegnate dal programma e molto simili a quelle stampate nell'articolo di Gardner che erano caratterizzate da spostamenti di angolo di 90 gradi e spostamenti sempre in senso orario. Nelle figure 3-11 e 3-12 i disegni hanno in alcuni casi degli angoli con gradi diversi da 90 e spostamenti sia in senso orario che antiorario. Alcune di queste figure sono simili a quelle dell'articolo di Gardner, altre sono differenti. Tutte sono state tracciate con l'uso del programma SPIRALILAT.

Dapprima il programma vi richiede, in linea 8, un fattore di scala. La cifra da voi inserita dipenderà dal tipo di terminale video usato, alcune prove basteranno per trovare il fattore che vi permetta di disegnare delle figure di dimensioni appropriate. Il programma vi chiede poi che angolo deve essere usato. Ad ogni passo la tartaruga girerà verso sinistra (angolo positivo) secondo dei gradi ANGOLO o verso destra (angolo negativo) secondo dei gradi ANGOLO. Per ultimo il programma chiede la "SEQUENZA:", a cui risponderete con una sequenza di caratteri "D" e "S" chiusa da <RET>. Per esempio, la parte (a) della figura 3-10 è stata prodotta con la sequenza

DDD

seguita da <RET>. Il programma risponde immediatamente tracciando tre linee. La

prima linea è lunga DIMENSIONE unità, la seconda 2*DIMENSIONE, la terza 3*DIMENSIONE. Se nella sequenza sono stati inseriti più di 3 caratteri, allora verranno tracciate delle linee supplementari, ognuna delle quali sarà più lunga della precedente di DIMENSIONE unità.

```
 1: PROGRAMMA SPIRALILAT;
 2: VAR SEQ:STRING;
 3:    DIMENSIONE,ANGOLO,I:INTEGER;
 4:    CH:CHAR;
 5: BEGIN
 6:    WRITELN('SPIRALILAT');
 7:    PENCOLOR(WHITE);
 8:    WRITE('DIMENSIONE:');
 9:    READLN (DIMENSIONE);
10:    WRITE('ANGOLO:');
11:    READLN (ANGOLO);
12:    WRITE('SEQUENZA:');
13:    READLN(SEQ);
14:    REPEAT
15:       I:=1;
16:       REPEAT
17:          MOVE(DIMENSIONE*I);
18:          IF SEQ[I]='D' THEN
19:             TURN(-ANGOLO)
20:          ELSE
21:             TURN(ANGOLO);
22:          I:=I+1
23:       UNTIL I > LENGTH (SEQ);
24:       READ(CH); (* Qualsiasi tasto: usa <DEL> per*)
25:                 (* bloccare la rappresentazione*)
26:    UNTIL CH=CHR (127);(*DEL*)
27: END.
```

Dopo aver svolto la sequenza una volta, il programma si arresta in linea 24 affinchè voi inseriate un carattere qualsiasi. Se questo carattere non corrisponde a <DEL>, REPEAT in linea 14 ritorna per un altro ciclo nel quale viene eseguita di nuovo la sequenza, dall'inizio. Questa volta la sequenza comincia nel punto dove la tartaruga era stata lasciata alla fine del ciclo precedente (linee da 16 a 23). Nelle parti (a), (c), (d) e (d) la tartaruga può raggiungere il punto da cui era partita all'origine e successive ripetizioni ritracciano semplicemente le figure già disegnate sul video.

Nelle parti (b) e (f) la tartaruga uscirà lentamente dallo schermo se il ciclo interno viene ripetuto per un certo numero di volte. La parte (a) è stata ottenuta con 3 D, (b) con 4, (c) con 5, e così di seguito. Come è stato fatto per TRACCIANOME, questo programma può essere fatto terminare usando il tasto < DEL > che arresta l'istruzione REPEAT esterna.

Tutte e tre le parti della figura 3-11 sono state ottenute con questo programma usando la SEQUENZA "DDSSD" ed un ANGOLO di 120 gradi. La parte (a) è stata disegnata con un solo ciclo interno, (b) con due e (c) con tre.

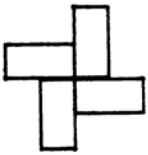
La figura 3-12 fornisce degli esempi di figure che possono essere prodotte con lo stesso semplice programma. In effetti la SEQUENZA da voi inserita rappresenta un nuovo tipo di "programma", interpretato dal programma SPIRALILAT che in questo caso funziona come se fosse un elaboratore di tipo speciale. Frequentemente i programmatori scrivono dei programmi conosciuti come *"interpreti"* che fanno sembrare l'hardware un software con sue proprie caratteristiche. Il sistema PASCAL usato nello studio di questo libro si avvale di un programma interprete abbastanza complesso applicabile ad una vasta gamma di piccoli elaboratori. Le sequenze e gli angoli usati per disegnare le figure in 3-12 sono riportati nella seguente tabella:

Parti	**Angoli**	**Sequenze**
a	90	SSDSSSDD
b	144	DDD
c	144	DDS
d	120	DSDDD
e	135	DSDDSSS

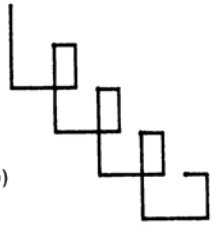
ESERCIZIO 3.4:
Disegnate con SPIRALILAT una figura diversa da quelle riportate nelle illustrazioni 3-10, 3-11, 3-12. Usate una SEQUENZA che contenga almeno 5 caratteri includendo sia "S" che "D", il vostro disegno non dovrebbe essere una semplice riproduzione di una delle figure mostrate nel libro. Progettate il disegno in modo che si ripeta dopo il primo ciclo REPEAT interno. Ricordate che la figura può essere ripetuta solo se il risultato di tutte le istruzioni TURN, dopo l'esecuzione di un certo numero di cicli interni, da ANGOLO uguale a 0 (zero) o un multiplo di 360 gradi.

ESERCIZIO 3.5:
Riscrivete i programmi REPEAT1 e SPIRALILAT sostituendo tutte le istruzioni REPEAT con WHILE, e mantenendo per il resto la stessa logica del programma qui riportato. Verificate sull'elaboratore i pro-

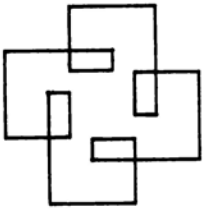
(a)



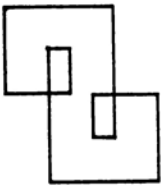
(b)



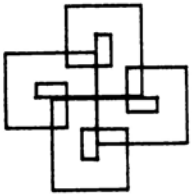
(c)



(d)



(e)



(f)

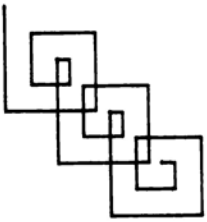
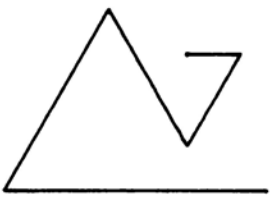
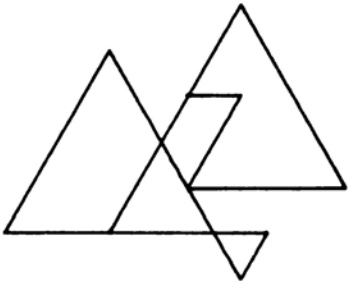


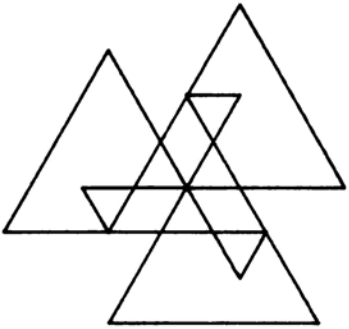
Figura 3-10



(a)



(b)



(c)

Figura 3-11

grammi così riveduti per assicurarvi che la conversione fatta sia corretta.

ESERCIZIO 3.6:
Riscrivete i programmi WHILETRA, WHILE1, WHILE 2 e TRACCIANOME sostituendo tutte le istruzioni WHILE con istruzioni REPEAT, mantenendo per il resto la stessa logica dei programmi sopra riportati. Verificate sull'elaboratore i programmi così riveduti, assicurandovi che la conversione sia stata fatta correttamente. Avvertimento: può accadere che la suddetta conversione richieda il cambiamento di posizione di una o più delle istruzioni READ affinchè la logica del programma venga rispettata.

ESERCIZIO 3.7:
Riscrivete i programmi FOR1 e POLIGONO in modo da sostituire alla istruzione WHILE l'istruzione FOR, ed i programmi FOR2 e GRAFCARTA in modo da sostituire l'istruzione FOR con REPEAT. Mantenete per il resto la stessa logica prevista dal nostro programma. Provate i nuovi programmi sull'elaboratore per assicurarvi che compiano le stesse azioni descritte nel libro.

ESERCIZIO 3.8:
Il programma REPEAT2 è stato scritto per invertire l'ordine dei caratteri contenuti nella variabile STRING S. Il programma vi richiede la battitura di una stringa qualsiasi ed è tale che sviluppi rapidamente ogni ciclo richiedendo l'inserimento di nuove stringhe. Potete facilmente verificare che all'inserimento della stringa:

INVERTIRE

il programma risponde con:

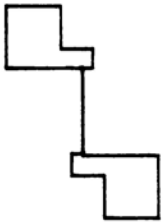
ERITREVNI

È comunque presente nel programma un errore di logica che dà un risultato abnorme qualora venga inserita la stringa:

PAROLAPARI

```
1: PROGRAMMA REPEAT2;
2: VAR S:STRING;
3:
4: PROCEDURE INVERTE;
```

(a)



(b)

(c)

(d)

(e)

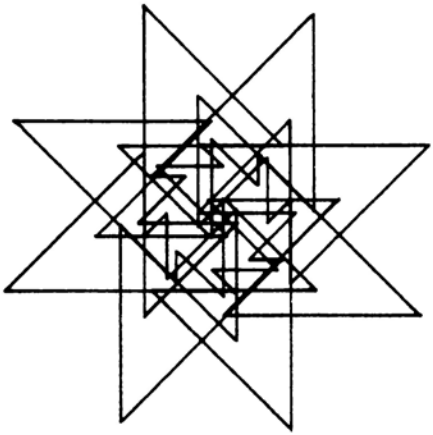


Figura 3-12

```
 5:    (*Inverte l'ordine dei caratteri in S*)
 6:  VAR NI,NF:INTEGER; (*puntatevi inizio e fine*)
 7:    SALVA:CHAR;
 8:  BEGIN
 9:    NI:=1;
10:    NF:=LENGTH(S);
11:    REPEAT
12:      (*scambia i caratteri NI e NF, muove NI e NF*)
13:      SALVA:=S[NF];
14:      S[NF]:=S[NI];
15:      S[NB]:=SALVA;
16:      NI:=NI+1
17:      NF:=NF-1
18:    UNTIL NI=NF;
19:  END (*INVERTE*);
20:
21:  BEGIN (*PROGRAMMA PRINCIPALE*)
22:    WRITELN ('BATTI QUALSIASI STRINGA SEGUITA DA <RET>');
23:    READLN(S);
24:    WHILE LENGTH(S)>0 DO
25:    BEGIN
26:      INVERTE;
27:      WRITELN(S);
28:      WRITELN;
29:      WRITELN('BATTI UN'ALTRA STRINGA');
30:      READLN(S);
31:    END;
32:  END.
```

Trovate l'errore e correggete il programma in modo che possa manipolare correttamente sia stringhe con un numero di caratteri dispari che stringhe con un numero di caratteri pari.

9. Variabili Booleane

Può essere conveniente, in certi casi, conservare il valore di una < espressione Booleana > per usi successivi. Sarebbe possibile assegnare ad una variabile intera il valore 1 se l'espressione fosse TRUE o 0 se fosse FALSE. Oppure si potrebbe assegnare ad una variabile stringa 'T' o 'F'. È comunque preferibile avere un < tipo > di variabile esplicitamente prevista per conservare il valore delle < espressioni Booleane > questo < tipo > è chiamato BOOLEAN.

Il programma BOOLDEMO considera una delle situazioni in cui è utile avere una variabile Booleana. Dapprima il programma inizializza la variabile STRING S a contenere un elenco di nomi (identificatori) separati da virgole. Il compito del programma è di *"scandire"* S, separare i nomi in essa contenuti e disporli su delle linee differenti. Questo compito è molto simile a quello svolto dal compilatore PASCAL quando separa gli identificatori dichiarati in un elenco come quello in linea 5 del programma. In questo esempio l'azione di scansione si arresta quando il programma incontra uno < spazio > bianco. Il compilatore ha bisogno di una logica più elaborata per determinare quando deve arrestare la scansione, questo perchè ogni simbolo può essere separato uno dall'altro da un numero arbitrario di spazi bianchi. Ecco le linee che dovrebbero risultare visualizzate da questo programma:

```
ALICE
BARBARA
CARLO
CARLA
LUISA
FRANCO
```

```
 1: PROGRAMMA BOOLDEMO;
 2: VAR CH:CHAR;
 3:    NOME,S:STRING;
 4:    BVIRGOLA:BOOLEAN;
 5:    NOMEP,LS,KS:INTEGER;
 6: BEGIN
 7:    S:='ALICE,BARBARA,CARLO,CARLA,LUISA,FRANCO   ';
 8:    KS:=1;
 9:    NOMEP:=1;
10:    LS:=LENGTH(S);
11:    REPEAT
12:       WHILE (S[KS]<>',') AND (S[KS]<>'') DO
13:          KS:=KS+1
14:       BVIRGOLA:=(S[KS]=',');
15:       NOME:=COPY(S,NOMEP,KS-NOMEP);
16:       WRITELN(NOME);
17:       KS:=KS+1;
18:       NOMEP:=KS;
19:    UNTIL (NOT BVIRGOLA) OR (KS>LS);
20: END.
```

La variabile KS funziona quale *"puntatore"* di caratteri in S, mentre NOMEP è posto all'inizio di un nome durante ciascuno dei cicli dell'istruzione REPEAT. Queste variabili sono inizializzate a puntare ad 'A' di 'ALICE' prima che il ciclo inizi. L'istruzione WHILE nelle linee 12 e 13 fa poi avanzare KS fino a quando non venga incontrata una virgola (",") o uno spazio bianco (' '). Al ritrovamento di uno di questi, la variabile Booleana BVIRGOLA sarà definita TRUE, in linea 14, se il carattere S[KS] è una virgola, altrimenti sarà definita FALSE. Notate che questa istruzione di assegnamento rappresenta un modo più efficiente di scrivere ciò che segue:

```
IF S[KS]=',' THEN BVIRGOLA:=TRUE ELSE BVIRGOLA:=FALSE
```

Nelle linee 17 e 18, i puntatori vengono spostati all'inizio del nome successivo nell'elenco. REPEAT terminerà in linea 19 se BVIRGOLA è FALSE o se KS è diventato superiore a LS, memorizzante LENGTH di S. Si sarebbe ottenuto lo stesso risultato usando "LENGTH(LS)" invece di "LS", ma questo avrebbe comportato diverse chiamate della procedura interna LENGTH. La chiamata di una procedura implica un maggior tempo di esecuzione di quanto non richieda il riferimento ad una semplice variabile (di <tipo> INTEGER, CHAR o BOOLEAN).

Quest'ultimo metodo risulta molto più efficiente in un programma abbastanza lungo in cui il tempo di esecuzione riveste un aspetto importante.

Per capire la ragione dell'uso della variabile Booleana BVIRGOLA, provate a rivedere il programma in modo che operi comparando i caratteri contenuti in S con una virgola in linea 19. Esistono naturalmente diversi modi per rispondere a questa richiesta, ma molti risultano essere più rozzi della tecnica qui usata.

Dopo aver introdotto l'idea di una variabile Booleana, ritorniamo ora all'<espressione Booleana> per vedere quali sono gli effetti di AND, OR, NOT e delle parentesi. Considerate la seguente *"tabella di verità"* nella quale entrambi A e B si presumono dichiarati BOOLEAN:

A	*B*	*Espressione*	*Risultato*
TRUE	TRUE	A OR B	TRUE
TRUE	TRUE	A AND B	TRUE
TRUE	FALSE	A OR B	TRUE
TRUE	FALSE	A AND B	FALSE
FALSE	TRUE	A OR B	TRUE
FALSE	TRUE	A AND B	FALSE
FALSE	FALSE	A OR B	FALSE
FALSE	FALSE	A AND B	FALSE
TRUE		NOT A	FALSE
FALSE		NOT A	TRUE
TRUE	FALSE	A AND NOT B	TRUE

Cioè: l'espressione (A OR B) è TRUE se uno dei due termini è TRUE, ma non lo è se entrambi sono FALSE. L'espressione (A AND B) è TRUE solo se entrambi i termini sono TRUE, altrimenti è FALSE. NOT seguito da un valore TRUE crea un valore FALSE, l'*"opposto"*, e viceversa.

10. Indicazioni sulle espressioni Booleane e sulle istruzioni IF

La complessità progressiva delle espressioni Booleane e delle istruzioni IF, comportante la valutazione TRUE o FALSE di un numero sempre maggiore di termini, può generare confusione. Questa sezione vi mostra alcune delle tecniche che vi possono aiutare a superare questa confusione.

La sintassi dell'< espressione Booleana >, figura 3-5, attesta l'esistenza di un ordine di precedenza simile a quello che fa precedere l'operatore della moltiplicazione ("*") a quello dell'addizione ("+") e della sottrazione ("–"). Cosicchè NOT viene eseguito prima di AND e AND precede OR. Se A=TRUE,B=FALSE,C=FALSE, e D=TRUE, allora la espressione

```
A AND NOT B OR C AND D
```

è valutata TRUE perchè A AND NOT B è valutato come TRUE. Non ha alcuna importanza che C AND D sia FALSE in quanto C è FALSE. Analogamente se X=2 e Y=1, allora

```
NOT A OR (X>Y) AND C
```

è valutato FALSE perchè NOT A è FALSE, inoltre essendo C FALSE forza (X>Y) and C FALSE, nonostante che (X>Y) sia TRUE.

Non sarete i soli a trovare confuse le due suddette espressioni!

Può essere molto utile raggruppare delle sotto-espressioni fra parentesi in modo tale da rendere più ovvio l'ordine di esecuzione. Come è dimostrato dalla sintassi, ciò che è contenuto fra parentesi *tonde* deve essere risolto per primo. Il valore trovato sostituirà quindi l'espressione contenuta fra parentesi e le parentesi stesse, quindi la risoluzione del resto dell'espressione continua. Come avviene per le espressioni aritmetiche, nelle espressioni Booleane si procede da sinistra verso destra all'interno di qualsiasi livello di *"nidificazione"* posto fra parentesi. Le espressioni precedentemente mostrate possono essere così riscritte:

```
(A AND(NOT B))OR(C AND D)
```

```
(NOT A)OR((X>Y)AND C)
```

senza che il risultato cambi. È inoltre possibile cambiare il significato della prima di queste espressioni nel seguente modo:

```
A AND ((NOT B)OR C)AND D
```

in cui le parentesi esterne sono state usate per far sì che venga eseguita l'operazione OR prima di qualsiasi delle operazioni AND.

Ritorniamo ora a considerare l'istruzione IF. In alcuni casi può essere utile *"nidificare"* diverse istruzioni IF invece di usare una unica e complicata espressione Booleana, in un'unica istruzione IF. Alcune precauzioni sono comunque necessarie. Per esempio:

```
IF A AND B THEN istruzione–1
```

equivale a

```
IF A THEN
   IF B THEN istruzione–1
      \
```

Ma

```
IF A AND B THEN istruzione–1 ELSE istruzione–2
```

non è lo stesso di

```
IF A THEN
   IF B THEN istruzione–1 ELSE istruzione–2
```

Nel primo caso l'istruzione–2 verrà eseguita se *uno dei* due termini, A o B è FALSE. Mentre nel secondo caso l'istruzione–2 verrà eseguita solo se A è TRUE e B è FALSE. Se A fosse FALSE, *nessuna* delle due istruzioni verrebbe eseguita.

Con la stessa ottica con cui vengono usate delle parentesi per raggruppare le espressioni che devono essere eseguite per prime, si può raggiungere lo stesso scopo nell'istruzione IF con l'uso di BEGIN...END.

Per esempio:

```
IF A THEN
  BEGIN
    IF B THEN istruzione–1;
  END ELSE
    istruzione–2;
```

aggiunge un'istruzione composta che altera l'effetto dell'istruzione IF nidificata. L'istruzione–2 verrà così eseguita nel caso in cui A è FALSE. L'istruzione–1 sarà eseguita solo se entrambi A e B sono TRUE. Nel caso in cui A fosse TRUE ma B fosse FALSE nessuna delle due istruzioni verrebbe eseguita. Per ovviare a una tale situazione con la logica dell'istruzione IF, si potrebbe usare:

```
IF A THEN
  BEGIN
    IF B THEN istruzione–1
         ELSE istruzione–2;
  END ELSE
    istruzione–2;
```

così da avere la certezza che una delle due istruzioni sarà comunque eseguita.

Un uso appropriato di questa applicazione dell'istruzione IF, può rendere il programma molto più veloce di quanto non sia possibile con l'utilizzo di complesse espressioni Booleane. La ragione risiede nel fatto che il sistema PASCAL sviluppa tutti i punti di una < espressione Booleana > complessa, anche quando il valore finale potrebbe essere trovato con un'analisi parziale dell'espressione stessa. Ci si può chiedere se sia preferibile questo procedimento o se invece sia possibile arrestare la valutazione una volta trovato il valore finale dell'espressione. Esistono delle ragioni per seguire sia uno che l'altro dei suddetti procedimenti.

Quando siete di fronte ad un complicato problema di decisioni, indipendentemente dal fatto che usiate espressioni Booleane o nidificazioni di istruzioni IF, è molto importante che componiate una *tabella di verità*, simile a quella usata nella sezione 9, che illustri i valori delle espressioni combinando valori Booleani con AND, OR e NOT. La tabella dovrebbe contenere un'entrata per ogni combinazione *possibile* dei valori dei componenti di una espressione. Esistono 4 combinazioni possibili di A e B, 8 combinazioni per A, B e C, 16 combinazioni di 4 variabili, e così di seguito. È preferibile analizzare l'azione svolta da IF (o WHILE o REPEAT) nel caso di *ognuna* delle singole combinazioni, prima di concludere che il programma è corretto. Ciò può risultare semplificato se voi sapete che una *stessa* azione dovrebbe risultare quando una delle variabili, B per esempio, ha valore FALSE, e che determinate altre azioni avvengono solo quando B ha valore TRUE. Risulta quindi più sicuro isolare la prova per B entro una istruzione IF semplice quale:

```
IF B THEN
  BEGIN
    altre-azioni-includenti-IF
    .. .. ..
  END ELSE
    azione-alternativa;
```

11. Note sull'indentazione

Notate che in molti degli esempi fin qui usati, abbiamo *"indentato"* il margine sinistro di una linea di ammontare dipendente da ciò che stavamo facendo. Per esempio indentiamo di due colonne tutte le istruzioni comprese fra BEGIN e END, in un'istruzione composta. Indentiamo due colonne in più per ogni linea contenuta in un'istruzione controllata da WHILE, REPEAT, FOR o IF. Dopo che le istruzioni controllate sono state completate, l'indentazione viene ridotta per lo stesso ammontare. Questo per rendere più facile la visualizzazione strutturale di un programma, che risulterebbe più difficoltosa senza l'uso dell'indentazione.

Può succedere che chi non si preoccupa abbastanza dell'indentazione, passi poi parecchio tempo a cercare dei semplici errori di logica solo perchè non gli è possibile avere una visione completa dell'organizzazione del programma. Molte complicazioni possono essere evitate osservando le seguenti regole:

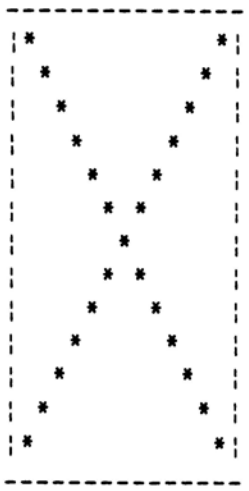
a) Usate la stessa indentazione per BEGIN e il corrispondente END. Non inserite BEGIN o END in qualsiasi punto dello schermo, ma inseriteli quale prima voce (cioè quale primo carattere non bianco) di una linea. In questo modo eviterete di disperdere gli END e BEGIN abbinati fra loro.

b) Qualora non sia possibile sistemare su una sola linea un'istruzione IF completa, inserite l'istruzione controllata da THEN e quella controllata da ELSE con la stessa indentazione. Entrambe dovrebbero essere indentate di due colonne in più verso destra, rispetto ad IF.

c) Eccetto che nel caso di un gruppo di istruzioni brevi, strettamente collegate fra loro, ed il cui ordine non abbia importanza, non dovreste mai inserire più di una istruzione per linea.

ESERCIZIO 3.9:
Scrivete e verificate un programma che visualizzi una figura geometrica come quella sotto riportata, usando i caratteri '*','—' e ':'. Suggerimento: usate istruzioni del tipo WRITE('*') o WRITE('-'), all'interno di cicli con l'uso di una qualsiasi delle istruzioni di controllo discusse in questo capitolo. Potete inoltre assegnare a una variabile S un certo numero di spazi bianchi, ed usare quindi delle istruzioni del tipo

S[N]:='*'. Dopo aver "definito" il contenuto di S, potete visualizzarlo con WRITELN (S).



ESERCIZIO 3.10:
I disegni riportati nella figura 3-13 sono stati prodotti da un piccolo programma simile ad uno studiato più avanti in questo capitolo. Le differenze fra i disegni sono il risultato dei valori inseriti in risposta alle istruzioni READLN. Scrivete un programma che possa disegnare le stesse figure e provate con il programma a disegnarne delle altre, diverse.

Problemi

PROBLEMA 3.1:
Ognuno dei seguenti programmi POLIBACO1,POLIBACO2 e POLIBACO3, contiene un errore di logica che impedisce il tracciamento di un poligono a 5 lati uguale a quello rappresentato in basso a sinistra nella figura 3-6. Trovate l'errore o gli errori di ciascun programma, e stabilite cosa avrebbe fatto il programma così come è stato scritto qui.

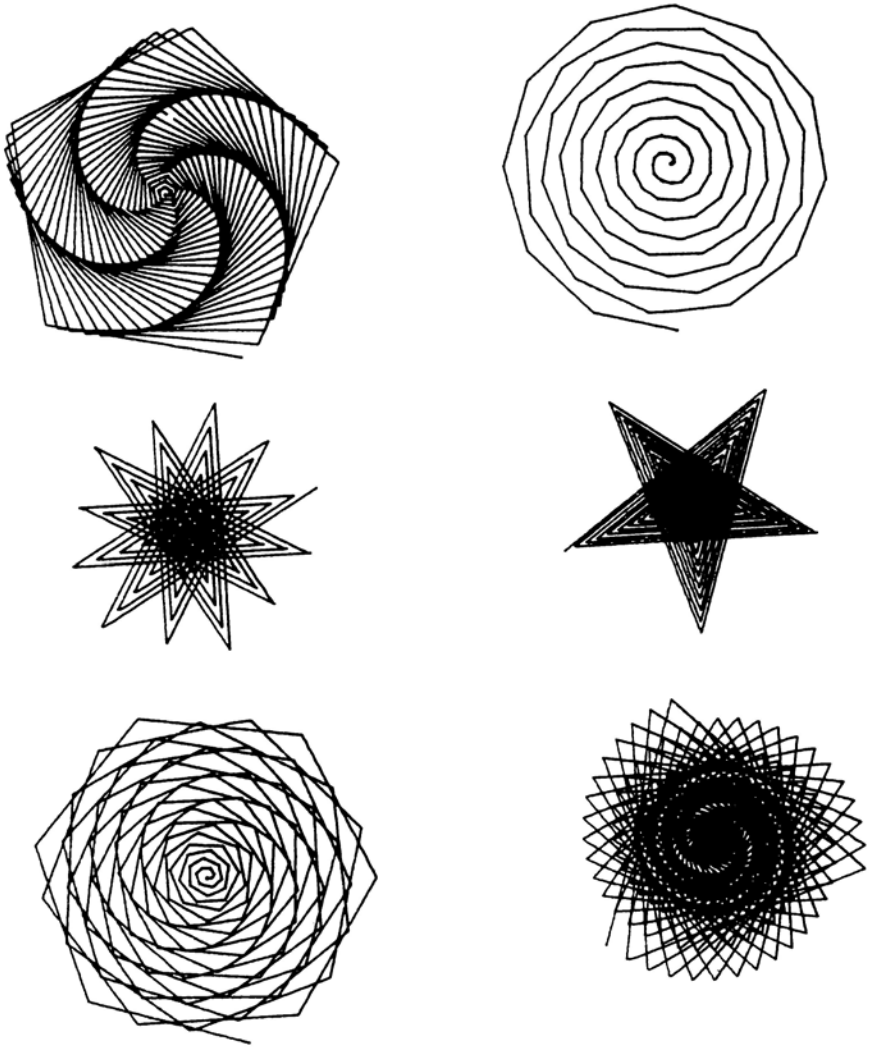


Figura 3-13

```
 1: PROGRAMMA POLIBACO1;
 2: VAR SCALA,K:INTEGER;
 3:
 4: BEGIN
 5:   SCALA:=60; (*60 per Tektronix 4006*)
 6:            (*20 per Terak; ?? per gli altri)
 7:   PENCOLOR(WHITE);
 8:   WHILE K<=5 DO
 9:   BEGIN
10:     MOVE(SCALA);
11:     TURN(72);
12:     KI=K+1;
13:   END;
14: END.
```

```
 1: PROGRAMMA POLIBACO2;
 2: (*Questo programma dovrebbe disegnare un poligono con 5 lati*)
 3: VAR SCALA,CONT:INTEGER;
 4:
 5: BEGIN
 6:   SCALA:=60; (*60 per Tektronix 4006*)
 7:          (* 20 per Terak; ?? per gli altri*)
 8:   PENCOLOR(WHITE);
 9:   CONT:=1;
10:   WHILE CONT<5 DO
11:   BEGIN
12:     MOVE(SCALA);
13:     TURN(72);
14:     CONT:=CONT+1; (*CONT è il numero di linee disegnate*)
15:   END;
16: END.
```

```
 1: PROGRAMMA POLIBACO3;
 2: (*Questo programma dovrebbe disegnare un poligono con 5 Lati*)
 3: VAR SCALA,CONT:INTEGER;
 4:
 5: BEGIN
 6:   SCALA:=60; (*60 per Tektronix 4006*)
 7:          (*20 per Terak; ?? per gli altri*)
 8:   PENCOLOR(WHITE);
 9:   CONT:=1;
10:   REPEAT
11:     MOVE(SCALA);
```

```
12:       TURN(72);
13:       CONT:=CONT+1;
14:             (*CONT è il numero di Linee disegnate*)
15:    UNTIL CONT=5;
16: END.
```

PROBLEMA 3.2:
Scrivete cosa verrà visualizzato dalle procedure VERSIONA e VERSIONB quando vengono chiamate usando i parametri riportati nella seguente tabella (una chiamata di ogni procedura per ogni linea della tabella):

X	Y	Z
1	10	100
5	5	5
10	10	10
5	8	10

```
PROCEDURE VERSIONA(X,Y,Z:INTEGER);
VAR W:INTEGER;
BEGIN
  IF X < 10 THEN
    BEGIN
      W:=1;
      IF Y > 5 THEN
        IF Z=10 THEN
          WRITELN('Y=',Y)
        ELSE
          WRITELN('Z=',Z)
    END;
  WRITELN ('W=',W, ', X=',X, ',Y=,Y, ',X=',Z);
  END (*VERSLONA*);


PROCEDURE VERSIONB(X,Y,Z:INTEGER);
VAR W:INTEGER;
BEGIN
  IF X <10 THEN
    BEGIN
      W:=1;
      IF Y > 5 THEN
        BEGIN
```

```
            IF Z = 10 THEN
              WRITELN('Y=',Y)
          END ELSE
            WRITELN('Z=',Z)
      END;
    WRITELN(W=',W, ',X=',X, ',Y=',Y, ',Z=',Z);
  END (*VERSIONB*)
```

In generale PASCAL è stato implementato in modo che l'uso del valore di una variabile alla quale non sia mai stato assegnato alcun valore, faccia terminare il programma in modo abnorme. Stabilite se questa situazione è applicabile a qualsiasi delle quattro condizioni riportate nella tabella. Modificate queste procedure in modo da evitare il sorgere di una simile situazione. (Quando la procedura è chiamata i valori vengono assegnati a ciascuno dei tre parametri).

CAPITOLO 4

AGGIUNTE SULLE PROCEDURE

1. Obiettivi

In questo capitolo verranno aggiunti dei nuovi importanti dettagli alla discussione delle Procedure. La comprensione di questi dettagli vi sarà utile per la scrittura di programmi più semplici e, contemporaneamente più corretti. I concetti considerati sono inoltre applicabili nella soluzione di problemi, al di fuori della soluzione con l'elaboratore.

1a. Apprendimento dell'uso di regole sull'*ambiente* degli identificatori per limitare le possibilità di uso erroneo delle variabili.

1b. Uso di procedure le cui dichiarazioni siano nidificate a diversi livelli.

1c. Sviluppo della capacità di progettare procedure che non necessitino di *comunicare* con le altre parti del programma più del necessario.

1d. Apprendimento dell'uso di procedure *recursive* come mezzo per semplificare certi problemi che sarebbero altrimenti troppo complessi. Contemporaneo apprendimento di quando *non* usare procedure recursive, anche se il programma che se ne otterrebbe risulterebbe più semplice.

1e. Scrittura ed uso di proprie *funzioni.*

1f. Studio della distinzione fra parametri *valore* (che abbiamo finora usato) e parametri *variabile.*

1g. Apprendimento dell'uso dell'istruzione *CASE.*

2. Premessa

Non si è ritenuto necessario o desiderabile per i problemi fin qui considerati, intro-

durre alcune delle più importanti caratteristiche di procedure ideate per la risoluzione di problemi complessi. Dette caratteristiche formano l'oggetto di questo capitolo in quanto dovranno essere usate più tardi.

Le caratteristiche qui discusse seguono due idee generali. Primo, il concetto di uso di procedure per dividere la soluzione di un problema in parti indipendenti e primitive, implica la riduzione al minimo delle informazioni che dovranno essere manipolate in comune da procedure diverse. Ciò permette la concentrazione sui dettagli di una procedura tralasciando i dettagli di variabili ed altre voci, esterne alla procedura stessa. Analogamente, mentre si sta lavorando su una parte del programma non ci si dovrebbe preoccupare degli effetti che una procedura, chiamata da un altro punto del programma, può avere su delle variabili da voi correntemente usate. Questi cambiamenti involontari in una variabile vengono chiamati *"effetti collaterali"* di una procedura.

La seconda idea generale riguarda l'uso di un dispositivo logico, chiamato *"pila"* ("stack") che semplifica il mantenimento di valori di dati che sono specifici di una particolare situazione o contesto, particolarmente quando dovete lasciare quel contesto per poi ritornarci più tardi. Una pila è un elenco di valori di dati paragonabile ad una pila di libri in una libreria, o ad una pila di barattoli in un supermercato. Come il numero di oggetti della pila aumenta, sarà certamente più facile rimuovere l'oggetto in cima prima di qualsiasi altro. Una pila logica ha la stessa proprietà, cioè l'ultimo punto aggiunto ad essa è il primo ad essere tolto. Procedure "recursive" permettono la manipolazione di pile di dati con sforzo minimo. Ecco perché la *"recursione"* è considerata uno strumento molto importante in informatica, ed in altri campi di risoluzione di problemi.

3. Ambiente (scope) degli identificatori di variabili

Considerate ora il programma AMBIENTEDEMO e le linee da esso visualizzate. Le variabili S e LIVELLO, dichiarate rispettivamente in linea 2 e 3, sono dette *"globali"* in quanto possono essere usate dall'intero programma. La variabile S, dichiarata in linea 15, è detta *"locale"* alla procedura Q, dovrà quindi essere usata solo all'interno di quella procedura. Nonostante abbiano lo stesso nome, le due dichiarazioni di S si riferiscono a due celle, completamente diverse della memoria dell'elaboratore, ed il compilatore non le confonde fra loro.

Può apparire strano che si usi uno stesso <identificatore> per fare riferimento a due <variabili> differenti, ma esistono spesso delle importanti ragioni per fare questo. Questo particolare programma vuole solo illustrare il funzionamento delle regole, risulta difficile elaborare un caso evidente del funzionamento di questa regola particolare in un programma di così piccole dimensioni.

```
 1: PROGRAMMA AMBIENTEDEMO;
 2: VAR S:STRING;
 3:    LIVELLO:INTEGER;
 4:
 5: PROCEDURE P;
 6: BEGIN
 7:    LIVELLO:=LIVELLO+1
 8:    WRITELN(' ':LIVELLO*2,'LANCIO P');
 9:    WRITELN(' ':LIVELLO*2,S);
10:    WRITELN(' ':LIVELLO*2,'FINE P');
11:    LIVELLO:=LIVELLO-1
12: END (*P*);
13:
14: PROCEDURE Q;
15: VAR S:STRING;
16: BEGIN
17:    LIVELLO:=LIVELLO+1;
18:    WRITELN(' ':LIVELLO*2,'LANCIO Q');
19:    S:='SON QUI!';
20:    WRITELN(' ':LIVELLO*2,S);
21:    P;
22:    WRITELN(' ':LIVELLO*2,'FINE Q');
23:    LIVELLO:=LIVELLO-1;
24: END (*Q*);
25:
26: PROCEDURE R;
27: BEGIN
28:    LIVELLO:=LIVELLO+1;
29:    WRITELN(' ':LIVELLO*2,'LANCIO R');
30:    S:='GUARDAMI ORA!';
31:    WRITELN(' ':LIVELLO*2, S);
32:    WRITELN(' ':LIVELLO*2, 'FINE R');
33:    LIVELLO:=LIVELLO-1;
34: END (*R*);
35:
36: BEGIN (*PROGRAMMA PRINCIPALE*)
37:    S:='PROGRAMMA PRINCIPALE';
38:    LIVELLO:=0;
39:    P; WRITELN(S);
40:    Q; WRITELN(S);
41:    R; WRITELN(S);
42: END.
```

```
 1: Visualizzazioni associate ad AMBIENTEDEMO
 2:
 3:    LANCIO P
 4:    PROGRAMMA PRINCIPALE
 5:    FINE P
 6: PROGRAMMA PRINCIPALE
 7:    LANCIO Q
 8:    SON QUI!
 09:       LANCIO P
10:       PROGRAMMA PRINCIPALE
11:       FINE P
12:    FINE Q
13: PROGRAMMA PRINCIPALE
14:    LANCIO R
15:    GUARDAMI ORA!
16:    FINE R
17: GUARDAMI ORA!
```

Se vi state concentrando sui dettagli di una procedura, quale è Q che non prevede alcun riferimento ad una variabile globale, quale S, allora vi saranno risparmiati parecchi sforzi in quanto non dovrete preoccuparvi della confusione che potrebbe insorgere fra nomi usati localmente nella procedura e nomi usati altrove nel programma.

Fate ora riferimento a ciò che viene visualizzato da AMBIENTEDEMO per capire le implicazioni di questa regola. Le linee comprese fra 3 e 5 sono prodotte dalla chiamata di P nella linea 39 del programma. La variabile globale LIVELLO è usata per illustrare, servendosi dell'indentazione, il numero di procedure in esecuzione in qualsiasi momento. In questo caso l'indentazione è di un livello (2 colonne) per mostrare che P è la sola procedura attualmente attiva. L'indentazione è ottenuta aggiungendo 1 a LIVELLO immediatamente dopo l'ingresso di ciascuna procedura, e sottraendo 1 da LIVELLO quale ultima azione prima che la procedura si concluda. Nelle procedure ciascuna delle istruzioni WRITE visualizza 2*LIVELLO spazi bianchi prima del suo messaggio scritto. La linea visualizzata 6 è prodotta dall'istruzione WRITELN in linea 39 del programma, ed è visualizzata senza indentazione per rappresentare LIVELLO=0, cioè nessuna procedura è al momento in esecuzione.

In linea 40 del programma viene chiamata Q. Questo risulta dalle due linee visualizzate dalle istruzione WRITELN in linea 18 e 20 del programma. In linea 20 è visualizzato "SON QUI!", in quanto questo è il valore assegnato ad S in linea 19. La S qui considerata è locale rispetto alla procedura, essendo stata dichiarata in linea 15. Non è uguale alla S usata dappertutto nel programma, *eccetto* che all'interno della procedura Q. In tal modo quando chiamiamo P in linea 21, questa procedura visualizza e-

sattamente le stesse tre linee come aveva fatto prima, ad eccezione dell'indentazione che è stata aumentata di un livello. In particolare il contenuto della linea visualizzata 10 è "PROGRAMMA PRINCIPALE" mostrante che il valore della variabile globale S non è stato cambiato. Ciò è di nuovo dimostrato dopo che Q termina visualizzando la linea 12, perché WRITELN in linea 40 del programma produce "PROGRAMMA PRINCIPALE" nella linea visualizzata 13.

R viene ora chiamata in linea 41 del programma. Il contenuto di R è molto simile al contenuto di P, eccetto che per il nuovo valore assegnato ad S in linea 30. Cosicchè la procedura R visualizza "GUARDAMI ORA!", quale valore della variabile globale S, in linea 15. Poiché R non è presente nessuna variabile globale S quando traduce le linee 30 e 31 di R. Questo processo ha così cambiato in modo permanente il valore della variabile globale S. Per questo dopo che l'ultima istruzione del programma è stata eseguita, cioè WRITELN in linea 41, si avrà di nuovo la visualizzazione di "GUARDAMI ORA!".

Per visualizzare il funzionamento della regola e l'azione svolta dal compilatore, fate riferimento al *"diagramma finestra"* in figura 4-1. Questo diagramma costituisce una rappresentazione semplificata del programma AMBIENTEDEMO. Ogni scatola chiusa, o "finestra", corrisponde a ciascun < blocco > del programma. La finestra più grande corrisponde al programma principale e racchiude tutte le altre finestre, così come la parte del programma principale posta in fondo al diagramma. Nessuna altra finestra ne racchiude altre. Queste finestre permettono di guardare in una sola direzione. Nella procedura P, ad esempio, la comparsa dell'identificatore S fa sì che il compilatore debba guardare all'esterno attraverso la finestra di P al fine di "vedere" l'identificatore globale S. Stessa cosa avviene per R. Nel caso in cui esistesse un unico identificatore locale dichiarato nella procedura P, sarebbe necessario guardare all'interno attraverso la finestra di P per poter vedere quell'identificatore da qualsiasi altro punto del programma. Ciò però non ci è permesso in quanto la finestra è fatta con un "vetro" a una direzione.

Nella procedura Q, il riferimento ad S è soddisfatto senza bisogno di guardare attraverso nessuna altra finestra in quanto è già presente una variabile locale dichiarata con questo stesso identificatore. Il compilatore controlla sempre il proprio elenco alla ricerca di una possibile dichiarazione locale di un identificatore, prima di guardare attraverso la finestra di un blocco per trovare una dichiarazione esterna dello stesso identificatore. Analogamente, il riferimento ad S nel programma principale conduce alla dichiarazione globale di S senza bisogno di guardare attraverso alcuna finestra.

Nella procedura Q viene chiamata la procedura P. Il compilatore cerca dapprima localmente l'identificatore P. Non trovandolo guarda attraverso la finestra di Q e scopre che l'identificatore P è stato dichiarato (quale nome di una procedura) nel < blocco > del programma principale.

AMBIENTEDEMO

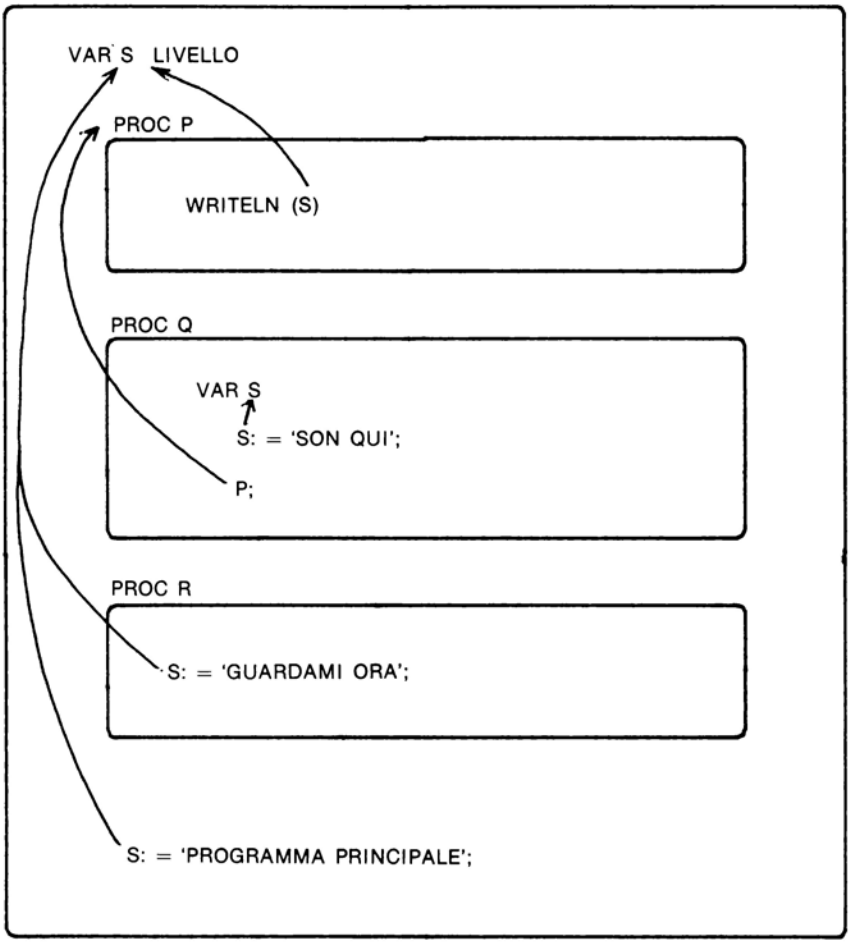


Figura 4-1

Quale ultima considerazione riguardante l'operazione del compilatore, osservate che un solo *"passaggio"* che parte all'inizio del programma e termina al raggiungimento di END., viene fatto sul vostro programma. Cosicchè al momento in cui la procedura P viene tradotta, gli identificatori delle procedure Q e R sono ancora ignoti al compilatore. Non è quindi possibile chiamare R o Q dall'interno di P, almeno fino a quando non sia stato introdotto qualcosa che possa risolvere questa situazione.

In alcuni casi può risultare conveniente avere la possibilità di chiamare una procedura in un punto del programma precedente quello in cui la procedura stessa viene dichiarata. Ciò può essere ottenuto con una dichiarazione *"FORWARD"*. Per esempio, per chiamare R dall'interno di P in AMBIENTEDEMO, senza dover anticipare nel programma la dichiarazione di R, inserite in linea 4 le seguenti linee:

```
PROCEDURE R; FORWARD;
```

Questa linea di dichiarazione FORWARD dovrebbe contenere ogni dichiarazione di parametri per la procedura, e l'elenco di parametri dovrebbe essere omesso dalla linea di dichiarazione della procedura stessa.

ESERCIZIO 4.1:
Analizzate il cambiamento prodotto nelle linee visualizzate da AMBIENTEDEMO, dalla rimozione della dichiarazione di variabile nella linea 15 del programma. Se necessario fate girare il programma così riveduto sull'elaboratore per vedere quello che succede. Spiegate poi il motivo dei cambiamenti avvenuti.

ESERCIZIO 4.2:
Provate ad usare la dichiarazione FORWARD precedentemente descritta, chiamando la procedura R dall'interno di P. Dopo averne dimostrato il corretto funzionamento sull'elaboratore, provate con la stessa tattica a chiamare la procedura Q dall'interno di P. Che tipo di problema incontrate? In che modo potreste evitare di perderne il controllo?

4. Procedure nidificate

Come implica la sintassi per <blocco>, illustrata nella figura 2-7, è possibile dichiarare una procedura locale all'interno di un'altra procedura. Questo procedimento prende il nome di *"nidificazione"*. Il programma tipo NIDDEMO è un'illustrazione sotto forma di programma scritto ad hoc per mostrare il funzionamento delle regole di nidificazione. Le linee visualizzate da questo programma sono riportate separatamente.

Come nell'illustrazione del programma AMBIENTEDEMO, la variabile LIVELLO è usata come traccia del numero di procedure attive in un dato momento. In NIDDEMO, LIVELLO prende la forma di un parametro unico usato da ciascuna procedura. In tal modo ogni volta che LIVELLO è presente nell'intestazione di dichiarazione di una procedura, stabilisce una nuova variabile locale alla procedura, che è in effetti un parametro il cui nome è uguale a quello di numerosi altri parametri. Ogni volta che una procedura viene chiamata dall'interno di un'altra, il parametro effettivo usatto diventa (LIVELLO+1). Conseguentemente ogni volta che una procedura viene lanciata, il valore di LIVELLO di quella procedura è superiore di 1 al valore di LIVELLO della procedura chiamante. La procedura PUNTINI è qui usata a dimostrazione di quanto sopra, visualizzando un punto in più per ogni procedura attivata.

```
 1: PROGRAMMA NIDDEMO;
 2:
 3: PROCEDURE PUNTINI(N:INTEGER);
 4: VAR I:INTEGER;
 5: BEGIN
 6:    FOR I:=1 TO N DO WRITE ('.');
 7: END(*PUNTINI*);
 8:
 9: PROCEDURE P1(LIVELLO:INTEGER);
10: BEGIN
11:    PUNTINI(LIVELLO);
12:    WRITELN('P1 IN ESECUZIONE');
13: END(*P1*);
14:
15: PROCEDURE P2(LIVELLO:INTEGER);
16:    PROCEDURE P2A(LIVELLO:INTEGER);
17:    PROCEDURE P2A1(LIVELLO:INTEGER);
18:    BEGIN
19:       PUNTINI(LIVELLO); WRITELN('P2A1 IN ESECUZIONE');
20:    END(*P2A1*);
21:
22:    BEGIN(*P2A*)
23:    PUNTINI(LIVELLO); WRITELN('LANCIO P2A');
24:    P1(LIVELLO+1);
25:    P2A1(LIVELLO+1);
26:    PUNTINI(LIVELLO); WRITELN('FINE P2A');
27:    END(*P2A*);
28:
29: BEGIN(*P2*)
30:    PUNTINI(LIVELLO); WRITELN('LANCIO P2');
```

```
31:    P1(LIVELLO+1);
32:    P2A(LIVELLO+1);
33:    PUNTINI(LIVELLO); WRITELN('FINE P2');
34: END(*P2*);
35:
36: BEGIN(*PROGRAMMA PRINCIPALE*);
37:    WRITELN('PROGRAMMA PRINCIPALE');
38:    P1(1);
39:    WRITELN('PROGRAMMA PRINCIPALE DOPO P1');
40:    P2(1);
41:    WRITELN('PROGRAMMA PRINCIPALE DOPO P2');
42:       (*NON È LECITO CHIAMARE P2A DA QUI*)
43: END.
```

```
 1: Visualizzazione associata a NIDDEMO
 2:
 3: PROGRAMMA PRINCIPALE
 4: . P1 IN ESECUZIONE
 5: PROGRAMMA PRINCIPALE DOPO P1
 6: . LANCIO P2
 7: . . P1 IN ESECUZIONE
 8: . . LANCIO P2A
 9: . . . P1 IN ESECUZIONE
10: . . . P2A1 IN ESECUZIONE
11: . . FINE P2A
12: . FINE P2
13: PROGRAMMA PRINCIPALE DOPO P2
```

Come esempio considerate P1 chiamata in linea 31 del programma e producente la visualizzazione in linea 7 di "P1 IN ESECUZIONE", con due livelli di indentazione. Il valore di LIVELLO all'interno di P2 è 1, com'è mostrato nella linea 6 dello schermo. Così il valore del parametro effettivo in P1, linea 31 è LIVELLO+1, cioè 2, conducente ai due livelli di indentazione in linea 7.

In linea 32, P2 chiama P2A (dove "A" significa che P2A è la prima procedura ancillare all'interno di P2) che produce le linee visualizzate da 8 a 11. P2A chiama di nuovo P1 in linea 24, questa volta il valore di LIVELLO passato a P1 è 3, provocando i tre livelli di indentazione in linea 9. Lo stesso avviene in linea 10, nonostante che P2A1 sia dichiarato come locale all'interno di P2A.

In figura 4-2 potete trovare il diagramma a finestre illustrante NIDDEMO. Questo diagramma mostra che la regola di guardare solo all'esterno, attraverso una finestra,

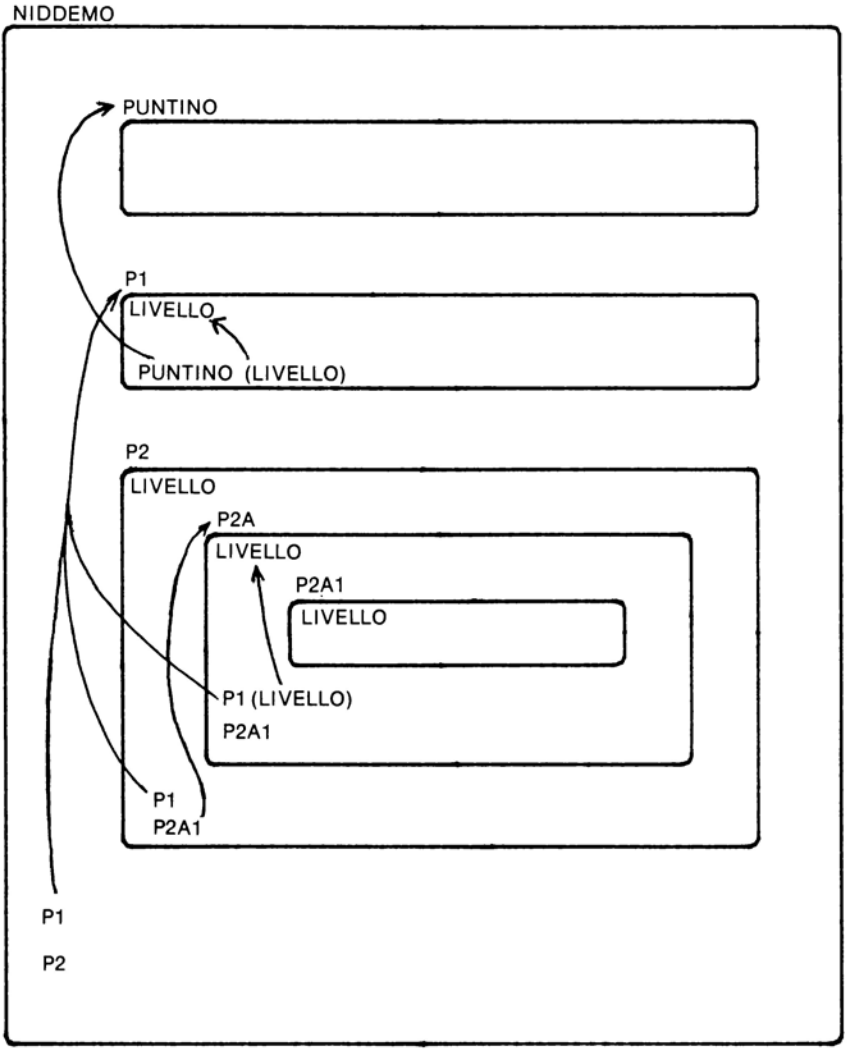

NIDDEMO
PUNTINO
P1
LIVELLO
PUNTINO (LIVELLO)
P2
LIVELLO
P2A
LIVELLO
P2A1
LIVELLO
P1 (LIVELLO)
P2A1
P1
P2A1
P1
P2

Figura 4-2

si applica nel caso di identificatori di procedure così come nel caso di variabili. Nel caso di parametri LIVELLO, ciò avviene all'interno della finestra della procedura nella cui intestazione è stato dichiarato. Le regole riguardanti l'ambiente degli identificatori di parametri sono le stesse che si applicano alle variabili dichiarate dopo VAR all'interno del <blocco> appartenente alla stessa procedura. Ciascuno dei parametri contenuto in questo programma è in realtà una variabile locale con la particolare caratteristica che il proprio valore è inizializzato dall'istruzione nella quale viene chiamata la procedura associata. In seguito in questo capitolo, parleremo di un altro tipo di parametro avente caratteristiche diverse.

Notate che le linee 8 e 11 di ciò che viene visualizzato hanno la stessa indentazione, questo perché LIVELLO ha valore 2 sia in linea 23 che in linea 26 del programma. Nel frattempo P1 e P2A1 sono stati chiamati rispettivamente in linea 24 e 25. La linea visualizzata 11 mostra che il valore di LIVELLO locale a P2A è rimasto invariato mentre le altre procedure stavano operando, agendo così allo stesso modo di una qualsiasi variabile locale. Durante l'attivazione di P1 non è possibile fare riferimento a nessuna variabile dichiarata locale a P2A quale procedura chiamante, a causa della finestra a via unica. Durante l'attivazione di P2A1 sarebbe invece stato possibile fare riferimento ad una variabile in P2A, P2 o ad una variabile globale del programma principale, questo grazie alla possibilità di guardare all'esterno attraverso una o più finestre.

Studiate ora attentamente le linee visualizzate da NIDDEMO ed accertatevi di essere in grado di capire perchè una linea venga visualizzata in un dato momento e perché venga fatta una data indentazione. La comprensione di questa dimostrazione è essenziale per capire il resto del capitolo.

ESERCIZIO 4.3:
Aggiungete a ciascuna delle procedure del programma NIDDEMO delle variabili STRING locali, dando loro dei nomi associabili a quelli delle procedure corrispondenti. Usate ad esempio S2 per la variabile locale alla procedura P2, S2A per la procedura P2A e così di seguito. Dichiarate inoltre una variabile globale S.

Aggiungete ora una istruzione a ciascun <blocco> assegnando un valore unico a ciascuna variabile che può essere vista da quel <blocco>. All'inizio di ogni <blocco> aggiungete delle istruzioni che visualizzino i valori correnti di tutte le variabili STRING visibili da quel <blocco> AND alle quali è già stato assegnato un valore. Assegnando a ciascuna variabile una breve stringa, vi sarà possibile visualizzare su una sola linea tutti i valori visibili ad un punto determinato del programma, ciò che vi permetterà di vedere gli sviluppi prodotti dai cambiamenti introdotti. Analizzate il programma e annotate i valori effettivamente visualizzati.

5. Studio particolare — Uso di procedure nidificate

Considerate la figura 4-3a rappresentante una semplice "pianta" con due fiori, disegnata dalla tartaruga. Notate che i fiori sono di dimensioni diverse e che allo stelo sono attaccate due foglie. Prima di continuare nella lettura, provate a definire una strategia che vi permetta la scrittura di un programma semplice che disegni questa figura.

Considerato che i due fiori sono identici, eccetto che per la dimensione, un'ovvia semplificazione potrebbe essere quella di scrivere una procedura che disegni un fiore. Un'altra procedura potrebbe essere progettata per disegnare le foglie. Infine il programma potrebbe disegnare lo stelo e chiamare le procedure appropriate nei punti dove si devono attaccare i fiori e le foglie. Fermatevi di nuovo e considerate in che modo potreste progettare la procedura che disegna il fiore.

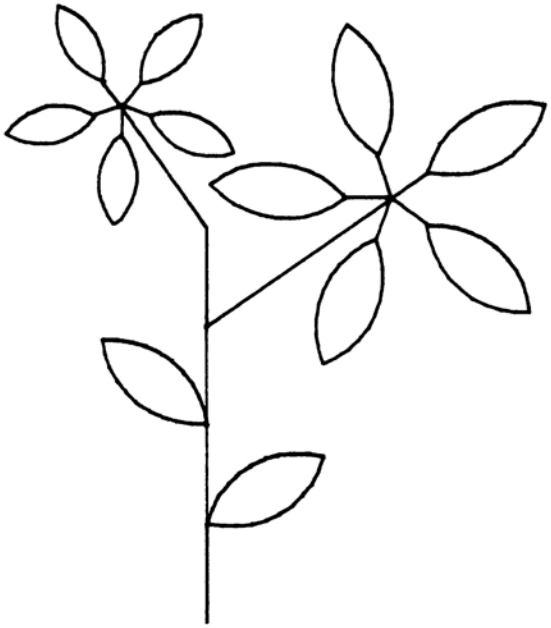
Concentriamoci sul problema del fiore analizzando la figura 4-3b. Notate che ciascun petalo sporge dallo stelo principale attaccato ad un altro stelo corto e diritto. Inoltre ognuno dei petali sembra essere formato da due lati uguali, simili a due archi (porzione di un cerchio). Questo ci suggerisce l'idea di una procedura che disegni un arco. Nel nostro caso un arco di 90 gradi, dieci passi di 9 gradi ciascuno, sarebbe sufficiente per ottenere l'aspetto desiderato. L'effetto è simile a quello prodotto dal cerchio disegnato dal programma POLIGONO nel capitolo 3, qui però sono necessari un quarto di passi per disegnare un quarto di cerchio. Fermatevi di nuovo e fate delle note sul possibile aspetto del programma che disegna il fiore.

Riferitevi ora al programma FIORE, che ha disegnato la figura 4-3b, ed è stato progettato seguendo le indicazioni sopra descritte. Notate che ARCO è una procedura locale a PETALO In quanto è usata esclusivamenete da PETALO. ARCO può fare riferimento a DIMENS, locale a PETALO, visto che quella variabile (parametro) è locale ad un blocco chiuso. Se ARCO fosse stata dichiarata all'esterno di PETALO, sarebbe stato necessario assegnarle un parametro DIMENS.

PETALO a sua volta è locale a BOCCIOLO in quanto è chiamato solo da BOCCIOLO. BOCCIOLO chiama PETALO cinque volte, muovendosi ogni volta di 5 unità per formare lo stelo del petalo. In questa illustrazione SCALA rappresenta una variabile globale e non un parametro di BOCCIOLO. Se il programma dovesse essere scritto quale procedura per disegnare un fiore, avremmo probabilmente progettato SCALA come parametro. Gli scopi principali del programma sono in questo caso, di inizializzare SCALA e di disegnare lo stelo principale del fiore.

Notate come a livello di ciascuna procedura sia possibile ignorare quasi completamente i dettagli delle altre procedure. Avendo definito una strategia globale di scrittu-

(a)



(b)

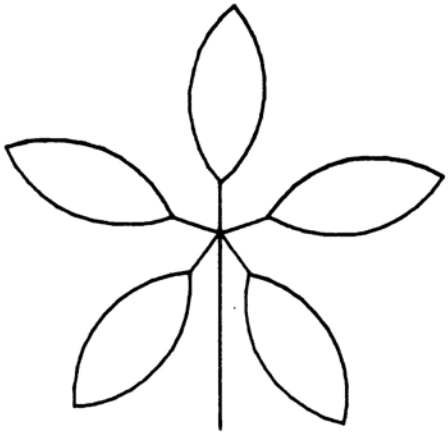


Figura 4-3

```
 1: PROGRAMMA FIORE;
 2: VAR SCALA:INTEGER;
 3:
 4: PROCEDURE BOCCIOLO;
 5: VAR I:INTEGER;
 6:
 7:   PROCEDURE PETALO(DIMENS:INTEGER);
 8:     PROCEDURE ARCO;
 9:     VAR I:INTEGER;
10:     BEGIN
11:       PENCOLOR(WHITE);
12:       FOR I:=1 TO 10 DO
13:       BEGIN
14:       MOVE(DIMENS*SCALA);
15:       TURN(9);
16:       END;
17:       PENCOLOR(NONE);
18:     END (*ARCO*);
19:
20:   BEGIN (*PETALO*);
21:     TURN(-45);
22:     ARCO;
23:     TURN(90);
24:     ARCO;
25:     TURN(135);
26:   END (*PETALO*);
27:
28: BEGIN (*BOCCIOLO*);
29:   FOR I:=0 TO 4 DO
30:   BEGIN
31:     PENCOLOR(WHITE);
32:     MOVE(5*SCALA);
33:     PETALO(2);
34:     MOVE(-5*SCALA);
35:     TURN(72);
36:   END;
37: END (*BOCCIOLO*);
38:
39: BEGIN (*PROGRAMMA PRINCIPALE*)
40:   SCALA:=12; (*120 PER TEKTRONIX 4006; *)
41:            (*40 PER TERAK; ?? PER ALTRI*)
42:   PENCOLOR(WHITE);
43:   MOVETO(0,-20*SCALA);
```

```
44:     MOVETO(0,0);
45:     TURN(90);
46:     BOCCIOLO;
47: END.
```

ra del programma, possiamo scriverne la parte principale includendo semplicemente una chiamata di BOCCIOLO pur ignorando per il momento, il funzionamento interno di BOCCIOLO.

Successivamente possiamo dichiarare BOCCIOLO, includendo una chiamata di PETALO come in linea 33. L'unico dettaglio da riportare in PETALO da BOCCIOLO è il problema della dimensione del petalo da disegnare, che decidiamo di comunicare alla procedura sotto forma di parametri.

Concentriamoci ora su PETALO, costituito principalmente da due chiamate di ARCO. In quanto l'arco copre 90 gradi, è necessario girare la tartaruga di —45 gradi (verso destra in questo caso) al fine che il petalo risulti bilanciato sulla punta dello stelo. Finito il primo arco, un giro di 90 gradi servirà a posizionare simmetricamente la tartaruga per disegnare l'altro arco. Infine, sarà necessario un giro di 135 gradi, in linea 25, per riportare la tartaruga nella sua posizione originaria. Questo è necessario per evitare complicate registrazioni nella procedura chiamante (BOCCIOLO). Per riportare la tartaruga nella sua direzione originaria dobbiamo semplicemente assicurarci che la somma di tutti i giri sia 0 (zero) gradi o un multiplo intero di 360 gradi. Nel nostro caso abbiamo deciso che arco userà 90 gradi per ogni chiamata, cosicchè due volte 90, o 180 gradi, dovrà essere aggiunto alla somma dei giri inclusi in PETALO.

Come ultimo passo dobbiamo completare i dettagli di ARCO. È già stato notato che ARCO assomiglia a POLIGONO in quanto disegna una sequenza di piccoli segmenti separati da brevi giri.

Notate che la variabile semplice I è usata sia in ARCO che in BOCCIOLO, ma con scopi completamente distinti. In quanto I è dichiarata locale a ciascuna delle procedure, separatamente, non dobbiamo preoccuparci delle possibili interferenze in BOCCIOLO provocate dall'uso di I in ARCO. Sarebbe stato possibile, per evitare qualsiasi interferenza, usare due identificatori diversi. Comunque, in questo caso, la via più "pigra" è la migliore. Assicurandoci che la variabile di controllo dell'istruzione FOR è stata dichiarata localmente nel <blocco> dove questa appare, possiamo essere certi di non incorrere in nessuna interferenza. Così non avremo bisogno di controllare eventuali interferenze mentre stiamo lavorando su altri <blocchi> del programma.

Quale regola generale dovreste abituarvi, in presenza di procedure nidificate, a dichiarare delle variabili il più possibile locali. In alcuni casi la stessa variabile verrà usata per scopi simili o identici a livelli differenti della nidificazione, come succede per SCALA nell'esempio appena considerato. Fino a quando non vengono assegnati dei nuovi valori alla variabile, dall'interno delle diverse procedure, possiamo tranquillamente inserirla nel < blocco > che racchiude tutte le procedure che usano quella variabile.

Potreste però incorrere in serie difficoltà nella correzione di un programma, qualora decideste di prendere una scorciatoia usando la stessa variabile, dichiarata ad un livello chiuso, per scopi diversi in procedure diverse. Cosicchè la procedura A può aver assegnato un determinato valore alla variabile, mentre C può avergliene assegnato uno diverso. Nel caso in cui la procedura B usasse il valore di quella variabile dovrete dapprima sapere se quel valore gli è stato assegnato da A o C, purchè il programma abbia senso. Spesso succede che un programmatore assegni un valore ad una variabile globale, X per esempio, in una procedura A e usi quel valore nella procedura B. Più tardi diventa necessario dichiarare una procedura C per un nuovo scopo e la "convenienza" porta all'uso di X per un nuovo compito all'interno di C. Il programmatore presume che, seguendo l'assegnamento a X all'interno di A, B sarà sempre chiamata prima di C. In questo modo il programmatore presume che non esistano problemi nel ri-usare la variabile ed evitare il piccolo sforzo necessario per fare una nuova dichiarazione.

Sfortunatamente succede spesso che si debba aggiungere una nuova funzionalità in stadi successivi del programma. Uno di questi stadi può essere raggiunto dopo che il programmatore ha già dimenticato che X viene usata per due scopi differenti. La nuova funzionalità potrebbe richiedere una chiamata di C immediatamente dopo A e prima della chiamata di B. A questo punto a X è stato assegnato un valore che non ha rilevanza alcuna per B, ed il programma non girerà correttamente.

MORALE: usate ognuna delle variabili dichiarate per un solo scopo, e mantenetela il più possibile locale al < blocco > dove viene usata. Questo è ciò che intendiamo quando diciamo che la "*comunicazione*" fra < blocchi > dovrebbe essere minima. Per le stesse ragioni si assegneranno dei compiti alle proprie procedure in modo tale da mantenere il numero di parametri necessari il più basso possibile. Onde evitare qualsiasi malinteso aggiungiamo subito che questa norma sulla minimizzazione della comunicazione fra < blocchi > , non dovrebbe spingervi ad inserire tutta la logica di un programma in un unico < blocco > gigante. La ragione di una suddivisione del programma in blocchi separati sta nell'evitare la necessità di mantenere punti di comunicazione fra parti del programma ben distinte aventi compiti concettualmente distinti.

ESERCIZIO 4.4:
Completate il progetto del programma affinchè disegni una "pianta" completa, come nell'illustrazione 4-3a. Notate che la richiesta di disegnare delle foglie sporgenti dallo stelo principale, richiede una riscrittura dei rapporti di nidificazione delle procedure, qualora cominciate con il programma FIORE quale procedura. Non risolvete questo problema con l'uso di due procedure diverse che disegnino i petali! Uno degli scopi principali di questo esercizio è di mostrarvi che la progettazione di programmi vi può richiedere, occasionalmente, un certo lavoro di ri-scrittura, anche dopo aver raggiunto un avanzato livello nella risoluzione del problema.

6. Dichiarazione delle proprie Funzioni

Come è già stato fatto notare in precedenza, in PASCAL una "*funzione*" è in realtà un tipo particolare di Procedura. Una procedura viene chiamata dando il suo nome in una istruzione separata, contenente quel nome più un elenco di parametri. Si chiama una funzione usando il suo identificatore all'interno di un'espressione, come se si trattasse di una normale variabile eccetto che deve essere incluso un qualsiasi elenco di parametri. Quando viene chiamata, una funzione esegue i suoi calcoli esattamente come fa una procedura, ma la funzione "*restituisce*" anche un valore che va a sostituirsi all'identificatore della funzione quando questa ha terminato l'esecuzione.

Con queste premesse non ci dovrebbero essere difficoltà per capire il programma CONTAPAROLE. Il programma vi richiede la battitura (mediante la tastiera) di una frase di una linea. Poi presume che il numero di parole della frase possa essere calcolato sommando gli spazi bianchi della frase più 1 per l'ultima parola. Naturalmente potrebbe succedere che voi usiate più di uno spazio per separare le parole fra di loro; in questo caso il risultato calcolato dal programma risulterebbe falso. Questa è una complicazione che può essere per il momento evitata, allo scopo di illustrare il funzionamento delle funzioni.

```
1: PROGRAMMA CONTAPAROLE;
2: VAR S:STRING;
3:
4: FUNCTION CONTEM(S:STRING):INTEGER;
5: VAR CONT,K:INTEGER;
6: BEGIN
7:    CONT:=0;
8:    FOR K:=1 TO LENGTH(S) DO
9:       IF S [K]=' ' THEN CONT:=CONT+1;
```

```
10:    CONTEM:=CONT+1;
11: END (*CONT*)
12:
13: BEGIN (*PROGRAMMA PRINCIPALE*)
14:    WRITELN ('CONTAPAROLE');
15:    WRITELN ('BATTI UNA QUALSIASI FRASE DI UNA RIGA');
16:    READLN(S);
17:    WHILE LENGTH(S) >0 DO
18:       BEGIN
19:          WRITELN(CONTEM(S), 'PAROLE');
20:          WRITELN;
21:          WRITELN('BATTINE ANCORA UNA');
22:          READLN(S);
23:       END;
24. END.
```

La dichiarazione della funzione CONTEM inizia in linea 4 del programma. Questa linea differisce dalla linea di intestazione della dichiarazione di una procedura per due aspetti. Primo, la parola riservata FUNCTION compare al posto di PROCEDURE. Secondo, il < tipo > di valore che sarà restituito dalla funzione deve essere posto dopo l'elenco di parametri. Come è mostrato nella figura 4-4, la sintassi specifica che un due-punti (":") deve comparire fra l'elenco parametri e il < tipo identificatore >. Il < tipo identificatore > deve riferirsi a un tipo *semplice*, cioè contenente un unico valore, quale INTEGER, CHAR o BOOLEAN. Non è possibile restituire una STRING quale valore di una funzione. Una funzione o una procedura possono restituire il valore di una STRING per mezzo di una *variabile "parametro"*, ma parleremo di questo in una delle prossime sezioni di questo capitolo.

Oltre che nella linea di intestazione una funzione differisce da una procedura anche per la richiesta che un valore sia assegnato all'identificatore della funzione all'interno del < blocco > appartenente alla funzione. Questo è il modo con cui il valore che deve essere "*restituito*" dalla funzione sarà reso disponibile all'interno della espressione dove la funzione è chiamata. Per esempio, a CONTEM viene assegnato un valore in linea 10 del programma. In questo contesto, CONTEM appare usato come se fosse un normale identificatore, ma questa apparenza è ingannevole. All'interno del < blocco > proprio a una funzione, non si può mettere l'identificatore della funzione stessa alla destra dell'operatore di assegnamento, o in qualsiasi altro punto dove un'espressione risulterebbe appropriata, senza far sì che la funzione chiami se stessa! Questo è chiamato "recursione", di cui discuteremo più tardi in questo capitolo. Non esistono problemi connessi all'assegnamento di valori all'identificatore di una funzione in punti diversi all'interno del < blocco > della funzione. Ciò può essere fatto all'interno di un complesso di istruzioni IF nidificate per assegnare dei valori differenti all'identificatore della funzione, relativamente a delle condizioni diverse.

Il programma CONTAPAROLE chiama la funzione CONTEM in linea 19, quale prima voce delle due che devono essere visualizzate dall'istruzione WRITELN.

ESERCIZIO 4.5:
Scrivete e correggete (debug) un programma contenente una funzione che esamini una linea di testo in input e riporti il numero di caratteri non alfabetici contenuti in quella linea. Ricordate che ciascun carattere alfabetico è maggiore o uguale a 'A' e minore o uguale a 'Z', oppure è maggiore o uguale a 'a' e minore o uguale a 'z'. Verificate il vostro programma usando 5 linee qualsiasi prese da questo libro che contengano diversi caratteri di punteggiatura quali '.', ';', ':', '<', etc. assicurandovi poi che il risultato visualizzato dal programma corrisponda al risultato da voi ottenuto manualmente.

7. Parametri variabile

Alle volte può essere desiderabile avere una procedura o funzione che restituisca due o più valori dopo aver completato il proprio lavoro. Questo può essere raggiunto con dei "*parametri variabile*", per distinguerli dai parametri usati finora. I parametri da voi usati fin qui, sono conosciuti come "*parametri chiamati per valore*" ("call-by-value parameters") o semplicemente "*parametri valore*". Questo implica che un parametro valore può *ricevere* un valore solo quando viene chiamata la procedura o funzione.

Il programma tipo PARAMDEMO è una semplice illustrazione della differenza fra parametri valore e parametri variabile. La sintassi per entrambi è illustrata nella parte (b) della figura 4-4, descritta come <elenco parametri>. La procedura VISUAL è usata ad ogni passo del programma per tracciare i risultati di quel passo, man mano che influiscono sulle variabili globali X e Y. Il programma è formato da 5 parti principali, separate per maggior chiarezza da linee bianche, oltre che alla inizializzazione nelle linee 25 e 26. Su una pagina a parte sono riportate le linee visualizzate dal programma PARAMDEMO.

```
1: PROGRAMMA PARAMDEMO;
2: VAR X, Y:INTEGER;
3:
4:    PROCEDURE PV(X: INTEGER);
5:    BEGIN
6:       WRITELN('ENTRANDO IN PV, X=', X);
7:       X:=X+S;
```

```
 8:      WRITELN('USCENDO DA PV, X=', X);
 9:    END (*PV*);
10:
11:    PROCEDURE PN(VAR N: INTEGER);
12:    BEGIN
13:      WRITELN('ENTRANDO IN PN, N=', N);
14:      N:=N+1;
15:      WRITELN('USCENDO DA PN, N=', N);
16:    END (*PN*);
17:
18:    PROCEDURE VISUAL;
19:    BEGIN
20:      WRITELN('X=',X, ', Y=',Y);
21:      WRITELN;
22:    END (*VISUAL*);
23:
24: BEGIN (*PROGRAMMA PRINCIPALE*)
25:    X:=1;
26:    Y:=2;
27:    VISUAL;
28:
29:    PV(Y);
30:    VISUAL;
31:
32:    PV(X);
33:    VISUAL;
34:
35:    PV(3*Y+X);
36:    VISUAL;
37:
38:    PN(X);
39:    VISUAL;
40:
41:    PN(Y);
42:    VISUAL;
43: END.
```

```
 1: Visualizzazione associata a PARAMDEMO
 2:
 3: X=1, Y=2
 4:
 5: ENTRANDO IN PV,    X=2
 6: USCENDO DA PV,     X=7
```

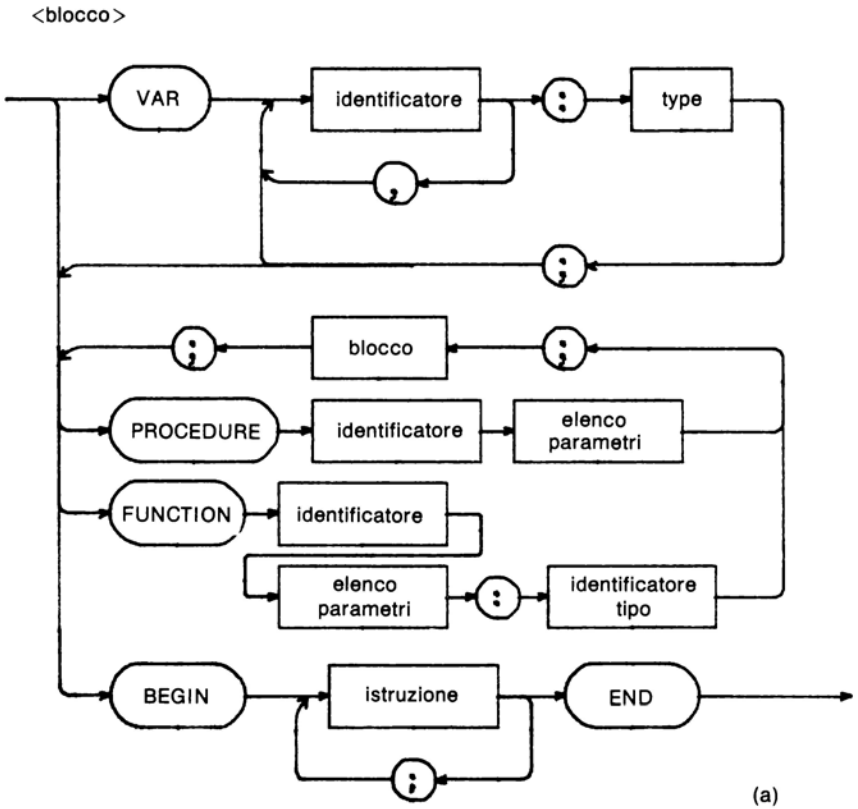
```
 7: X=1, Y=2
 8:
 9: ENTRANDO IN PV,    X=1
10: USCENDO DA PV,     X=6
11: X=1, Y=2
12:
13: ENTRANDO IN PV,    X=7
14: USCENDO DA PV,     X=12
15: X=1, Y=2
16:
17: ENTRANDO IN PN,    N=1
18: USCENDO DA PN,     N=2
19: X=2, Y=2
20:
21: ENTRANDO IN PN,    N=2
22: USCENDO DA PN,     N=3
23: X=2, Y=3
```

La procedura PV ha un parametro *valore* X. La procedura PN ha un parametro *variabile* o "*chiamato per nome*" ("call-by-name"), N. La sintassi richiede che l'identificatore riservato VAR compaia davanti a qualsiasi elenco di identificatori che devono essere considerati parametri variabile. VAR non deve comparire davanti a identificatori che devono essere considerati dei parametri chiamati per valore. Questo richiede che VAR compaia una volta per ogni <tipo> distinto associato ai parametri. Per esempio:

```
PROCEDURE P(A:INTEGER; VAR B,C:INTEGER; D:INTEGER;
        VAR E:BOOLEAN; VAR S:STRING);
```

i parametri B,C,E ed S sono *variabile* mentre A e D sono parametri *valore.*

In linea 29 del programma viene chiamata PV usando Y quale suo parametro effettivo. Y viene valutato ed il suo valore 2 viene assegnato al parametro X all'interno di PV, come è dimostrato dalla linea 5 del display. Poi la procedura somma 5 a X, in linea 7, e visualizza il valore risultato 7 in linea 6. Notate che la linea 7 del display mostra che questo processo non ha portato nessun cambiamento nelle variabili globali X e Y. Solo la variabile locale X nella procedura, cioè il parametro, è cambiata. Analogamente la chiamata di PV con la globale X quale suo parametro effettivo, in linea 32 del programma, non produce nessun cambiamento nel valore della globale X, come viene visualizzato in linea 11.


<blocco>
VAR
identificatore
:
type
,
;
;
blocco
;
PROCEDURE
identificatore
elenco parametri
FUNCTION
identificatore
elenco parametri
:
identificatore tipo
BEGIN
istruzione
END
;
(a)

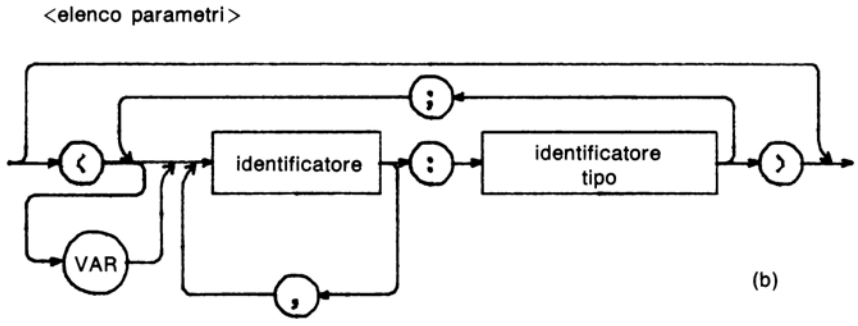

<elenco parametri>
;
(
identificatore
:
identificatore tipo
)
VAR
,
(b)

Figura 4-4

In linea 35 del programma, il parametro effetivo è una espressione aritmetica piuttosto che un semplice riferimento a una variabile. È possibile usare un'espressione come parametro effettivo se l'associato parametro formale (contenuto nella linea di dichiarazione della procedura o funzione) è chiamato per valore e dello stesso <tipo> dell'espressione. Ecco che la linea visualizzata 13 mostra che il parametro X è stato fissato a 7 quando veniva chiamata la procedura PV. Potete facilmente verificare che questo è il valore di 3*Y+X, in questo punto. Ancora una volta la chiamata di PV non ha influito sui valori delle variabili globali X e Y.

In linea 38 del programma viene chiamata PN con un parametro effettivo X. Le linee visualizzate 17 e 18 mostrano che il parametro N è cambiato, come era previsto, dalla addizione in linea 14 del programma. Questa volta però la linea visualizzata 19 mostra che la variabile globale X ha assunto un nuovo valore, cioè il valore avuto dal parametro N al termine di PN. Questo perchè un parametro *variabile* ha preso l'aspetto, all'interno della procedura, del parametro effettivo usato quando la procedura viene chiamata. In questo caso il parametro formale è *sostituito* dal parametro effettivo. Cosicchè la chiamata di PN(X) in linea 38 del programma dovrebbe essere letta come causante la ri-designazione con X di ogni riferimento a N nella procedura PN. Inoltre la chiamata di PN(Y) in linea 41 provoca la lettura di Y in ogni riferimento a N, nella procedura. In tal modo, entrambe le chiamate a PN hanno l'effetto di provocare dei cambiamenti permanenti nelle variabili globali usate come parametri effettivi.

Come regola generale, quando volete comunicare informazioni DENTRO una procedura o funzione dovreste usare parametri *valore*. Quando avete invece bisogno di estrarre informazioni DA una procedura o funzione e non è conveniente usare il meccanismo della restituzione del valore di una funzione, allora dovete senz'altro usare parametri *variabile*. In questo ultimo caso, il compilatore genera delle istruzioni che traducono ogni referenza al parametro variabile formale in referenze alla locazione di memoria del parametro effettivo. Per variabili semplici è generalmente meglio usare il meccanismo del parametro valore. Per strutture di dati complesse, come ne incontreremo nei capitoli 8, 9, e 10, il meccanismo del parametro valore necessita di un elevato tempo di elaborazione che può risultare inutile.

In questi casi sarà preferibile l'uso di parametri variabile. Notate che il meccanismo di sostituzione associato con i parametri variabile significa che essi possono essere usati sia per comunicare informazioni DENTRO una procedura o funzione, sia per estrarre —FUORI— da esse.

ESERCIZIO 4.6:

Rivedete il programma da voi scritto per contare i caratteri alfabetici, nell'esercizio 4.5, in modo che usi una procedura con parametro variabile invece di una funzione. Correggete il programma e verificate che lavori correttamente su almeno 5 linee di testo.

ESERCIZIO 4.7:
Uno dei compiti abituali dei programmatori è quello di progettare una procedura che analizzi (scan) una linea di testo, partendo da una posizione data, e che restituisca quale suo valore la < stringa > associata con il "pezzo" di testo successivo. Nell'ambito di questo esercizio, un "pezzo" potrà essere sia un'unica parola italiana che un qualsiasi altro carattere che non sia "spazio".

Scrivete e correggete un programma che suggerisca ed accetti una qualsiasi stringa di caratteri inserita dalla tastiera, e che visualizzi poi ciascun pezzo della linea immessa su una sua linea a parte. Per facilitare la correzione del programma, è utile visualizzare anche il valore della variabile da voi usata per indicare l'attuale posizione di analisi nella variabile STRINGA contenente la linea immessa al momento in cui ogni pezzo viene visualizzato. Potreste inoltre visualizzare il numero di caratteri che il vostro programma "crede" associati al pezzo visualizzato. Assicuratevi che questa somma non includa nessuno spazio bianco.

Affinchè il vostro programma possa manipolare dei caratteri non alfabetici che non siano però spazi, dovrete usare per l'analisi del testo, delle istruzioni IF piuttosto che la funzione interna POS. In ogni caso COPY e DELETE vi potranno aiutare nel manipolare le sottostringhe che voi estraete per ogni pezzo successivo. Per verificare la correttezza del vostro programma, usate quali linee immesse la 11 e la 13 del programma PARAMDEMO. Controllate i risultati restituiti dal programma cercando visivamente i pezzi contenuti in quelle linee.

8. Procedure recursive

Una procedura è detta *"recursiva"* quando chiama se stessa. Questo meccanismo, concettualmente molto semplice, è uno degli strumenti più potenti nel campo dell'informatica. È molto simile al principio usato in matematica nella descrizione di rapporti che altrimenti risulterebbero troppo complessi. Discutiamo delle procedure recursive a questo punto del libro per due ragioni. Primo, abbiamo constatato che è necessario capire le procedure recursive al fine di raggiungere una comprensione del funzionamento delle procedure in generale.

Secondo, la recursione è così fondamentale nei metodi di soluzione di problemi, che dovreste imparare ad usarne il concetto anche se poi avrete poche occasioni di metterlo in pratica su problemi complessi una volta finito questo libro. La recursione

è spesso considerata un argomento troppo complicato per essere incluso in un corso di programmazione di tipo introduttivo. Noi abbiamo constatato che la recursione non si presenta troppo complessa per la maggior parte degli studenti. Comunque un piccolo sforzo in più per comprendere il materiale sulla recursione oggetto di questo capitolo sarà ben speso.

Come primo passo, fate riferimento al programma RCONT che svolge lo stesso compito di contare parole, eseguito dal programma CONTAPAROLE che abbiamo già descritto. In questo caso, invece di usare l'istruzione FOR per controllare i cicli, chiamiamo CONTEM in linea 9 del programma se il valore della stringa parametro effettivo S contiene almeno uno spazio bianco. Il valore passato alla nuova richiesta di CONTEM quale suo parametro è una COPY di tutti i caratteri di S seguenti il primo carattere bianco. In questo modo la successiva richiesta di CONTEM riceve, quale suo valore di S, una stringa con uno spazio bianco in meno. Infine, ci sarà una richiesta di CONTEM in cui S non ha più spazi bianchi. La clausola ELSE in quella richiesta di CONTEM farà sì che a CONT venga assegnato il valore 1 in linea 11, e la funzione restituirà il valore 1, indicante che è rimasta una sola parola.

```
 1: PROGRAMMA RCONT;
 2: VAR S:STRING;
 3:
 4: PROCEDURE PUNTINI(N:INTEGER);
 5: VAR I:INTEGER;
 6: BEGIN
 7:    FOR I:=1 TO N DO WRITE ('. ');
 8: END (*PUNTINI*);
 9:
10: FUNCTION CONTEM (S:STRING;
11:                  LIVELLO: INTEGER):INTEGER;
12: VAR CONT,K,L:INTEGER;
13: BEGIN
14:    K:=POS(' ',S); L:=LENGTH(S);
15:    PUNTINI(LIVELLO); WRITELN(S, ', L=',L);
16:    IF K>0 THEN
17:         CONT:=1
18:           +CONTEM(COPY(S,K+1,LENGTH(S)-K),LIVELLO+1)
19:      ELSE
20:         CONT:=1;
21:    CONTEM:=CONT;
22:    PUNTINI(LIVELLO);
23:    WRITELN('ESCO, CONT=',CONT, ', L=',L);
24: END (*CONTEM*);
25:
```

```
26: BEGIN (*PROGRAMMA PRINCIPALE*)
27:   WRITELN('CONTAPAROLE');
28:   WRITELN('BATTI UNA QUALSIASI FRASE DI UNA RIGA');
29:   READLN(S);
30:   WHILE LENGTH(S)>0 DO
31:     BEGIN
32:       WRITELN(CONTEM(S,O),' PAROLE');
33:       WRITELN;
34:       WRITELN('BATTINE ANCORA UNA');
35:       REALDLN(S);
36:     END;
37: END.
```

```
 1: Visualizzazione associata con RCONT
 2: IMPORTANTE È LA ROSA
 3: IMPORTANTE È LA ROSA, L=21
 4: . È LA ROSA, L=10
 5: . .  LA ROSA, L=7
 6: . . .  ROSA, L=4
 7: . . .  ESCO, CONT=1, L=4
 8: . . ESCO, CONT=2, L=7
 9: .  ESCO, CONT=3, L=10
10: ESCO, CONT=4, L=21
11: 4 PAROLE
```

L'ultima richiesta di CONTEM terminerà ora lasciando un 1 al suo posto in linea 9 della successiva precedente richiesta, (da cui era stata chiamata l'ultima richiesta di CONTEM). Viene poi fatta la somma, e la penultima richiesta della funzione restituisce il valore 2. Questo valore sostituisce CONTEM in linea 9 della successiva precedente richiesta ed il processo continua fino a quando tutte le richieste saranno terminate ad eccezione della prima. La prima richiesta di CONTEM era stata chiamata dalla linea 32 del programma all'interno dell'istruzione WRITELN. Quando questa richiesta termina, il programma continua a elaborare l'istruzione WRITELN così come era stato fatto nel programma CONTAPAROLE.

È importante rendersi conto che ogni *richiesta* o *attivazione* di una funzione o di una procedura corrisponde a una nuova copia di quella funzione o procedura dichiarata nel vostro programma con un nome leggermente diverso. Potreste pensare a ciascun esempio come avente un colore diverso, così da poterli distinguere fra loro. Cosicchè, quando la richiesta "nera" di CONTEM richiede il "marrone", la funzione "marrone" deve alla fine ritornare al punto dal quale era stata chiamata in linea 18. Quel punto è all'interno della richiesta "nera", non in linea 32 del programma princi-

pale. Ciascuna nuova richiesta ha le sue proprie variabili locali e parametri che sono completamente diversi da tutti gli altri parametri e variabili di tutte le altre richieste. Questo è esattamente come avere una nuova "finestra" nel diagramma a finestre per ciascuna richiesta della funzione o procedura, nonostante che ogni richiesta sembra avere lo stesso nome.

Per un'ulteriore illustrazione dell'operato di RCONT fate riferimento al "listato" visualizzato da questo programma. In linea 3 è rappresentata la linea di testo inserita nel programma dalla tastiera. La linea 4 riprende la stessa informazione ed è visualizzata da WRITELN in linea 15 del programma all'interno della procedura CONTEM, la prima volta che viene chiamata. Questa prima richiesta di CONTEM è chiamata dal riferimento a CONTEM nell'istruzione WRITELN in linea 32 all'interno del programma principale. Notate che WRITELN in linea 32 non perviene alla visualizzazione del proprio output fino a quando CONTEM non è stato chiamato per altre 5 volte, creando altre 5 richieste della procedura che devono terminare per permettere al programma di continuare.

In RCONT come nel programma NIDDEMO, abbiamo usato l'espediente di mostrare punti e indentare le linee visualizzate da una procedura, allo scopo di illustrare quante procedure sono correntemente attivate. Le linee comprese fra 5 e 9 del "listato" sono generate dalle chiamate di PUNTINI e WRITELN (S) in linea 15 della funzione CONTEM, ogni linea essendo visualizzata da una nuova richiesta della funzione. In questo modo ogni linea mostra il valore del parametro S che diminuisce di una parola ad ogni richiesta successiva della funzione. Avendo raggiunto la sesta richiesta di CONTEM, per il quale il valore del parametro LIVELLO è 5 (5 puntini) il valore assegnato a K da POS in linea 14 diventa zero poichè non ci sono spazi bianchi nella stringa 'ROSA'. In questo esempio della funzione viene eseguito il lato ELSE dell'istruzione IF, fissando CONT a 1 senza chiamare CONTEM un'altra volta.

Questo rompe il ciclo delle chiamate ripetute di CONTEM dall'interno dello stesso e la linea visualizzata 10 è generata mentre l'ultimo esempio della funzione sta terminando.

Questo è il punto in cui molti studenti sembrano perdere la traccia di ciò che sta succedendo, per questo qui si impone uno studio accurato del programma e dei "listati" visualizzati, fino a quando sarete in grado di capire perchè ciò che viene visualizzato è generato in un dato modo. Il punto principale da ricordare è che ogni *richiesta* della funzione CONTEM deve terminare normalmente come terminerebbe se fosse una procedura distinta del programma PASCAL.

Poichè CONTEM è una funzione, deve restituire un valore che venga poi usato nel completare il calcolo del valore della espressione all'interno della quale è chiamata la funzione. In tal modo la sequenza di azioni, iniziante quando viene generata la linea

10, è che a CONT nelle linee 17 e 18 del programma viene assegnato il valore 2, (1 è una costante nell'espressione, 1 è il valore restituito dall'ultimo esempio di CONTEM come è mostrato dalla linea 10 del display), poi a CONTEM viene assegnato il valore 2 in linea 21, quindi la linea 11 del display viene generata. Successivamente il penultimo esempio di CONTEM termina facendo sì che il valore 2 sia aggiunto all'interno dell'espressione nelle linee 17 e 18, cosicchè 3 è il valore assegnato a CONT nel precedente esempio di CONTEM. La elaborazione continua, ogni attivazione di CONTEM termina e lascia il suo valore per l'uso nella successiva precedente attivazione della funzione.

Per illustrare ulteriormente l'azione svolta da questo programma è stata aggiunta una variabile locale L nella funzione CONTEM, il cui unico scopo è di aiutarvi a capire il tracciato seguito nell'esecuzione del programma. Ogni volta che CONTEM viene lanciata, a L viene assegnato il valore della LENGTH di S all'interno del corrente esempio di CONTEM. Questo valore è poi visualizzato con S da WRITELN in linea 15, ed ancora mentre la funzione termina in linea 23. Nel primo esempio di CONTEM, a L è assegnato il valore 26, come si nota in linea 4 del display. Successivamente, dopo che tutti gli altri esempi hanno concluso il loro lavoro, l'elaborazione ritorna alla prima richiesta di CONTEM, e viene generata la linea 15. Questa linea mostra che in quell'esempio della funzione, L ha ancora il valore 26 poichè non esistono altre istruzioni, nella funzione, dove a L vengono assegnati nuovi valori. Notate che questa è la dimostrazione che *ogni attivazione della funzione ha la sua propria e distinta serie di variabili locali*, che non vengono divise con nessun'altra attivazione, anche se le istruzioni del programma all'interno di ogni richiesta sono le stesse.

ESERCIZIO 4.8:

Il meccanismo della recursione appena descritto si applica sia alle procedure che alle funzioni. Rivedete il programma RCONT trasformando la funzione CONTEM in procedura. Per il valore di CONT dovrete aggiungere un valore calcolato nella successiva attivazione di CONTEM.

Per restituire questo valore alla richiesta della procedura all'interno della quale state lavorando dovrete usare un parametro variabile poichè il meccanismo del ritorno del valore della funzione non è applicabile. Notate che un modo per restituire questo valore da un'attivazione della procedura all'attivazione che lo ha chiamato, è quello dell'uso di una variabile globale. L'uso di una variabile globale a questo scopo funziona per caso in questo semplice esempio di recursione, ma non sempre funzionerà quando viene usata la recursione. Per cui il meccanismo del parametro variabile è quello che dovreste usare in questo esercizio.

9. Uso improprio della recursione in PASCAL

Mentre il programma RCONT presentava un esempio di cui ci siamo serviti per spiegare il funzionamento della recursione, il programma CONTAPAROLE rappresenta in realtà il metodo migliore per risolvere il problema contenuto in entrambi i programmi. La ragione principale dell'uso di questo approccio risiede nel nostro voler evitare di complicare il problema fino a quando non avrete capito questo meccanismo. I programmi presentati in questa sezione vi permetteranno di sviluppare una maggior comprensione di come opera la recursione, ma anch'essi potrebbero essere programmati abbastanza velocemente per operare più efficacemente in un modo non - recursivo.

In generale la recursione sarà usata in modo appropriato nei casi in cui l'algoritmo richiede un'attivazione di una procedura che chiami se stesso o una procedura affine, almeno per due volte. Le variabili locali, nella procedura, servono allora per conservare delle informazioni dal momento in cui la prima nuova richiesta è chiamata fino a quando quell'attivazione ritorna e vengono fatti i preparativi per la prossima chiamata. Questo principio sarà illustrato in diversi programmi della prossima sezione. Un altro uso appropriato si può presentare quando sia necessario conservare ciascun elemento di una sequenza di valori per diversi usi successivi che si presenteranno nell'ordine inverso a quello dell'elaborazione iniziale.

Il programma FATTORIALE illustra l'esempio di recursione probabilmente più usato nei testi introduttivi alla programmazione, in modo improprio. La funzione fattoriale ha in matematica la seguente semplice definizione:

fattoriale(N) = N * fattoriale(N–1)

Un ulteriore dato è che fattoriale (1) è definito avere un valore di 1. Per esempio:

```
fattoriale(2) = 2 *  1
fattoriale(3) = 3 *  2 =   6
fattoriale(4) = 4 *  6 =  24
fattoriale(5) = 5 * 24 = 120
```

e così di seguito. Il nostro programma computa il valore della funzione fattoriale per un *"argomento"*, cioè un valore da usarsi quale parametro effettivo, che voi inserite dalla tastiera. Il programma permette che l'argomento sia una qualsiasi cifra compresa fra 0 e 7.

Valori maggiori risultano dalla computazione di valori troppo grandi per essere tenuti in una sola cella della memoria del microelaboratore da voi usato. Questi valori potranno essere calcolati con esattezza su elaboratori più grandi, o con approssimazione su un microelaboratore, come vedremo nel prossimo capitolo.

```
 1: PROGRAMMA FATTORIALE;
 2: VAR X,RISULTATO:INTEGER;
 3:
 4: PROCEDURE PUNTINI (N:INTEGER);
 5: VAR I:INTEGER;
 6: BEGIN
 7:    FOR I:=1 TO N DO WRITE('.');
 8: END (*PUNTINI*);
 9:
10: FUNCTION RFATT(N,LIVELLO:INTEGER):INTEGER;
11: VAR RF:INTEGER;
12: BEGIN
13:    PUNTINI(LIVELLO);
14:    WRITELN('ENTRO,N=', N);
15:    IF N>1 THEN RF:=RFATT(N-1,LIVELLO+1)*N
16:           ELSE RF:=1;
17:    RFATT:=RF;
18:    PUNTINI(LIVELLO);
19:    WRITELN('ESCO, N=',N, ', RF=',RF);
20: END (*RFATT*)
21:
22: FUNCTION FATT (N:INTEGER):INTEGER;
23: VAR I,X:INTEGER;
24: BEGIN
25:    X:=1;
26:    FOR I:=N DOWNTO 2 DO X:X*I;
27:    FATT:=X
28: END (*FATT*);
29:
30: BEGIN (*PROGRAMMA PRINCIPALE*)
31:    WRITELN('FATTORIALE');
32:    WRITELN('BATTI QUALUNQUE CIFRA TRA 0 E 7, POI <RET>);
33:    READLN(X);
34:    WHILE (X=>0) AND (X<=7) DO
35:       BEGIN
36:     RISULTATO:=RFATT(X,0);
37:          WRITELN('X=',X, ',RFATT=', RISULTATO,
38:                              ',FATT=',FATT(X) );
39:          WRITELN;
40:          WRITELN('CONTINUA');
41:          READLN(X);
42:       END;
43: END.
```

```
 1: Visualizzazione associata a FATTORIALE
 2:
 3: BATTI QUALUNQUE CIFRA TRA 0 E 7
 4: 1
 5: ENTRO, N=1
 6: ESCO, N=1 RF=1
 7: X=1, RFATT=1, FATT=1
 8:
 9: CONTINUA
10: 2
11: ENTRO, N=2
12: . ENTRO, N=1
13: . ESCO, N=1, RF=1
14: ESCO, N=2, RF=2
15: X=2, RFATT=2, FATT=2
16:
17: CONTINUA
18: 3
19: ENTRO, N=3
20: . ENTRO, N=2
21: . . ENTRO, N=1
22: . . ESCO, N=1; RF=1
23: . ESCO, N=2; RF=2
24: ESCO, N=3, RF=6
25: X=3, RFATT=6, FATT=6
26:
27: CONTINUA
28: 4
29: ENTRO, N=4
30: . ENTRO, N=3
31: . . ENTRO, N=2
32: . . . ENTRO, N=1
33: . . . ESCO, N=1, RF=1
34: . . ESCO, N=2, RF=2
35: . ESCO, N=3; RF=6
36: ESCO, N=4, RF=24
37: X=4, RFATT=24, FATT=24.
```

La pagina riportante la visualizzazione del programma FATTORIALE contiene esempi per argomenti di 1,2,3 e 4. Una delle ragioni principali per dare le quattro illustrazioni è quella di mostrare che il programma determina il numero di esempi della funzione RFATT che saranno chiamati, in base ai valori dei dati con cui sta lavoran-

do. Quando il parametro N non è superiore a 1, allora la funzione restituisce 1 piuttosto che chiamare RFATT con un valore di N–1 per parametro effettivo.

Usiamo la variabile locale RF per contenere il valore restituito dalla funzione, così che questo valore possa essere usato all'interno dell'istruzione WRITELN, in linea 19, senza causare una nuova chiamata della funzione. Oppure sarebbe stato possibile assegnare direttamente dei valori a RFATT in linea 15 e 16.

Ora concentrate la vostra attenzione sulla funzione FATT. Come viene chiaramente mostrato dalla visualizzazione, FATT calcola lo stesso valore di RFATT quando il valore del parametro è lo stesso. FATT si basa sull'osservazione che la tabella precedentemente riportata può essere riscritta nel seguente modo:

```
fattoriale(2) = 2 *1
fattoriale(3) = 3 *2*1
fattoriale(4) = 4 *3*2*1
fattoriale(5) = 5 *4*3*2*1
```

L'uso dell'istruzione FOR nel calcolo di questo risultato è abbastanza semplice tanto da non richiedere il ricorso all'espediente, peraltro meno efficace, della funzione o procedura recursiva. Se tutto ciò che dobbiamo fare consiste nel computare il valore della funzione fattoriale occasionalmente per un valore, allora la differenza in efficienza non ha molta rilevanza. Comunque è spesso necessario scrivere una procedura o funzione che sarà usata un elevato numero di volte. In quel caso è importante evitare l'ulteriore tempo di elaborazione necessario per chiamare una procedura o funzione, se questo non implica però una maggiore difficoltà nella correzione del programma.

Per chiarire maggiormente questo punto analizzeremo ora delle funzioni recursive e non-recursive per il calcolo di membri di una sequenza di numeri, importante in matematica, e conosciuta come sequenza di "Fibonacci". Ogni membro successivo della sequenza è uguale alla somma dei precedenti due membri. Possiamo così ottenere:

```
0 1 1 2 3 5 8 13 21 34 55 89
```

etc, dove i primi due membri, 0 e 1 sono definiti per iniziare la sequenza. Possiamo quindi dire che:

```
fib(N) = fib(N–1) + fib(N–2)
```

dove l'argomento è il numero del membro della sequenza partendo da 0 (zero). Per esempio:

```
fib(3) = fib(2) + fib(1) = 1 + 1 = 2
fib(6) = fib(5) + fib(4) = 5 + 3 = 8
```

e così via. Il programma TRACFIB contiene sia funzioni recursive (RFIB) che funzioni non-recursive (FIB) per calcolare il membro N di questa sequenza. Le linee visualizzate ottenute per N valori di 2,3 e 4 sono mostrate su una pagina a parte. La cosa più importante da notare è che RFIB viene chiamata più di una volta per calcolare il valore dei membri aventi il numero più basso della sequenza. Per esempio nelle linee 32, 37 e 41 della visualizzazione, la traccia ci dice che RFIB è stata chiamata per calcolare RFIB(1), tutto nel corso del calcolo di un solo membro della sequenza, RFIB(4). Il numero di volte che RFIB verrà chiamata in questa soluzione recursiva al problema, diventerà estremamente elevato in breve tempo. Per esempio RFIB(11) richiederà la chiamata di RFIB per 1024 volte!

Per contro la funzione FIB usa un semplice ciclo FOR che conserva i valori di due elementi della sequenza che userà poi nel calcolo dell'elemento successivo. Una volta che il nuovo elemento della sequenza è stato assegnato ad X, nuovi valori possono essere assegnati a XDIETRO1 e XDIETRO2, da usarsi nel ciclo successivo. L'ammontare di elaborazione necessario aumenta in proporzione al valore N passato a FIB, mentre aumenta in proporzione a 2 elevato alla potenza (N−1) nel caso recursivo! Ovviamente la soluzione migliore a questo problema è quella non recursiva.

```
 1: PROGRAMMA TRACFIB;
 2: VAR X,RISULTATO:INTEGER;
 3:
 4: PROCEDURE PUNTINI(N:INTEGER);
 5: VAR I:INTEGER;
 6: BEGIN
 7:    FOR I:=1 TO N DO WRITE('. ');
 8: END (*PUNTINI*)
 9:
10: FUNCTION RFIB(N,LIVELLO:INTEGER):INTEGER;
11: VAR RF:INTEGER;
12: BEGIN
13:    PUNTINI(LIVELLO); WRITELN('ENTRO, N=',N);
14:    IF N>1 THEN
15:       RF:=RFIB(N-1,LIVELLO+1) + RFIB(N-2,LIVELLO+2)
16:       ELSE
```

```
17:         IF N=1 THEN RF:=1
18:           ELSE RF:=0;
19:     RFIB:=RF;
20:     PUNTINI(LIVELLO);
21:     WRITELN('ESCO, N=',N, ', RF='RF);
22: END (*RFATT*)
23:
24: FUNCTION FIB(N:INTEGER):INTEGER;
25: VAR XDIETRO1;XDIETRO2,SALVA,X,I:INTEGER;
26: BEGIN
27:     IF N<=1 THEN FIB=N ELSE
28:     BEGIN
29:       XDIETRO1:=1; XDIETRO2:=0;
30:       FOR I:=2 TO N DO
31:         BEGIN
32:           X:=XDIETRO1 + XDIETRO2;
33:           (*prepara ora per il prossimo giro*)
34:           SALVA:=XDIETRO1; XDIETRO1:=X; XDIETRO2:=SALVA;
35:         END;
36:       FIB:=(X);
37:     END;
38: END (*FIB*);
39:
40: BEGIN (*PROGRAMMA PRINCIPALE*)
41:     WRITELN('FIBONACCI');
42:     WRITELN('BATTI QUALSIASI CIFRA TRA 0 E 7, POI <RET>');
43:     READLN(X);
44:     WHILE (X>=0) AND (X<=7) DO
45:       BEGIN
46:         RISULTATO:=RFIB(X,0);
47:         WRITELN('X=',X, ', RFIB=', RISULTATO,
48:                          ',FIB=', FIB(X));
49:         WRITELN;
50:         WRITELN ('CONTINUA');
51:         READLN(X);
52:       END;
53: END.
```

```
1: Visualizzazione associata a TRACFIB
2:
3: BATTI QUALSIASI CIFRA TRA 0 E 7, POI <RET>
4: 2
```

```
 5: ENTRO, N=2
 6: . ENTRO, N=1
 7: . ESCO, N=1, RF=1
 8: . ENTRO, N=0
 9: . ESCO, N=0, RF=0
10: ESCO, N=2, RF=1
11: X=2, RFIB=1, FIB=1
12:
13: CONTINUA
14: 3
15: ENTRO, N=3
16: . ENTRO, N=2
17: . . ENTRO, N=1
18: . . ESCO, N=1, RF=1
19: . . ENTRO, N=0
20: . . ESCO, N=0, RF=0
21: . ESCO, N=2, RF=1
22: . ENTRO, N=1
23: . ESCO, N=1, RF=1
24: ESCO, N=3, RF=2
25: X=3, RFIB=2, FIB=2
26:
27: CONTINUA
28: 4
29: ENTRO, N=4
30: . ENTRO, N=3
31: . . ENTRO, N=2
32: . . . ENTRO, N=1
33: . . . ESCO, N=1, RF=1
34: . . . ENTRO, N=0
35: . . . ESCO, N=0, RF=0
36: . . ESCO, N=2, RF=1
37: . . ENTRO, N=1
38: . . ESCO, N=1, RF=1
39: . ESCO, N=3, RF=2
40: . ENTRO, N=2
41: . . ENTRO, N=1
42: . . ESCO, N=1, RF=1
43: . . ENTRO, N=0
44: . . ESCO, N=0, RF=0
45: . ESCO, N=2, RF=1
46: ESCO, N=4, RF=3
47: X=4, RFIB=3, FIB=3.
```

Nonostante che RFIB richieda la conservazione del valore di N localmente nella funzione così da poterlo usare nella elaborazione dei valori N–1 e N–2 quando si chiamano dei nuovi esempi di RFIB, il programma non deve mai conservare più di due valori dalla sequenza, al fine di arrivare al valore successivo. È generalmente vero che quando la recursione è la soluzione migliore, il programma deve conservare un numero indefinitivamente elevato di valori per usi successivi. I programmi illustrati nella prossima sezione hanno tutti questa caratteristica e risulterebbero molto più difficili da scrivere senza la recursione.

ESERCIZIO 4.9:

Scrivete e correggete un programma contenente una procedura recursiva INV scritta per invertire l'ordine dei caratteri di una stringa qualsiasi. In altre parole la logica generale della procedura dovrebbe essere la seguente: uso di un unico parametro di < tipo > STRING, che potrà essere chiamato "SORGENTE" in riferimento al valore originario della stringa da invertire. All'interno di INV, se la lunghezza di SORGENTE è superiore a un carattere, conservate il primo carattere in una variabile STRING locale, poi chiamate di nuovo INV, con una copia della stringa SORGENTE contenente tutti i suoi caratteri originali eccetto il primo. Seguendo la chiamata a INV, concatenate o inserite il carattere conservato alla fine di una variabile STRING globale DEST. Se la lunghezza di SORGENTE all'entrata di INV, è di un solo carattere, allora inizializzate DEST assegnandogli il valore di SORGENTE.

Aggiungete delle istruzioni WRITELN al fine di tracciare il valore di SORGENTE ogni volta che INV è chiamata, e al fine di mostrare i valori di SORGENTE, DEST e del carattere conservato appena prima del termine di INV. Verificate il programma con stringhe di varia lunghezza. Inserite dalla tastiera. Analizzate il vostro video ed assicuratevi di capire la derivazione dal programma di ogni linea.
Cercate di implementare lo stesso algoritmo in un modo non-recursivo dopo che avete verificato il funzionamento della versione recursiva.

10. Applicazione della recursione

In questa sezione presenteremo tre programmi rappresentativi della complessità di problemi che possono essere risolti con programmi abbastanza semplici. Tutti e tre sono orientati graficamente per questo motivo possono essere usati per darvi una

visione sequenziale della recursione che è difficilmente esprimibile in un testo. Analizzeremo in dettaglio solo uno di questi programmi. Esso si serve di un algoritmo, parente stretto di una famiglia di algoritmi molto importante, usata per una vasta gamma di applicazione in informatica. Questi algoritmi, che usano un dispositivo logico chiamato *"albero"*, si stanno rapidamente diffondendo anche in settori commerciali.

Fate riferimento al programma SUALBERO, ed alla figura 4-5 illustrante esempi di figure visualizzate da questo programma. Come primo passo nella spiegazione del programma è necessario parlare della *"istruzione CASE"* che compare nelle linee comprese fra la 10 e la 19 del programma. L'istruzione CASE è in effetti simile ad un'istruzione IF che dia più di due scelte possibili di azione. Come per l'istruzione IF, solo una delle scelte possibili è eseguita, dopodichè la esecuzione continua al termine dell'istruzione CASE. La figura 4-6 illustra sintassi e diagramma di flusso dell'istruzione CASE.

Entro la "catena" dell'istruzione CASE esistono diverse istruzioni indipendenti, ognuna delle quali è segnata da una o più costanti seguite da un carattere due punti (":"). Solo una delle istruzioni all'interno di questa catena viene eseguita. Quell'istruzione è selezionata dall'intero gruppo dal valore dell'< espressione > nella linea di intestazione dell'istruzione CASE, che deve corrispondere a uno dei segni delle costanti. Se il valore dell'< espressione > non corrisponde a nessuno dei segni, allora l'azione da prendere è lasciata indefinita dalla definizione standard di PASCAL. Generalmente è meglio usare delle istruzioni IF appena prima della linea di intestazione dell'istruzione CASE, il cui scopo è quello di assicurare che il valore del selettore dell'< espressione > è presente all'interno della catena di segni delle costanti. Ci dovrebbe essere un'istruzione all'interno della catena di questi segni che corrisponda ad ogni possibile valore della < espressione > non rifiutato dall'istruzione IF. Il compilatore indicherà un errore di sintassi se *l'ultima* istruzione all'interno della catena dell'istruzione CASE viene terminata con un punto e virgola (";").

```
 1: PROGRAMMA SUALBERO;
 2: VAR SCALA ORDINE:INTEGER;
 3:
 4: PROCEDURE ALBERO(X,Y,LUNG,DIR:INTEGER);
 5:   PROCEDURE CAMBIAXY;
 6:   VAR DX,DY:INTEGER;
 7:   BEGIN
 8:     IF DIR<0 THEN DIR:=DIR+8;
 9:     IF DIR>=8 THEN DIR:DIR-8;
10:     CASE DIR OF
11:       0: BEGIN DX:=1; DY:=0; END;
12:       1: BEGIN DX:=1; DY:=1; END;
```

```
13:          2: BEGIN DX:=0; DY:=1; END;
14:          3: BEGIN DX:=1; DY:=1; END;
15:          4: BEGIN DX:=-1; DY:=0; END;
16:          5: BEGIN DX:=1; DY:=1; END;
17:          6: BEGIN DX:=0; DY:=-1; END;
18:          7: BEGIN DX:=1; DY:=-1; END;
19:        END (*CASE*)
20:        X:=X+SCALA*LUNG*DX;
21:        Y:=Y*SCALA*LUNG*DY;
22:     END (*CAMBIAXY*);
23: BEGIN (*ALBERO*)
24:     PENCOLOR(NONE);
25:     MOVETO(X,Y);
26:     TURNTO(DIR*45);
27:     PENCOLOR(WWHITE);
28:     CAMBIAXY;
29:     MOVETO(X,Y);
30:     IF LUNG>1 THEN
31:     BEGIN
32:        ALBERO(X,Y,LUNG-1,DIR+1);
33:        ALBERO(X,Y,LUNG-1,DIR-1);
34:     END;
35: END (*ALBERO*);
36:
37: BEGIN (*PROGRAMMA PRINCIPALE*);
38:     WRITE ('SCALA:');
39:     READLN(SCALA);
40:     WRITE('ORDINE:');
41:     READLN(ORDINE);
42:     ALBERO(0,-SCALA*ORDINE-100,ORDINE,2);
43: END.
```

Nel programma SUALBERO, l'istruzione CASE equivale a ciò che segue:

```
IF DIR=0 THEN
  BEGIN DX:=1; DY:=0; END
  ELSE IF DIR=1 THEN
    BEGIN DX:=1; DY:=1; END
    ELSE IF DIR=2 THEN
      BEGIN DX:=0; DY:=1; END
      ELSE IF DIR=3 THEN
        BEGIN...
          etc...
```

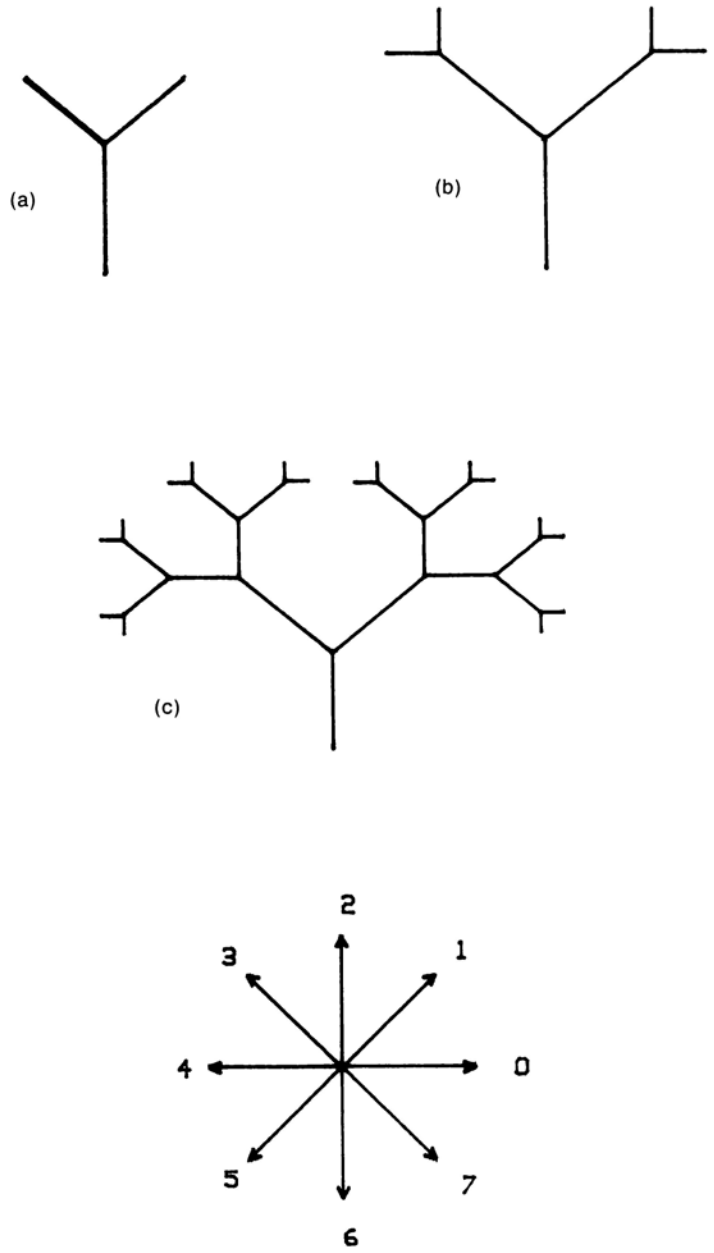
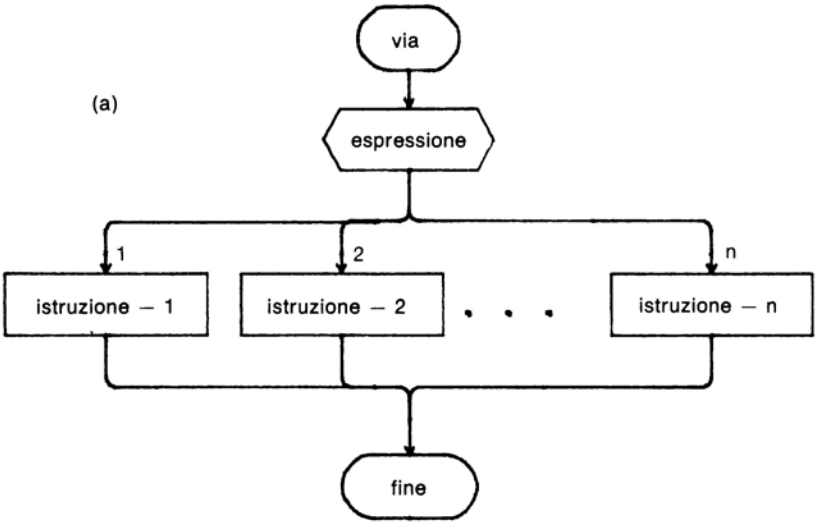


Figura 4-5


(a)
via
espressione
1
2
n
istruzione – 1
istruzione – 2
istruzione – n
fine

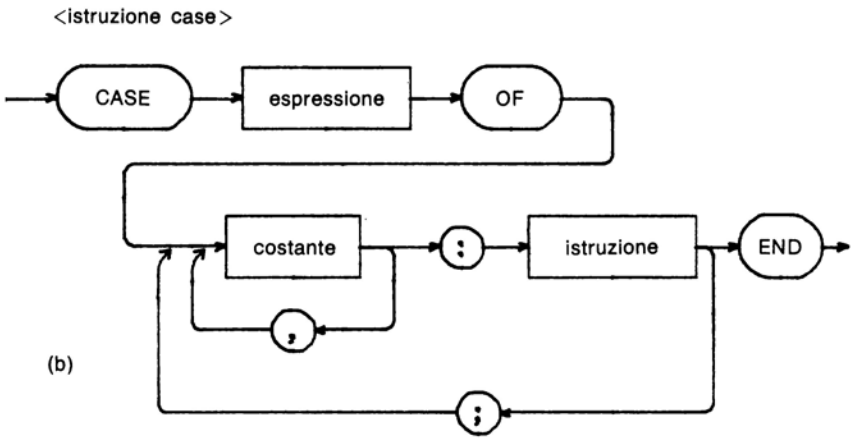

<istruzione case>
CASE
espressione
OF
costante
;
istruzione
END
,
(b)
;

Figura 4-6

L'istruzione CASE risulta più semplice da scrivere e leggere di una lunga sequenza di istruzioni IF nidificate, come sopra. In media l'istruzione CASE verrà eseguita più velocemente dell'equivalente nido di istruzioni IF.

La parte (d) della figura 4-5 illustra lo scopo della procedura CAMBIAXY nel programma tipo contenente l'istruzione CASE. La figura associa un intero della serie 0, 1, 2, ..,7 con ciascuna delle otto frecce direzionali della tartaruga. CAMBIAXY calcola nuove coordinate di posizione usate poi dall'istruzione MOVETO, in linea 29 del programma. L'effetto delle linee comprese fra la 25 e la 29 del programma, la chiamata di CAMBIAXY inclusa, è quello di far sì che la tartaruga disegni una linea in una di queste otto direzioni, come specificato dal parametro valore DIR della procedura ALBERO.

La lunghezza della linea è definita da LUNG. Le linee delle quattro direzioni diagonali (DIR=1, 3, 5, o 7) sono più lunghe di quelle delle direzioni orizzontali o verticali in quanto sia DX che DY hanno valore assoluto 1.

La principale azione svolta dal programma è compiuta dalle due chiamate che la procedura ALBERO fa a se stessa in linea 32 e 33. Questa azione è illustrata nella parte (a) della figura 4-5 in cui la variabile ORDINE viene inizialmente fissata a 2 da READLN in linea 41. ALBERO chiamato in linea 42, assegna il valore 2 al parametro LUNG. Le linee 24 e 25 provocano lo spostamento della tartaruga, senza che disegni alcuna linea, alla posizione dello schermo definita dai valori di X e Y. Le linee da 27 a 29 formano la riga verticale costituente il "tronco" dell'albero a due rami, oggetto della figura da disegnare. Poichè LUNG ha un valore maggiore di 1, ALBERO viene chiamato 2 volte: una volta nella direzione alla sinistra del tronco (DIR+1), un'altra volta nella direzione alla destra del tronco (DIR–1). In entrambi i casi, il nuovo LUNG è specificato essere più corto di una unità cioè 2–1=1. In questo modo i due "rami" dell'albero risultano essere la metà del tronco.

La parte (b) della figura 4-5 mostra l'albero disegnato da questo programma quando ORDINE è inizializzato a 3, la parte (c) illustra l'albero con ORDINE=5.

ESERCIZIO 4.10:

Implementate (cioè battete sulla tastiera e lanciate) il programma SUALBERO sul vostro elaboratore. Verificate che la visualizzazione da voi ottenuta sia la stessa di quella mostrata nella figura 4-5 per ciascuno dei tre valori di ORDINE 2, 3 e 5, lì illustrati. Ora inserite un'istruzione READLN fra le linee 29 e 30 del programma. Quando lanciate il programma così modificato, dovreste ottenere quale risultato la visualizzazione del tronco dell'albero, dopodichè il programma

aspetterà per ulteriori istruzioni. Premete una volta <RET>, concludendo così l'istruzione READLN, ed osservate che verrà disegnata un'altra linea. Ogni volta che premerete il tasto <RET> apparirà un'altra linea.

Ora fate operare il programma con un valore di ORDINE (immesso da tastiera) 2. Osservate l'azione linea per linea, assicurandovi di capire quello che viene fatto dal programma ad ogni passo. Ora fate operare il programma con un valore di ORDINE più elevato, con 5 o 6 per esempio. Di nuovo osservate l'azione svolta linea per linea e notate l'ordine di tracciamento delle diverse linee. Notate che ciascuna delle due linee disegnate dalle chiamate di ALBERO (linee 32 e 33 del programma), da ciascuna delle *richieste* di ALBERO, "spuntano" dalla stessa locazione cioè dagli stessi valori di X e Y. Considerato che durante l'intervallo fra il momento di cui viene disegnato il primo ramo principale e quello in cui viene disegnato il secondo, vengono tracciati diversi piccoli rami, risulta chiaro che i valori di X e Y corrispondenti ai rami principali sono stati conservati, senza cambiamenti, dal primo esempio della procedura ALBERO.

Nel caso di alberi più grandi, si può applicare la stessa osservazione per ogni livello di ramificazione superiore corrispondente al livello di attivazione superiore della procedura.

Qualora questa dimostrazione non sia sufficiente per spiegare l'operazione del programma, vi suggeriamo di inserire delle istruzioni WRITELN, che visualizzino i valori di tutti i parametri di ALBERO, appena prima delle linee 24 e 35 rispettivamente. Ciò non risulterà molto pratico su alcuni dispositivi-video che non possono manipolare separatamente e simultaneamente sia grafici che caratteri alfanumerici.

ESERCIZIO 4.11:
Modificate il programma SUALBERO in modo che visualizzi gli alberi illustrati nella figura 4-7. Il cambiamento principale è costituito dal fatto che da ogni punto di ramificazione spuntano tre rami. Notate che in ciascun caso il ramo di mezzo non si ramifica in rami più piccoli. Nota: questa modifica risulterà molto più semplice da apportare se capite l'algoritmo. È necessario in questo caso aggiungere o cambiare non più di poche linee.

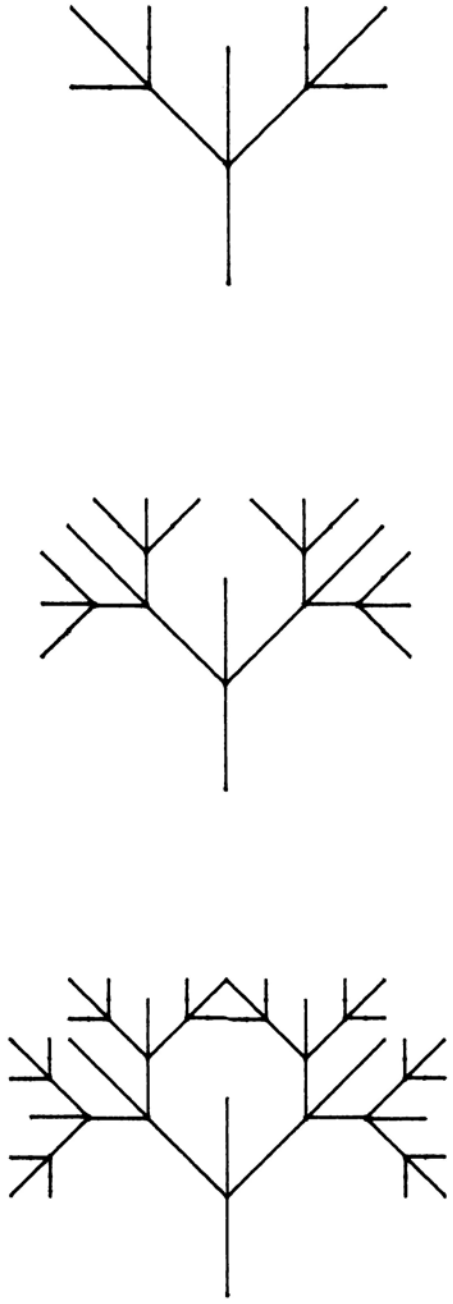


Figura 4-7

ESERCIZIO 4.12:

Modificate il programma SUALBERO al fine di visualizzare l'"albero di mele" rappresentato nella figura 4-8. Anche questa modifica dovrebbe risultare molto semplice.

Ora fate riferimento al programma DRAGONI illustrante la complessità di un'interessante figura che può essere disegnata da un programma molto semplice. Il "dragone" mostrato nella figura 4-9 risulta dalla risposta con la cifra 8 alla richiesta di una "dimensione". L'algoritmo richiede il disegno di una linea solo quando il valore del parametro LENGTH della procedura DRAGON è zero.

ESERCIZIO 4-13:

In pratica l'unico modo efficace per riuscire a comprendere il funzionamento del programma DRAGONI è di eseguire il programma passo passo, linea per linea. Implementate il programma sul vostro elaboratore, aggiungendo un'istruzione READLN fra le linee 5 e 6. Se la vostra unità video permette la sovrapposizione sia di grafici che di caratteri alfanumerici, allora aggiungete anche WRITELN ('L', LENGTH) nello stesso punto. Ora fate girare il programma per diverse volte con valori diversi di DIMEN partendo da 1. Seguite l'azione passo passo mantenendo una traccia di ciò che succede confrontando con il programma stampato, o con un listato reperito dal vostro centro elaborazione.

Quando pensate di aver capito come è stato disegnato il drago, provate a modificare il programma in modo che disegni una figura riflessa del drago illustrato nella figura 4-9, cioè fate sì, che il drago sia rivolto verso destra invece che verso sinistra.

```
 1: PROGRAMMA DRAGONI;
 2: VAR DIMEN:INTEGER;
 3:
 4: PROCEDURE DRAGONE(LUNG:INTEGER);
 5: BEGIN
 6:   IF LUNG=0 THEN MOVE(30)
 7:   ELSE
 8:     IF LUNG>0 THEN
 9:       BEGIN
10:       DRAGONE(LUNG-1);
11:       TURN(90);
12:       DRAGONE(-(LUNG-1));
13:       END
14:   ELSE
```

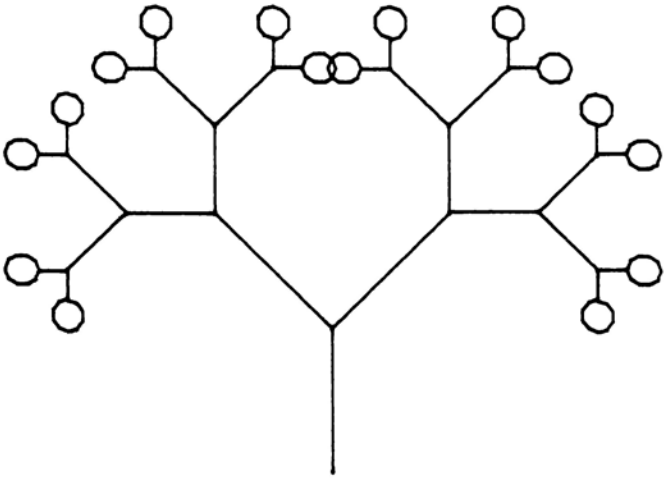


Figura 4-8

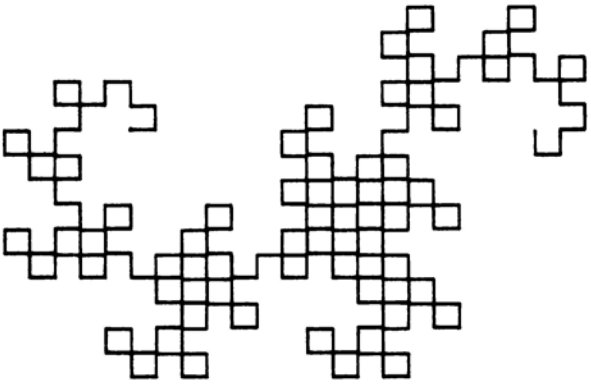


Figura 4-9

```
15:       BEGIN
16:         DRAGONE(-(LUNG+1));
17:         TURN(270);
18:         DRAGONE(LUNG+1);
19:       END;
20: END (*DRAGONE*);
21:
22: BEGIN (*PROGRAMMA PRINCIPALE*)
23:   WRITE('DIMENSIONE DEL DRAGO:');
24:   READLN(DIMEN);
25:   PENCOLOR(NONE);
26:   MOVETO(-200,0);
27:   (*USATE 1/5 DELLA LARGHEZZA DELLO SCHERMO INVECE DI 200*)
28:   PENCOLOR(WHITE);
29:   DRAGONE(DIMEN);
30: END.
```

Eseguite questi cambiamenti sulla carta prima di praticare dei cambiamenti nel programma. Il cambiamento è molto semplice e dovreste riuscirvi la prima volta. Poi provate a tracciare la figura del drago capovolto, ma rivolto nella stessa direzione della figura 4-9.

Se voi trovate la "potenza" rivelata da questo semplice programma, interessante, ciò dovrebbe spingervi a sperimentare con nuovi cambiamenti al programma da voi progettato. Provate a disegnare delle figure più interessanti del drago con un programma altrettanto semplice. (L'autore sarebbe contento di ricevere delle copie di qualsiasi esempio, abbastanza interessante, per future pubblicazioni).

ESERCIZIO 4.14:
Se avrete trovato i due esercizi precedenti troppo semplici per le vostre capacità, provate a sperimentare con gli esempi illustrati nella figura 4-10. Il programma che disegna questa figura segue un algoritmo molto conosciuto, simile ad algoritmi che possono essere usati per creare una vasta gamma di esempi complessi ripetitivi. Prima di fare riferimento al programma HILBERT, che ha disegnato tutti gli esempi della figura 4-10, vedete se riuscite ad escogitare una procedura recursiva che disegni lo stesso esempio.

La parte (a) della figura 4-10 illustra la figura di base a tre lati che viene ripetuta con diversi orientamenti per creare tutte le altre. La parte (a) è stata disegnata con ORDINE inizializzato a 1. La parte (b) è stata costruita usando la figura di base della parte (a) per quattro

volte, con tre diversi orientamenti. È stato necessario disegnare tre linee diritte per congiungere le quattro figure di base. Ognuna di queste linee diritte ha la stessa lunghezza del lato di ognuna delle figure di base.

```
 1: PROGRAMMA HILBERT;
 2: VAR DIMEN,DELTA,N:INTEGER;
 3:   ORDINE:INTEGER;
 4:   PROCEDURE HIL(I:INTEGER);
 5:   VAR A,B:INTEGER;
 6:     PROCEDURE HIL1;
 7:     BEGIN
 8:       TURN(A); HIL(-B); TURN(A);
 9:     END (*HIL1*);
10:     PROCEDURE HIL2;
11:     BEGIN
12:       MOVE(DIMEN);
13:       HIL(B);
14:       TURN(-A); MOVE(DIMEN); TURN(-A);
15:       HIL(B);
16:       MOVE(DIMEN);
17:     END (*HIL2*);
18:   BEGIN (*HIL*)
19:     IF I=0 THEN TURN(180)
20:     ELSE
21:     BEGIN
22:       IF I>0 THEN
23:       BEGIN
24:       A:=90; B=I-1;
25:       END
26:       ELSE
27:         BEGIN
28:           A=-90; B:=I+1;
29:         END;
30:       HIL1; HIL2; HIL1;
31:     END;
32:   END (*HIL*);
33: BEGIN (*PROGRAMMA PRINCIPALE*);
34:   WRITE('DIMENSIONE:');
35:   READLN(DIMEN); (*DIMENSIONI SCHERMO*)
36:   WRITE('ORDINE:'); READLN(ORDINE);
37:     PENCOLOR(NONE);
```

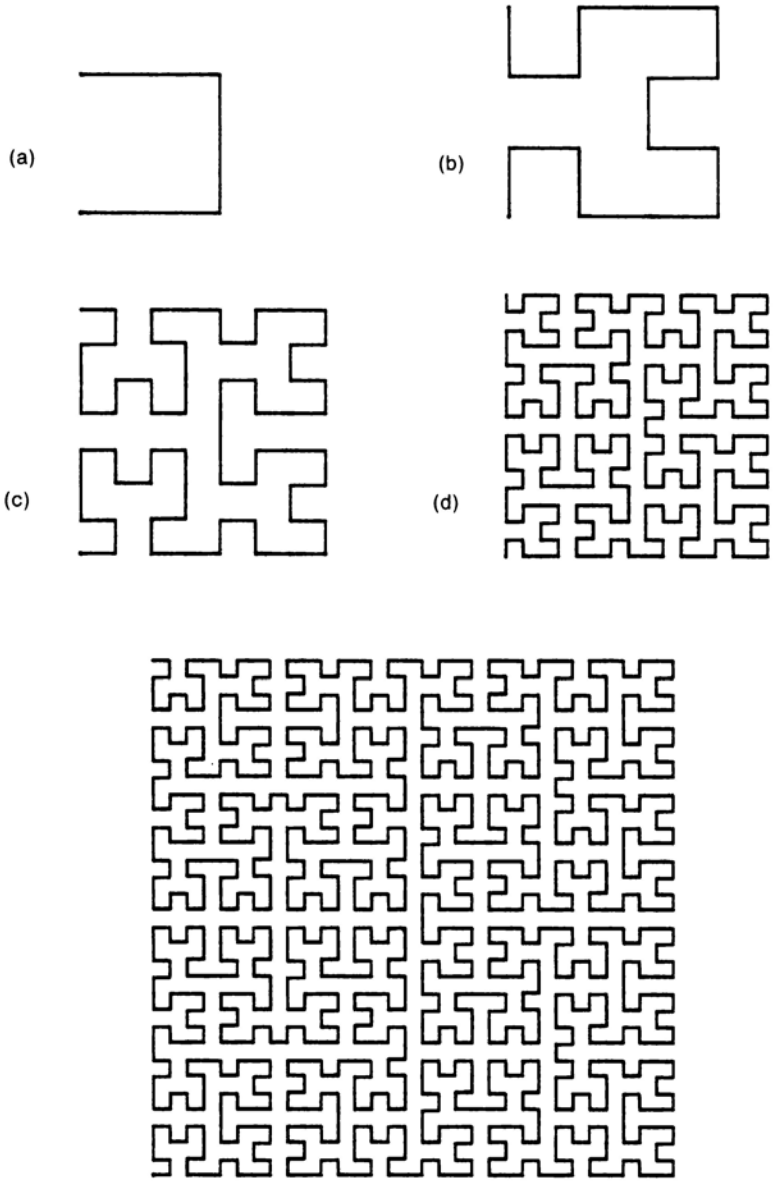


Figura 4-10

```
38:      N:=ORDINE-1;
39:      DELTA:=SIZE;
40:    WHILE N>0 DO
41:    BEGIN (*CALCOLA IL PUNTO DI PARTENZA X,Y*)
42:      DELTA:=DELTA*2;
43:      N:=N-1;
44:    END;
45:    MOVETO(-DELTA,-DELTA);
46:    PENCOLOR(WHITE);
47:    HIL(ORDINE);
48: END.
```

La parte (c) della figura 4-10 è ottenuta (ORDINE=3) congiungendo quattro figure come quella riportata nella parte (b). Di nuovo sono state usate tre linee diritte per congiungere le suddette quattro figure. Analogamente la parte (d) è stata prodotta con quattro ripetizioni congiunte della parte (c), infine la parte (e) è stata composta allo stesso modo servendosi della parte (d).

Se non riuscite nello sforzo di scrivere un programma che disegni queste figure, non ne siate troppo delusi. È molto utile studiare l'operato del programma HILBERT inserendo un'istruzione READLN appena prima di ciascuna istruzione MOVE del programma. A questo punto potete guardare lo sviluppo dell'azione linea per linea ad un ritmo che vi permetta di mantenere traccia dell'azione mentre il programma gira.

Problemi

PROBLEMA 4.1:
Rispondete con parole vostre alle seguenti domande:

Quali sono le condizioni che vi possono condurre ad usare lo stesso nome per variabili differenti e distinte fra loro, in punti differenti del programma?
Descrivete la regola che vi permette di fare ciò.

In che modo un programma risulta semplificato dalla presenza di procedure nidificate all'interno di altre?
Spiegate il significato di quanto affermato.

In che modo vi può essere d'aiuto nello scrivere un programma il ridurre il più possibile il numero di variabili a cui si possono riferire in comune parti separate del programma?
Come sarebbe scritto un programma che risponda a questa esigenza?

Descrivete le differenze fondamentali dell'azione di un parametro variabile dall'azione di un parametro valore.
A che scopo viene usato ciascuno dei due parametri?

Qual è la differenza fra una procedura e una funzione?
Quali sono i passi principali per trasformare una procedura in funzione?

Cosa distingue una procedura o funzione recursiva da una procedura o funzione ordinaria?

In quali condizioni pensereste di usare una procedura o funzione recursiva al fine di semplificare la soluzione di un problema?

Cosa è un'istruzione CASE e come opera?
Quando risulta utile l'uso di un'istruzione CASE per semplificare un programma?

CAPITOLO 5

LAVORANDO CON I NUMERI

1. Obiettivi

L'obiettivo principale di questo capitolo è di insegnarvi a risolvere, tramite elaboratore, dei problemi abbastanza semplici che hanno a che fare con dei numeri. Come in tutti gli altri capitoli si enfatizzerà, comunque, la problematica della programmazione e della soluzione dei problemi e non della matematica.

1a. Sviluppo di una prima comprensione di come la "macchina elaboratore" lavora: spiegando cioè, come, con molti piccoli passi, si possono portare a termine dei lavori più ampi.

1b. Studio della rappresentazione binaria dei numeri all'interno dell'elaboratore e delle relazioni con il suo modo di lavorare. Apprendimento ad usare la rappresentazione binaria di numeri e caratteri.

1c. Apprendimento a lavorare con espressioni aritmetiche, in modo abbastanza generale, con l'elaboratore e a convertire queste espressioni dalla forma algebrica a quella "capita" dall'elaboratore, e viceversa. Apprendimento ad usare le regole di precedenza.

1d. Uso delle variabili REAL.

1e. Comprensione di come gli errori di arrotondamento limitino l'accuratezza dei calcoli eseguiti da un elaboratore, e di come il valore di un'espressione dipende dall'ordine in cui l'espressione è scritta.

1f. Studio e comprensione di tutti i programmi riportati a titolo esemplificativo.

2. Premessa

Sono veramente poche le persone che lavorano con un elaboratore, sia che si inte-

ressino d'arte e letteratura, scienza e ingegneria, prenotazioni e gestione o qualsiasi altra cosa, che possano evitare di imparare i principi base sul come manipolare i numeri. Per gli studenti di scienze naturali ed ingegneria i problemi numerici sono quelli base ed infatti dovranno studiare "calcolo numerico" che è una branchia della matematica. Gli studenti di scienze sociali possono usare l'elaboratore per problemi non numerici, ma certamente lo useranno in applicazioni numeriche in statistica. Parecchi altri dovranno conoscere quel tanto che basta, nel campo delle applicazioni numeriche, per risolvere semplici problemi di aritmetica.

Questo capitolo cerca di porre le basi per calcoli numerici al livello necessario per tutti gli studenti. Alcuni dei programmi riportati, sono presi da applicazioni matematiche; per gli studenti con orientamento matematico sarà interessante vedere come questi esempi sono legati agli altri lavori che stanno facendo. Comunque, se non avete un indirizzo matematico, non dovreste avere difficoltà a capire il problema che il programma campione cerca di risolvere. In entrambi i casi, a noi interessa, soprattutto, aiutarvi a porre le basi per risolvere problemi futuri. Il fatto di usare esempi numerici o non numerici, a livello di questo testo propedeutico, non porta a sostanziali differenziazioni nel metodo.

Il nostro primo compito sarà quello di dare una breve sintesi del metodo usato da quasi tutti gli elaboratori "*digitali*" per manipolare i numeri. Il termine "digitale" implica che i dati elaborati sono costituiti da numeri. Il termine "*elaboratore analogico*" si applica a quelle macchine che usano segnali elettrici, leve meccaniche, pressioni pneumatiche o altri mezzi simili per simulare il comportamento di altri sistemi. La stragrande maggioranza degli elaboratori attualmente in uso sono di tipo digitale; non dedicheremo altra attenzione agli elaboratori analogici.

Potreste essere sorpresi nel sentire che tutte le informazioni manipolate da un elaboratore digitale sono rappresentate come numeri. Solitamente i numeri sono costituiti da una sequenza di cifre binarie. Una cifra binaria ("*bit*" nel gergo informatico) può valere solo 0 o 1 (diversamente dal sistema decimale nel quale una cifra può avere uno qualsiasi dei 10 valori 0, 1, 2,..., 9). Per trattare informazioni che non sono di per sè numeriche, è necessario accordarsi su qualche schema per convertire dati non numerici in "*codici*" numerici che possano poi essere elaborati. Per trattare variabili di tipo STRING come quelle da noi usate, ciascun carattere è convertito in un codice numerico "*ASCII*" (American Standard Code for Information Interchange) che è uno dei codici più usato.

Quando si sceglie di considerare le cifre memorizzate nell'elaboratore come numeri, piuttosto che un qualche codice, solitamente si ha bisogno della macchina per compiere operazioni aritmetiche (somma, sottrazione, moltiplicazione o divisione). L'hardware è capace di eseguire una sola di queste operazioni per volta, e comunque sempre con due soli numeri per volta. Fatta una di queste operazioni, il risultato è ri-

memorizzato nella memoria dell'elaboratore e può così iniziare una nuova operazione. Spesso si vogliono eseguire dei calcoli, di tipo algebrico, che necessitano di più di due numeri. Tutti i più importanti linguaggi di programmazione danno la possibilità di scrivere delle espressioni algebriche in una forma speciale: sarà poi il software di sistema che spezzerà quelle sequenze complicate, in piccoli passi capiti dall'hardware. Nei capitoli precedenti abbiamo già fatto uso di queste forme.

Il nostro secondo compito, in questo capitolo, sarà quello di analizzare l'uso di < espressione aritmetica > in PASCAL e spiegare come un'espressione algebrica è convertita in una serie di istruzioni che l'elaboratore può eseguire. Come vedrete, l'elaboratore non esegue i calcoli aritmetici con un'accuratezza perfetta, per cui bisogna stare attenti sul come tener conto di ciò.

3. Logica base dell'elaboratore

Praticamente tutti gli elaboratori attuali usano dei programmi memorizzati che sono costituiti da una sequenza di passi di computazione veramente semplici. Negli elaboratori che si usavano un quarto di secolo fa, ognuno doveva scrivere dei programmi nei quali ogni singolo passo doveva essere espresso nei programmi stessi. Divenne ben presto chiaro che gli elaboratori avrebbero potuto essere usati anche per semplificare il processo di generare programmi in "*linguaggio macchina*". Mentre le macchine possono capire solo delle sequenze codificate di numeri, per le persone è molto più facile ricordare elementi di un programma identificati da parole familiari, o, perlomeno, da sequenze di lettere e cifre. Inoltre, era possibile raggruppare delle sequenze semplici, usate spesso, in operazione di "*più alto livello*" più vicine ai problemi che la gente voleva risolvere che al progetto hardware. Questo ha portato alla scrittura di programmi traduttori chiamati "*compilatori*", ai quali abbiamo già fatto riferimento, che convertono programmi scritti in linguaggi ad alto livello (come PASCAL, FORTRAN, BASIC,...) in sequenze di codici macchina che possono essere capite dalle macchine stesse. Uno dei primi compiti dati ai compilatori fu quello di tradurre espressioni algebriche in sequenze di istruzioni per eseguire le operazioni aritmetiche.

In figura 5-1 sono rappresentate le principali "*strutture base*" di un sistema di elaborazione. Molti elaboratori attuali sono composti da un "*unità di elaborazione centrale*", spesso abbreviata in "*CPU*", da gestori di dati "*immessi*" od "*emessi*" (IOP), una certa quantità di "*memoria*" ad alta velocità di accesso, memorie di massa a bassa velocità e costo ma a più alta capacità, ed una grande varietà di "*dispositivi periferici*". Sia che la macchina sia grande o piccola, questi vari "*moduli*" indipendenti possono essere sistemati per costituire un "*sistema*" completo in accordo con le esigenze dell'utente. La varietà di tali combinazioni è davvero sbalorditiva. Inoltre,

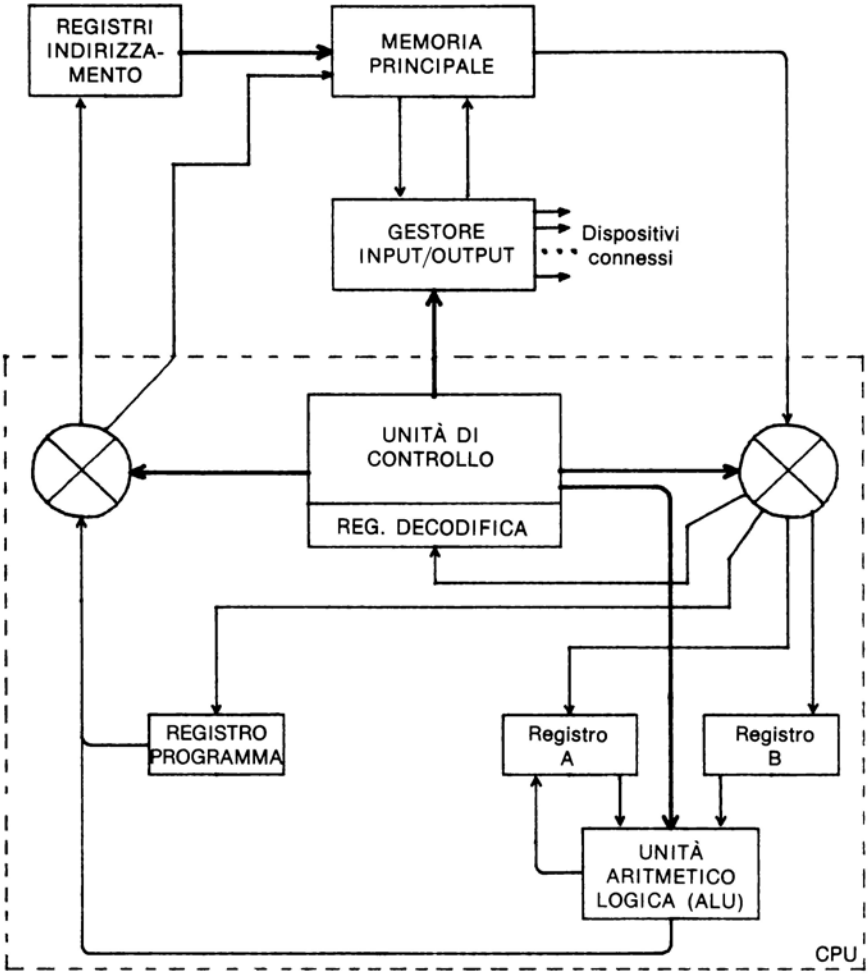

REGISTRI INDIRIZZA-MENTO
MEMORIA PRINCIPALE
GESTORE INPUT/OUTPUT
Dispositivi connessi
UNITÀ DI CONTROLLO
REG. DECODIFICA
REGISTRO PROGRAMMA
Registro A
Registro B
UNITÀ ARITMETICO LOGICA (ALU)
CPU

Figura 5-1

il miglioramento nei progetti dei moduli per sistemi di elaborazione dati è così veloce che possiamo predire, senza ombra di dubbio, che la costituzione dei moduli prodotti fra una decina d'anni differirà sensibilmente da quella attuale. Mentre gli elaboratori del futuro avranno probabilmente parecchie CPU, ciascuna con una sua specifica funzione, sembra verosimile che il funzionamento delle CPU rimarrà simile a quelle degli elaboratori odierni.

Potete pensare alla figura 5-1 come rappresentante un livello di astrazione abbastanza sommario in uno schema che mostra come lavora un qualsiasi elaboratore. Molto spesso, anche quei programmatori che lavorano ancora in linguaggio macchina (o l'equivalente "*linguaggio assemblativo*") conoscono solo un livello più alto di rappresentazione della struttura interna effettiva della macchina.

4. Logica binaria

Praticamente tutti i moderni elaboratori digitali usano una logica binaria. La ragione di questo sta nel fatto che la maggior parte dei dispositivi elettronici più veloci e più a buon mercato scoperti fino ad ora può funzionare nel modo migliore in uno dei due "*stati*" stabili. Ci sono stati molti sforzi per produrre dispositivi con più di due stati stabili, ma i risultati sono stati, nella maggior parte dei casi, meno affidabili e più costosi dei corrispettivi dispositivi a due stati. I più importanti dispositivi binari sono usati:

a) Per memorizzare cifre binarie (0 o 1)

b) Per smistare (fornire un'alternativa all'instradamento)
le informazioni che riportano lo stato dei dispositivi di memorizzazione

c) Per fornire una combinazione logica di "*segnali*" che riportano lo stato dei dispositivi di memorizzazione

Un "*registro*" è costituito da un gruppo di dispositivi di memorizzazione binaria usati assieme. In qualsiasi elaboratore alcuni (pochi) registri sono costruiti usando componenti elettronici molto veloci ma relativamente cari. Nella figura 5-1 questi includono i registri A e B, il registro programma, il registro codice ed il registro di indirizzamento della memoria. Spesso un elaboratore contiene molti registri più veloci di quelli per scopi particolari. Un numero molto più alto di registri è raggruppato assieme nel sistema "*memoria principale*". Questi sono sempre fatti con componenti che permettono di "impilare" un gran numero di registri in un volume piccolo, e che permettono di usare un basso numero di dispositivi di smistamento per "*indirizzare*" (i.e. per connettere) un solo registro della pila per volta. Solitamente tutti questi registri

contengono lo stesso numero di cifre binarie (chiamate "*bits*"). Il contenuto di un registro di memoria è chiamato "*parola*", ed il numero di bits in una parola è detto "*lunghezza della parola*". Per quanto riguarda l'hardware, l'indirizzo di una parola in memoria è il numero del registro contenente la parola.

La velocità dei vari registri dipende dalle dimensioni e dal prezzo dell'elaboratore: i più veloci sono i più cari. Il tempo necessario per memorizzare od estrarre una parola dalla memoria principale va da circa 0,0000001 secondi (100 nanosecondi) a circa 0,00001 secondi (10 microsecondi=10.000 nanosecondi). I registri veloci della CPU sono, tipicamente, 10 volte più veloci dei registri della memoria principale (10 nanosecondi rispetto ad 1 microsecondo). Su molte macchine i segnali possono passare attraverso 10 componenti logici o di diramazione nel tempo speso per accedere ad un'informazione in un registro veloce della CPU. Le velocità dei vari componenti usati in un elaboratore sono scelte per fornire un bilancio globale ragionevole: serve veramente a poco avere dei registri della CPU particolarmente veloci su macchina con memoria lenta; similmente è di poca utilità avere una memoria con tempi di accesso di 100 nanosecondi se i registri della CPU non sono più veloci di 1000 nanosecondi (i.e. 1 microsecondo).

Il significato delle informazioni memorizzate nei vari registri dell'elaboratore (la configurazione di bits in ogni registro) dipende dall'uso che viene fatto della informazione. In prima approssimazione, l'informazione può essere "*operando*" (dato) o "*codice*". Nei primi elaboratori solo gli operandi potevano essere posti nei registri della memoria: "codice" faceva riferimento ad una sequenza di configurazioni di bits usata dall'unità di controllo per posizionare dei deviatori che a loro volta avrebbero determinato come le informazioni sarebbero passate da un registro all'altro. Nelle prime macchine i codici erano rappresentati da connessioni "*filate*". La fantastica crescita nell'uso dei moderni elaboratori digitali è legata ad una osservazione di John Von Neumann, subito dopo la 2ª guerra mondiale: egli asserì che sia il codice che gli operandi potevano essere depositati nella stessa memoria principale e che entrambi avrebbero potuto essere manipolati dalla logica dell'elaboratore.

Un "*operando*" è una voce di un'informazione che l'elaboratore deve manipolare in qualche modo. Il significato di una particolare configurazione di bits, memorizzati in un registro, dipenderà dall'uso che si vuol fare di quella parola. L'uso più familiare di queste configurazioni è quello di rappresentare dei numeri. I più semplici numeri che si possono depositare in un registro fanno parte dell'insieme degli INTEGER (INTERI):

.... −2, −1, 0, 1, 2, 3, 4 ...

In genere i bits di un registro, su carta, vengono rappresentati come le cifre di un numero decimale: cioè, da sinistra a destra, con la cifra meno significativa nell'ultima

posizione a destra. Perciò un registro contenente l'equivalente binario del numero decimale 10 avrà la seguente configurazione:

0 0 0 0 1 0 1 0

Si è assunto che la lunghezza della parola sia di 8 bits.

La posizione di ciascun bit nel registro ne determina il peso come la corrispondente cifra nel sistema di numerazione binaria. Per cui il numero riportato sopra è equivalente a:

$0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$

(Chiarimento per i lettori non abituati a questa notazione:

2^2 significa: 2 elevato alla seconda (i.e. $2*2 = 4$)
2^3 significa: 2 elevato alla terza (i.e. $2*2*2 = 8$)

Per capire la linea sopra prendete una tavola con le potenze di 2 con esponente da 1 a 7. 2 elevato a 0 vale 1.)

Questo registro a 8-bit potrebbe contenere un qualsiasi intero appartenente all'insieme:

0, 1, 2, 3, 4, ..., 255

Parecchi dei mini-elaboratori attuali hanno una parola di memoria di 16 bits, che permette la memorizzazione di interi dell'insieme:

0, 1, 2, 3, 4, ..., 65535

Questo è vero per quasi tutti i mini o micro-elaboratori che voi presumibilmente state usando con questo libro. Gli elaboratori IBM della linea 360/370 usano una parola di 32 bits; i Burroughs B6700/7700 di 48 bits; i Control Data Cyber di 60 bits... e così via. Come verrà descritto nella sezione 6, in tutte queste macchine sono date delle possibilità per rappresentare i numeri negativi.

Ora per capire l'osservazione di Von Neumann possiamo ritornare alla figura 5-1. Procedendo passo passo nel calcolo, i numeri di cui abbiamo parlato vanno dalla memoria ai registri A o B via deviatore (mostrato nel disegno con dei cerchi con all'interno una croce). Se invece di rappresentare un numero, l'informazione nella parola di memoria rappresenta un "codice" (vale a dire istruzioni codificate per l'elaboratore) allora il deviatore incanala la parola di codice al registro codice (per la decodifica dell'istruzione). Una tipica unità di controllo può permettere fino a 256 diverse opera-

zioni. Nella prossima sezione descriveremo alcuni dei generi di operazioni possibili. I progettisti hardware usano gli 8 bits della parola memorizzata nel registro codice per determinare quale delle 256 possibili operazioni verrà eseguita in ogni passo di elaborazione. Potete pensare associato ad ogni operazione un numero che va da 0 a 255!

Per ogni passo di elaborazione, il CONTROLLO dapprima estrae un'altra parola di controllo di 8-bit, poi posiziona uno o entambi i deviatori rappresentati nel disegno ad una certa combinazione di registri. Nello stesso tempo l'unità aritmetico-logica (ALU) è predisposta per svolgere una particolare azione del suo repertorio (Somma, Moltiplicazione, AND, Negazione, etc — vedete la sezione 5). Il risultato di questa operazione logica o aritmetica è quindi depositato in memoria e nel registro A. Infine, al termine del passo di elaborazione, il valore numerico della informazione nel Registro Programma è incrementato di 1, ed il valore risultante è posto nel Registro di Indirizzamento della Memoria. La CPU è quindi pronta per un nuovo passo di elaborazione. Avrete notato che non abbiamo parlato di come il Registro di Indirizzamento della Memoria viene posizionato per far sì che il risultato dell'operazione dell'ALU sia memorizzato nella parola corretta. Sulle macchine più piccole questo indirizzo sarà stato posizionato come risultato di un passo di elaborazione separato. Su macchine più grandi ci può essere una logica addizionale all'interno della CPU, o informazioni addizionali nella parola di codice memorizzata nel Registro Codice per far sì che l'indirizzo venga posizionato nel tempo di un passo operativo (spesso chiamato "*ciclo*").

Dettagli nel gestore immissione/emissione dati sono aldilà degli scopi di questo libro. L'I/OP è praticamente una CPU speciale dedicata a trasmettere dati *ad* un dispositivo esterno in "*emissione*", o a ricevere dati *da* un dispositivo esterno in "*immissione*". Come esempi di questi dispositivi si possono portare: la tastiera che state usando, o un lettore di schede come immissione, uno schermo, una stampante seriale od una parallela come emissione. Certi dispositivi come quelli che trattano dischi morbidi, cassette, o nastri magnetici, possono lavorare sia in immissione che emissione.

Avrete forse notato che il CONTROLLO può anche mandare informazioni dalla memoria al Registro Programma. Solitamente il codice di un programma è memorizzato in un gruppo di parole adiacenti in memoria. Il Registro Programma punta ad uno di quegli indirizzi in ogni passo di elaborazione, e l'elaborazione procede in modo sequenziale da un indirizzo al prossimo ad ogni passo. Tuttavia, è a volte necessario alterare il flusso del controllo, deviandolo dalla sequenzialità ordinaria. Questo può, ad esempio, succedere nel corso dell'esecuzione di istruzioni WHILE, REPEAT o FOR. Per far questo il CONTROLLO scarica da memoria a Registro Programma un indirizzo completamente nuovo. La sequenza di elaborazione salta quindi ad una nuova locazione e da lì procede ancora in modo sequenziale. Nella sua forma più semplice questa operazione di salto è rappresentata dal costrutto GO TO dei linguaggi ad alto livello. Nel capitolo 11 discuteremo l'istruzione GOTO di PASCAL.

5. Operazioni logiche ed aritmetiche

L'Unità Aritmetico-Logica (ALU), mostrata in figura 5-1, è progettata per accettare due operandi (2 parole) dai registri A e B e per combinare opportunamente questi operandi per produrre un unico risultato. L'unità di CONTROLLO determina quale delle parecchie possibili combinazioni sarà il risultato, come indicato dalla linea più marcata dal CONTROLLO all'ALU. Per esempio, l'informazione contenuta nel registro A (che può essere designata racchiudendo "A" in parentesi, come: (A)) può essere sommata al contenuto del registro B (designato: (B)) dando la somma algebrica delle due quantità. Se usiamo come indicatore generico di un singolo registro di memoria "(M)", senza preoccuparci, per ora, del suo indirizzo, allora l'operazione di ADD può essere scritta:

$$(M) \leftarrow (A) + (B)$$
$$(A) \leftarrow (A) + (B)$$

in cui la freccia orientata a sinistra indica un assegnamento di valore. Usiamo una freccia orientata a sinistra per evitare confusione, in questo libro, con l'uso di ":=" in PASCAL; dal momento che le operazioni descritte in questa sezione non possono, in generale, essere eseguite direttamente usando istruzioni PASCAL.

Le formule riportate sopra dicono che l'operazione di addizione della ALU fa sì che il contenuto della locazione di memoria M, e del registro A, siano sostituiti dalla somma aritmetica del contenuto del registro A e del registro B, così come essi erano all'inizio del passo di elaborazione. Alla fine del passo il vecchio contenuto di M ed A è distrutto, sono cioè riscritti con il nuovo valore.

Nel prossimo passo di elaborazione, il nuovo valore di (A) potrebbe essere usato come uno dei due valori da consegnare alla ALU per qualche altra operazione. Su parecchie macchine questo farebbe risparmiar tempo evitando la necessità di andare nella memoria principale per entrambi gli operandi quando si usa la ALU; spesso è anche possibile non memorizzare il risultato in memoria, risparmiando perciò altro tempo (sempre che la logica del programma lo permetta).

Su grossi elaboratori, la ALU generalmente può trattare le operazioni di SOMMA, SOTTRAZIONE, MOLTIPLICAZIONE e DIVISIONE su operandi numerici di parecchi tipi. Su piccoli elaboratori, la MOLTIPLICAZIONE, la DIVISIONE come altre operazioni complesse possono essere simulate da un programma che usa operazioni meno complesse. Solitamente la ALU può anche eseguire operazioni logiche di vario tipo bit per bit. Per esempio l'operazione:

$$(A) \leftarrow (A) \text{ AND } (B)$$

significa che per ogni bit della parola, il risultato contiene un 1 se entrambi (A) e (B) contengono un 1 nella stessa posizione, quando inizia il passo di elaborazione. Diversamente il bit risultato è 0: cioè,
se prima dell'inizio dell'operazione A conteneva 01010100 e
B conteneva 10101100 alla fine A conterrà 00000100.

Similmente l'operazione di OR mette ad 1 un bit del risultato se il corrispondente bit in (A) oppure in (B) è a 1.

Un altro tipo di operazione logica permette di confrontare gli operandi (A) e (B). Per esempio:

(A) ← 1 IF (A) = (B)

In altre parole se quando inizia l'operazione i contenuti dei registri A e B sono uguali, allora il bit meno significativo del registro A è posto ad 1 mentre tutti gli altri bit sono a zero. Molte macchine hanno anche delle operazioni logiche che permettono di far "scorrere" di un certo numero di posizioni la configurazione di bits dei registri A o B. Infine ci sono operazioni in cui l'unità di CONTROLLO in modo esplicito "estrae" informazioni dalla memoria e le pone in un registro, per esempio:

(A) ← (M)

l'operazione di "memorizzazione" fa invece l'opposto:

(M) ← (A)

Le ALU di parecchie macchine eseguono anche certe operazioni particolari, comunque le operazioni trattate in questa sezione fanno parte del repertorio base di tutti gli elaboratori odierni. Dal momento che sapete che gli elaboratori sono usati per risolvere problemi molto complessi e con grandi quantità di dati, potreste stupirvi che ciò possa essere fatto sfruttando sequenze di operazioni cosi semplici. La capacità di scomporre dei grossi problemi in sequenze di tal genere è la ragione principale per cui gli elaboratori possono essere usati nei più svariati campi di applicazione in cui si trovano oggi impiegati.

6. Rappresentazioni dei numeri

Avrete forse già capito che il metodo abbastanza semplice, discusso nella quarta sezione, per memorizzare dei numeri interi in forma binaria non è sufficiente per mol-

ti tipi di elaborazione. Due altri mezzi sono forniti su tutti gli elaboratori per scopi generali, e cioè:

a) Trattamento di numeri negativi

b) Trattamento di numeri più grandi del massimo intero contenuto nella parola e di numeri più piccoli di 1.

Sfortunatamente sono tre i metodi principali usati sugli elaboratori per rappresentare i numeri negativi. Sebbene parecchi utenti non dovranno mai sapere nulla sul come i numeri negativi sono rappresentati all'interno della macchina, ci sono parecchie situazioni che richiedono questa conoscenza. La differenza tra i vari costruttori per quanto riguarda la rappresentazione dei numeri negativi spiega alcuni dei poco chiari problemi di convertire programmi scritti in un linguaggio "standard" come FORTRAN da una macchina ad un'altra.

Abbiamo già visto come il numero decimale intero 10 potrebbe essere rappresentato in un elaboratore con una parola di 8-bit. Mostriamo ora come il numero negativo 10 (vale a dire −10) è rappresentato in una macchina a 8-bit con i tre metodi di rappresentazione dei numeri negativi attualmente usati:

a) Valore-segno 10001010
Il bit più a sinistra serve come bit di "segno"; possono perciò essere rappresentati numeri da −127 a +127. −0 è distinguibile da +0, fatto abbastanza interessante in certe applicazioni matematiche.

b) Complemento a 1 11110101

Ciascun bit del numero positivo è complementato: gli 0 diventano 1 e gli 1 diventano 0. Si possono rappresentare numeri che vanno da −127 a +127 e lo zero negativo è distinguibile da +0.

c) Complemento a 2 11110110

È ottenuto sommando 1 al complemento a 1. Si possono rappresentare numeri che vanno da −128 a +127 e −0 non è distinguibile da +0.

In tutti e tre i metodi, il bit di *"ordine più alto"* (il bit più a sinistra della parola) è posto a 1 se il numero è negativo e a 0 se positivo. La scelta del metodo di rappresentazione è un problema di ingegneria costruttiva. La Burroughs ha scelto valore-segno per il B6700; la Digital Equipment Corporation ha scelto il complemento a 2 per il PDP-11 mentre la CDC usa il complemento a 1 sulle sue macchine grosse.

Dei numeri troppo grandi o troppo piccoli per essere rappresentati come interi possono essere rappresentati in *"virgola-mobile"*. Questo significa che la rappresentazione binaria della virgola decimale "si muove" in relazione ai bits depositati in memoria.

Un numero in virgola mobile è costituito da due *"campi"* interi, uno chiamato *"mantissa"* e l'altro *"esponente"*. Se M è il valore intero della mantissa, ed E il valore intero dell'esponente, allora il valore rappresentato dal numero in virgola mobile è:

$$M * 2^E$$

su parecchie macchine, o su certe:

$$M * 8^E$$

vale a dire 2 o 8 elevati all'esponente E. Ciò significa 2 o 8 moltiplicati per se stessi (E−1) volte. Sia la mantissa che l'esponente, od entrambi, possono essere negativi usando uno dei metodi di cui abbiamo parlato prima.

Un numero in virgola mobile, per essere di un valore "accettabile", deve occupare, includendo sia mantissa che esponente, almeno 32 bits. Se voi, mentre studiate questo testo, usate un micro-elaboratore avrete quasi sicuramente un numero rappresentato su 32 bits. Il numero di bits assegnati alla mantissa determina la precisione aritmetica e cioè il numero di "posti" binari o decimali nel numero in virgola mobile. Il numero di bits dell'esponente determina il valore massimo e minimo del numero in virgola mobile. Su macchine che usano potenze con base 8, la mantissa deve includere degli zero binari in coda; queste macchine con meno bits di esponente permettono di coprire un intervallo più grande a spese di 2 bits di precisione persi nella mantissa.

In PASCAL come in FORTRAN, ALGOL e PL/1 i numeri in virgola mobile sono manipolati con variabili di <tipo> *"REAL"*. In PASCAL potete convertire un numero REAL R in un INTEGER I contenente solo la parte intera di R usando

```
I := TRUNC (R)
```

mentre usando

```
I := ROUND (R)
```

potete convertire R nell'intero *più vicino*.

COBOL è orientato all'uso di numeri decimali. Molte implementazioni realizzano

questo, usando valori interi durante l'elaborazione effettiva ed inserendo poi la virgola decimale quando e dove serve. APL e BASIC evitano di preoccupare l'utente con la distinzione tra variabili INTEGER e REAL.

Come dovrebbe già esservi chiaro, noi dichiariamo una <variabile> REAL in PASCAL con la sintassi che voi avete già visto usare per variabili STRING, BOOLEAN, CHAR e INTEGER. Per esempio:

```
VAR X,Y,Z: REAL;
```

Diamo ora qualche esempio di <costanti> REAL accettabili in PASCAL:

```
1.234
0.5
43210.5
1.5E-3 (*equivalente a 0.0015*)
-1.E+6 (*equivalente ad 1 milione negativo*)
```

La sintassi per costruire <numeri con segno> in PASCAL è riportata in figura 5-2. Nei microelaboratori più diffusi la mantissa di un numero REAL può contenere fino a 6 cifre decimali.

Mentre le funzioni interne ROUND e TRUNC sono necessarie per convertire in valore REAL per assegnarlo ad una variabile INTEGER, PASCAL permette di assegnare un valore INTEGER ad una variabile REAL: durante l'assegnazione converte il valore INTEGER in forma REAL. Diversamente è possibile solo assegnare valori REAL a variabili REAL e valori INTEGER a variabili INTEGER.

7. Espressioni aritmetiche — Assegnazione di valori

Con l'eccezione di APL, tutti gli altri più importanti linguaggi di programmazione usano le tecniche di trattamento delle espressioni algebriche nate con FORTRAN. Ci sono comunque dei sottili punti di differenza tra i linguaggi e dovreste quindi fare molta attenzione nel passare da PASCAL ad un altro linguaggio. In questo capitolo riportiamo le tecniche base.

Poichè la CPU (figura 5-1) ha solo due registri principali per gli operandi, è chiaramente necessario scomporre delle espressioni algebriche complicate in una sequenza di operazioni più semplici. Per esempio, per calcolare il valore della espressione:

$$\frac{X+Y}{C-D} \qquad (7\text{-}1)$$

sarà probabilmente seguita la seguente sequenza:

(T1) ← (X) + (Y)
(T2) ← (C) − (D)
(Registro A) ← (T1) : (T2)

<numero con segno>

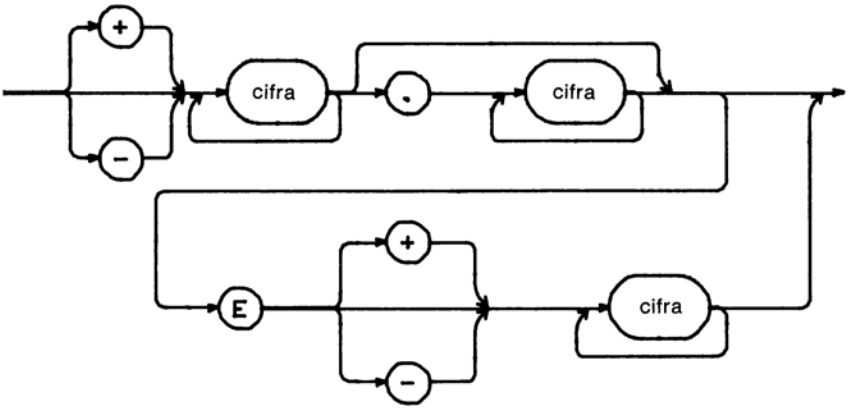


Figura 5-2

che in modo discorsivo diviene:

Somma il contenuto della locazione X al contenuto della locazione Y ed assegna il risultato alla locazione T1

Sottrai il contenuto della locazione D dal contenuto della locazione C ed assegna il risultato alla locazione T2.

Dividi T1 per T2 e poni il risultato nel registro A.

Naturalmente abbiamo aggregato più passi di elaborazione in ognuna delle tre formule. Per esempio, la prima potrebbe essere ulteriormente scomposta in:

(Registro A) ← (X)
(Registro B) ← (Y)
(T1) ← (Registro A) + (Registro B)

In questa discussione gli identificatori T1 e T2 rappresentano delle locazioni di me-

moria che saranno normalmente usate dal compilatore senza nessuna visibilità per l'utente. In effetti il compilatore non userà "T1" o "T2" come identificatori, ma userà dei numeri invece che simboli così che non sorgeranno conflitti se voi userete "T1" o "T2" come variabili nel vostro programma. Questa discussione dà un esempio di compiti di "contabilizzazione" di cui si fa carico il compilatore per snellire il lavoro del programmatore.

Benchè sia ovvio quale operazione facciano eseguire i simboli ADD ('+'), SUBTRACT ('–'), MULTIPLY ('*'), e DIVIDE ('/'), per la divisione ci sono delle complicazioni. Se dividete due interi, ad esempio 5/3, dovreste aspettarvi – con 6 cifre di precisione – il numero REAL 1,66667. Poichè la divisione generalmente dà un risultato REAL, c'è cioè un resto, un'< espressione aritmetica > contenente '/' produce un risultato di < tipo > REAL. Se volete o l'INTEGER più vicino od il resto della divisione, questo è attendibile in PASCAL mediante due altri operatori. Userete *"DIV"* al posto di '/' se volete solo il quoziente INTEGER a *"MOD"* al posto di '/' se volete solo il resto.

In PASCAL, DIV e MOD possono solo essere usati per dividere un'espressione INTEGER con un'altra ed entrambi danno un risultato INTEGER. Un'espressione che contiene '/' è automaticamente di < tipo > REAL anche se tutti i componenti dell'espressione sarebbero altrimenti di < tipo > INTEGER. La sintassi permette che '/', DIV e MOD siano usati ovunque è possibile usare '*' in un'espressione aritmetica, e tutti e tre hanno la stessa precedenza di '*'.

Supponendo I < variabile > INTEGER ed R REAL, ecco alcuni esempi:

```
R := 10/4       (*il valore di R diviene 2.50000*)
I := 10 DIV 4   (*il valore di I diviene 2      *)
I := 11 MOD 4   (*il valore di I diviene 3      *)
I := 20 MOD 4   (*il valore di I diviene 0      *)
R := 20/4       (*il valore di R diviene 5.00000*)
```

Se voi aveste visualizzato, tramite WRITELN (R), il risultato della divisione avreste potuto ottenere, in dipendenza dal tipo di elaboratore e dalle caratteristiche del linguaggio, una delle due stringhe seguenti:

```
5.00000
4.99999
```

Naturalmente il primo risultato è corretto mentre il secondo differisce dal valore esatto per una quantità uguale ad *1 bit* nella posizione di valore più basso nel registro che contiene il valore REAL. Quando esiste un errore, questo sorge perchè è necessario rappresentare un numero REAL con un numero limitato di bits nei registri dell'e-

laboratore. In certi casi il risultato di un'operazione di divisione richiederà più bits di quanti ne fornisce l'elaboratore se si vuole ottenere una rappresentazione accurata. È quindi necessario o *"arrotondare"* il risultato al valore più vicino che può essere espresso da quel numero di bits, o semplicemente considerare tutti i bits che non possono essere rappresentati come degli 0. La seconda alternativa è detta *"Troncamento"* e ne abbiamo già brevemente parlato. In altre parole i bits che non possono essere contenuti in memoria sono molto semplicemente buttati via.

Per assegnare il risultato del calcolo che valuta l'espressione (7—1) ad una variabile RISULTATO, dovremo usare la seguente espressione di assegnamento:

```
RISULTATO := (X + Y) / (C – D)
```

Come vedete abbiamo, diversamente dalle rappresentazioni algebriche tradizionali, posto tutti i termini dell'espressione (7—1) su un'unica linea. Questo perchè molti dispositivi di immissione degli elaboratori sono progettati per trattare gruppi di caratteri, riga per riga, ed è perciò molto scomodo associare informazione sulle stesse colonne e su righe adiacenti.

Le convenzioni dicono che il compilatore dovrebbe *"scandire"* una linea immessa in modo ordinato da sinistra a destra. Deve anche decidere l'ordine in cui più operazioni devono essere eseguite quando si calcola il valore di un'espressione. Per esempio, l'espressione:

```
X + Y / C – D
```

potrebbe portare a risultati ambigui se non ci fosse una regola che dice se la divisione debba essere fatta prima della somma o viceversa. Per chiarire questo, supponete di avere la seguente espressione intera:

```
5 + 1 / 4 – 2
```

Il valore di questa espressione è 3 o 3.25 o —0.5? L'ambiguità è eliminata dalla convenzione che stabilisce che le operazioni di moltiplicazione e divisione vengono eseguite prima di quelle di somma e sottrazione, in accordo con le regole di precedenza.

Nell'espressione riportata sopra la divisione è eseguita per prima, dando:

```
5 + 0.25 – 2
```

Si dice che la moltiplicazione e la divisione hanno una maggior *"precedenza"* ri-

spetto all'addizione o sottrazione. Per operazioni della stessa precedenza il calcolo è fatto da sinistra a destra. Perciò:

```
5 + 0.25 – 2
```

diviene:

```
5.25 – 2
```

ed il risultato finale è

```
3.25
```

Il nostro esempio originale (espressione 7–1) mostrava però che l'addizione e la sottrazione avvenivano prima della divisione. Questo è possibile in quanto esiste una convenzione che stabilisce che il valore di un'espressione racchiusa tra parentesi viene calcolata completamente prima di considerare le operazioni fuori parentesi. Nel caso di parentesi racchiuse tra parentesi per prima viene valutata l'espressione più interna. L'espressione più interna è sostituita da un valore semplice salvato temporaneamente dall'elaboratore, e le parentesi che la racchiudevano scompaiono. Il processo di calcolo si ripete quindi per l'espressione risultante racchiusa tra la prossima copia di parentesi ed il processo continua fino a che non sono sparite tutte le parentesi.

Un esempio chiarirà quanto detto:

```
((X + 5) * X + 8) * X + 7                    (7–2)

(T1) ← (X) + 5
(T2) ← (T1) * (X)
```

e così via. Abbiamo ora ridotto l'espressione a:

```
(T2 + 8) * X + 7

(T3) ← (T2) + 8
(T4) ← (T3) * X
(T5) ← (T4) + 7
```

ed il calcolo è completato.

Le regole di precedenza dicono dunque l'ordine nel quale il compilatore valuterà le espressioni aritmetiche. Le regole legate alle parentesi vi permettono di variare l'ordine per ottenere risultati diversi. Spesso un programmatore troverà seccante il fatto

di verificare l'ordine di elaborazione di un'espressione senza parentesi. A volte è più semplice, anche se può sembrare ridondante, forzare l'ordine dell'elaborazione inserendo coppie di parentesi.

Un'altra ragione per usare le parentesi nelle espressioni può semplicemente essere quella di rendere più facilmente leggibili e capibili quelle espressioni alle persone. Le parentesi permettono di considerare una complicata espressione aritmetica come composta da un numero di *"moduli"* più piccoli, ognuno dei quali è esso stesso una espressione. Ancora una volta, vediamo che le persone possono far fronte con successo a strutture complicate se queste sono spezzate in blocchi di modeste dimensioni comprensibili senza un grande sforzo di analisi. Quindi lo stesso concetto di strutturazione delle espressioni aritmetiche si applica alle istruzioni di un programma che può essere suddiviso in moduli usando procedure ed istruzioni composte.

8. Un programma campione — Conversione decimale -binario

In questa, e nelle prossime sezioni, diamo dei programmi campioni che usano i principi trattati nella prima parte di questo capitolo. Ormai conoscete abbastanza PASCAL per riuscire ad usarlo come mezzo per imprimere concetti logici che sarebbe difficile esprimere in Italiano. Studiate questi programmi con questa idea in mente, e cercate di capire cosa fanno. Se vi trovate invischiati in problemi, un buon metodo per liberarvi è quello di cercare di far girare i programmi campioni sull'elaboratore per ottenere gli stessi risultati dichiarati nel libro. Inserite quindi delle istruzioni WRITE in punti strategici per aiutarvi a chiarire in dettaglio come prosegue l'elaborazione.

Il programma DECBIN esegue la conversione di numeri decimali in notazione binaria. In risposta al suggerimento, battete una costante intera non minore di 0 e non maggiore di 1023 e terminate con <RET>. Il programma risponde visualizzando per primo l'equivalente binario, e quindi, sulla stessa linea, il numero decimale. Per esempio:

```
DECBIN
INTRODUCI UN INTERO TRA 0 E 1023
0000001000 =   8
0000100000 =  32
0000111111 =  63
0010000011 = 131
```

Questo programma si basa sul fatto di determinare se il valore di DEC è maggiore o uguale al *"peso"* di ogni cifra binaria. Se non è più piccolo del peso del bit allora è visualizzato '1' ed il peso è sottratto da ciò che rimane in DEC.

Si analizza quindi il peso immediatamente inferiore dividendo BITVAL per 2 nella ventiquattresima riga del programma. Continuando così l'ultimo bit sarà stampato ed il valore di DEC sarà azzerato.

```
 1: PROGRAMMA DECBIN;
 2: VAR DEC: INTEGER;
 3:
 4: PROCEDURE STAMPABIN(DEC:INTEGER);
 5: NAR BITVAL,RESTO:INTEGER;
 6: BEGIN
 7:   IF DEC>=1024 THEN
 8:     WRITELN('MI SPIACE', DEC, 'È TROPPO GRANDE')
 9:     ELSE
10:     BEGIN
11:       BITVAL:=512; (* 2 elevato alla nona = 512*)
12:       RESTO:=DEC;
13:       WHILE BITVAL>0 DO
14:       BEGIN
15:         IF RESTO>=BITVAL THEN
16:         BEGIN
17:           WRITE('1');
18:           RESTO:=RESTO-BITVAL;
19:         END ELSE
20:           WRITE('0');
21:         BITVAL:=BITVAL DIV 2;
22:         (* prossima potenza inferiore di 2*)
23:       END;
24:       WRITELN('=', DEC);
25:     END (*DEC nell'intervallo permesso*)
26:   END (*STAMPABIN *);
27:
28: BEGIN (*programma principale*)
29:   WRITELN('DECBIN');
30:   WRITELN('INTRODUCI UN INTERO TRA 0 E 1023*);
31:   READLN(DEC);
32:   WHILE DEC>=0 DO
33:   BEGIN
34:     STAMPABIN(DEC);
35:     WRITELN;
36:     WRITELN('INTRODUCINE UN ALTRO');
37:     READLN(DEC);
38:   END;
39: END.
```

Per evitare di perdere il valore originale di DEC, per problemi di visualizzazione, il valore è dapprima copiato in RESTO e quindi RESTO serve al posto di DEC per il resto dell'elaborazione. Questo programma può trattare qualunque numero non negativo espresso in meno di 10 bits. L'istruzione in linea 7 blocca l'elaborazione se il numero è troppo grande. Verifiche di questo genere dovrebbero sempre essere previste per programmi che debbano trattare dati che, occasionalmente, potrebbero uscire dell'intervallo "normale" per cui il programma è stato scritto. In questo caso si è deciso di accettare un massimo di 10 bits per rendere la visualizzazione più leggibile che non nel caso in cui si fossero visualizzati tutti i bits di una parola. PASCAL ha delle caratteristiche che riducono lo sforzo necessario per far verifiche di "validità" di questo genere, caratteristiche che si considereranno in dettaglio nel capitolo 9.

ESERCIZIO 5.1:
Scrivete e correggete un programma per visualizzare un intero decimale in binario usando l'algoritmo che segue che differisce da quello usato in DECBIN. Partendo dal valore di posizione più bassa, vale a dire 2 elevato a 0, visualizzate un '1' se la rimanente parte del valore originario è dispari, altrimenti visualizzate 0. Dividete poi per due la parte rimanente usando la divisione intera (DIV). Se il risultato della divisione non è 0, ritornate a verificare se il valore rimanente è dispari. La conversione è completa quando il risultato della divisione è 0.

Notate che questo algoritmo produce per primo il bit di ordine più basso, che è l'opposto della direzione in cui di solito si leggono le cifre di un numero. Il vostro programma potrebbe salvare le cifre generate dal ciclo appena descritto fino a che tutte le cifre sono state generate. Dovrebbe quindi generale le cifre nel loro ordine corretto, con la cifra più significativa sulla sinistra.

Verificate che i numeri binari visualizzati dal vostro programma siano uguali a quelli visualizzati da DECBIN per i seguenti numeri decimali, e per tutti gli altri che ritenete utile provare:

0, 1, 2, 3, 4, 8, 16, 31, 32, 33, 63, 64, 128,
131, 255, 256, 511, 512, 1023

ESERCIZIO 5-2:
Scrivete e correggete un programma che fa l'operazione opposta a quella di DECBIN. Usate una variabile STRING per introdurre un numero binario costituito da una stringa di caratteri '0' e '1'. Approccio che vi consigliamo: accumulate il valore decimale in una variabile INTEGER, DEC; scorrete la variabile STRING da sinistra a destra; per

ogni cifra nel numero binario, moltiplicate DEC per 2, aggiungete poi 1 a DEC se la cifra binaria è '1'.

Per controllare il vostro programma usate i seguenti numeri binari:

0, 1, 10, 11, 100, 1000, 10000, 11111, 100000,
100001, 111111, 1000000, 10000000, 10000011, 11111111,
100000000, 111111111, 1000000000,
1111111111

che sono gli equivalenti binari della sequenza decimale data nell'esercizio 5-1.
Verificate che il vostro programma visualizzi quei numeri decimali per i numeri binari dati nell'elenco.

9. Un programma campione — ALGEBRA

Il programma Algebra è riportato per illustrare 2 punti:

a) Conversione di complicate espressioni algebriche in una forma accettabile per PASCAL

b) Problemi di precisione che possono sorgere per il fatto che la rappresentazione dei numeri REAL , in un elaboratore, è fatta con un numero limitato di bits.

```
 1: PROGRAMMA ALGEBRA;
 2: VAR R,W,X,Y,Z,EPS:REAL;
 3:
 4: PROCEDURE ARROTERR(EPS:REAL);
 5: BEGIN
 6:    R:=((1+EPS) * (1+EPS) - 1)/EPS;
 7:    WRITELN('((1+EPS) * (1+EPS) -1)/EPS= ',R);
 8:
 9:    R:=((1 +2 *EPS + EPS*EPS) -1)/EPS;
10:    WRITELN('((1 + 2*EPS + EPS*EPS) -1)/EPS= ',R);
11:
12:    R:=(2*EPS + EPS*EPS)/EPS;
13:    WRITELN('(2*EPS + EPS*EPS)/EPS= ',R);
14: END (*ARROTERR*);
15:
```

```
16: BEGIN (*PROGRAMMA PRINCIPALE*)
17:    (*INIZIALIZZAZIONE VARIABILI*)
18:    W:=2; X:=5; Y:=4; 2:=10;
19:    EPS:=0.0006;
20:
21:    R:=(X + Y*5)/(X - Y/0.2);
22:    WRITELN('(X + Y*5)/(X - Y/0.2)= ',R)
23:
24:    R:=(((W+3)*W + 2) * W - 10) * W + 4;
25:    WRITELN('(((W+3)*W + 2) *W-10) * W + 4 = ',R);
26:
27:    R:=4*(X+Y)*(X+Y)/(X-Y);
28:    WRITELN('4*(X+Y)*(X+Y)/(X-Y) = ',R);
29:
30:    R:=(1 + Z*Z/X*X)/(1 - Z*Z/X*X) + Y;
31:    WRITELN('(1 + Z*Z)/X*X)/(1 - Z*Z/X*X) + Y = ',R);
32:
33:    R:=(1 + Z*Z/(X*X))/(1 - Z*Z/(X*X)) + Y;
34:    WRITELN('(1 + Z*Z/(X*X))/(1 - Z*Z/(X*X)) + Y = ',R);
35:
36:
37:    WRITELN('BATTI UN NUMERO REALE PICCOLO PER EPS');
38:    READLN(EPS);
39:    WHILE EPS>0.0 DO
40:    BEGIN
41:       ARROTER(EPS);
42:       WRITELN;
43:       WRITELN('BATTINE UN ALTRO');
44:       READLN(EPS);
45:    END;
46: END.
```

```
 1: Visualizzazione associata con il programma ALGEBRA
 2:
 3:
 4: (X + Y*5)/(X - Y/0.2)= -1.666666
 5:
 6: (((W+3)*W + 2) * W - 10) * W + 4 = 32.0
 7:
 8: 4*(X+Y)*(X+Y)/(X-Y) = 324.0
 9:
10: (1+ Z*Z/X*X)/(1 - Z*Z/X*X) + Y = 2.979797
11:
```

```
12: (1+ Z*Z/(X+X))/(1 - Z*Z/(X*X) + Y = 2.333333
13:
14: BATTI UN NUMERO REALE PICCOLO PER EPS
15:
16: 0.000025
17:
18: ((1+EPS) * (1+EPS) -1)/EPS = 2.002716
19:
20: ((1 + 2*EPS + EPS*EPS) -1/EPS = 1.997947
21:
22: (2*EPS + EPS*ESP)/EPS = 2.000025
```

Il programma prima visualizza i risultati del calcolo delle cinque espressioni nelle righe 21, 24, 27, 30 e 33 del programma. Le righe visualizzate sono rappresentate, con numerazione distinte, subito dopo il programma. Potete facilmente verificare la correttezza dei primi tre risultati con carta e penna. La linea 21 calcola la seguente espressione algebrica:

$$\frac{X+5Y}{X-\frac{Y}{0,2}}$$

La linea 24 mostra un metodo efficiente per calcolare la seguente espressione:

$$W^4 + 3W^3 + 2W^2 - 10W + 4$$

Questa espressione è meno efficiente di quella in riga 24 perchè richiede un numero maggiore di moltiplicazioni, che, su piccoli elaboratori, spesso richiedono tempi lunghi. Come esercizio fate le moltiplicazioni e verificate che questa espressione e la riga 24 danno lo stesso risultato. PASCAL non dà metodi espliciti per "l'elevamento a potenza" (per esempio, il primo termine in questa espressione, in forma algebrica, è W elevato alla quarta). Non preoccupatevi se il significato matematico di tutto questo non vi è chiaro; dovete prestare attenzione a come PASCAL tratta le espressioni algebriche e non al loro significato matematico.

La riga 27 calcola il valore della seguente espressione:

$$\frac{4(X+Y)^2}{X-Y}$$

Notate che la forma algebrica di questa espressione non richiede parentesi per racchiudere il denominatore (la parte della frazione sotto la linea, in questo caso X-Y), mentre PASCAL, per ottenere lo stesso risultato, le richiede. Eseguite questa o-

perazione con carta e penna per essere sicuri che ottenete lo stesso risultato visualizzato dall'elaboratore. Quale sarebbe stato il risultato se il denominatore in riga 27 non fosse stato racchiuso tra parentesi?

Le due espressioni in riga 30 e 33 mostrano un modo corretto ed uno non corretto di scrivere le seguenti espressioni algebriche in PASCAL:

$$-\frac{(1+\frac{Z^2}{X^2})}{(1-\frac{Z^2}{X^2})}+Y$$

Come potete facilmente verificare la prima di queste due forme non è corretta. Prima di andare avanti con la lettura, vedete di capire cosa c'è che non va nella prima forma confrontandola con la seconda.

Nella riga 30, l'espressione PASCAL Z*Z/X*X appare 2 volte. In entrambi i casi, ha il seguente effetto espresso in forma algebrica:

$$(\frac{Z^2}{X})\,X$$

In altre parole, Z è elevato al quadrato, quindi il risultato è diviso per X, quindi il nuovo risultato è moltiplicato per X. Il risultato finale è semplicemente il quadrato di Z, essendo X "eliminato". Nella linea 33, la moltiplicazione di X per se stesso per formare il quadrato di X è "forzata" prima della divisione avendo racchiuso X*X tra parentesi. Il calcolo del valore corretto dell'espressione, con i valori delle variabili come sono stati inizializzati in riga 18, è molto semplice da fare con carta e penna. Fate questo calcolo, verificando che ottenete lo stesso risultato dell'elaboratore (0.333333 è equivalente a 1 terzo).

Tutte e tre le espressioni nella procedura ARROTERR dovrebbero calcolare il valore della stessa espressione algebrica, che semplificata al massimo diventa:

$$2 + EPS$$

I tre esempi sono stati preparati per illustrare le imprecisioni che spesso sorgono quando si fanno operazioni con numeri REAL. Il programma è stato preparato per sperimentare diversi valori di EPS. L'identificatore "ESP" dovrebbe suggerire il carattere greco "Epsilon" che è spesso usato in matematica per rappresentare una piccola variazione di grandezza. Ovviamente, solo la terza espressione, nella riga 12 del programma, dà un risultato vicino a quello corretto (in questo caso il risultato è cor-

retto). Gli altri due metodi danno dei risultati che sono molto imprecisi considerando che l'elaboratore esegue i suoi calcoli con sei cifre di "accuratezza".

ESERCIZIO 5-3:
Introducendo diversi valori di EPS fate delle prove con il programma ALGEBRA: verificate i risultati. Notate l'effetto di piccole variazioni di EPS sull'imprecisione ed il fatto che gli errori non sono nella stessa direzione nè associati con la stessa espressione per ogni valore di EPS. Notate anche che errori molto grossi sono associati con pochi valori "patologici" di EPS. I programmatori fanno spesso l'assunzione, che dovranno presto rinnegare, che i casi patologici sono così inverosimili e che non debbano prendere precauzioni per evitare errori generati da questi casi. Come minimo, verificate il programma con i seguenti valori di EPS:

0.000025 (il risultato dovrebbe essere lo stesso del libro)
0.000024, 0.000026,
0.00003, 0.00002,
0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1
1E−7 (vale a dire 0.0000001)

L'imprecisione in queste espressioni nasce dal fatto che l'elaboratore non ha abbastanza bits in una parola di memoria per far sì che la somma (1 + EPS*EPS) sia memorizzata con sufficiente precisione. Quando, per esempio, il valore di EPS è 0.00001 la somma sarà eseguita nel modo seguente:

```
1.000000 |
0.000000 | 0001
---------+-----
1.000000 |
```

La linea verticale mostra dove parte il troncamento dovuto al numero insufficiente di bits di una parola. Il troncamento avviene quando l'elaboratore cerca di allineare i due numeri che devono essere sommati in modo che i punti decimali siano incolonnati.

Il numero di colonna, dal punto decimale stesso, non è materia di interesse. Per esempio, se EPS = 0.00003 allora EPS*EPS=9,0E−10 (cioè 0.0000000009). La ragione di questo è che gli "*zero in testa*", quella a sinistra della cifra diversa da zero più significativa, non devono essere tenuti in memoria in un numero in virgola mobile. Il valore memorizzato dopo il calcolo di EPS*EPS in questo esempio ha una mantissa equivalente a 900000, e un esponente equivalente a 10 elevato a −15. In altre parole si ha l'equivalente di:

$$\frac{900000}{10^{15}}$$

Il processo di allineare le colonne di due numeri che devono essere sommati, di modo che i punti decimali sono in accordo, si chiama "*normalizzazione*". Quando un numero REAL è memorizzato, è normalizzato in modo che non ci siano zeri in testa. Questo fa sì che il numero sia memorizzato con il massimo di bits possibili, ma non garantisce che tutti quei bits saranno usati nelle addizioni future.

I problemi di precisione come quelli descritti sono uno dei principali ostacoli nei calcoli scientifici su un elaboratore. Molta parte dell'analisi numerica è dedicata alla ricerca di strategie per minimizzare gli errori nei calcoli eseguiti da elaboratori. Queste strategie tengono conto sia degli errori di arrotondamento che di troncamento e cercano di ordinare i vari passi dell'elaborazione in modo da minimizzare questi errori.

I problemi di precisione con i numeri REAL sono molto importanti anche nelle applicazioni gestionali. Una delle strategie usate in parecchie implementazioni di COBOL ("Common Businnes Oriented Language") fa si che molte elaborazioni siano fatte con interi, anche se il programmatore pensa che i numeri siano delle frazioni decimali: COBOL permette al programmatore di determinare il numero di cifre decimali di precisione che vuole. Senza che il programmatore lo sappia, una <variabile> che debba rappresentare Dollari e Cents (2 cifre dopo il punto decimale) viene in effetti trattata come una rappresentazione intera dei Cents. A dispetto di questi metodi anche in programmi gestionali succedono errori di arrotondamento e troncamento. Un esempio abbastanza classico è quando dei dollari devono essere moltiplicati per un tasso (es. calcolo del pagamento degli interessi).

ESERCIZIO 5-4:
Scrivete e provate un programma che calcola il pagamento totale su un mutuo per un montare originale PRINCIPAL ed usando un tasso di interesse IRATE. Mettete a punto il programma in modo che possiate osservare cosa succede usando valori variabili di PRINCIPAL e IRATE i quali debbono essere variabili REAL. Il calcolo base richiesto è veramente semplice:

```
TOTALE = PRINCIPAL * (1 + IRATE)
```

Dovreste eseguire i calcoli due volte: la prima usando aritmetica REAL e la seconda intera. Dovreste quindi confrontare i valori di TOTALE. Per lavorare con numeri interi esprimete tutte le quantità in Cents. IRATE deve rimanere REAL sia nel calcolo in virgola mobile che in quello intero. Per il caso intero calcolate il pagamento degli interessi usando:

```
INTERESSE := ROUND(IRATE*PRINCIPAL*100)
```

e sommate quindi PRINCIPAL come un intero in Cents. Per poter vedere gli errori di arrotondamento battete 123.45 per PRINCIPAL e 0.11 per l'interesse. Provate ulteriori valori sia più grandi che più piccoli. Attenti alla possibilità che il vostro programma "salti in aria" quando cerca di usare un intero maggiore di 32767. Aggiungete degli IF per evitare che questo succeda.

10. Un programma campione — CONVERGE

Il programma CONVERGE, esegue un calcolo che, le persone che hanno un minimo di basi matematiche, riconosceranno essere lo sviluppo in serie di:

$$e^X$$

dove "e" è la base dei logoritmi naturali. Non è comunque neceessaria una preparazione matematica per capire quello che fa il programma. L'idea è di calcolare il valore della seguente sommatoria:

$$1+\frac{x^2}{1}+\frac{x^3}{2*1}+\frac{x^4}{3*2*1}+\frac{x}{4*3*2*1}+\cdots$$

Il simbolo "..." (chiamato "*ellissi*") significa che dei "*termini*" addizionali devono essere aggiunti a queste "*serie*" sino a che la somma risulta essere sufficientemente accurata. Ogni frazione compresa tra i simboli "+" in questa somma è chiamata "termine". Solitamente "sufficientemente accurata" significa che l'elaboratore sul quale si sta lavorando non può rappresentare il risultato con maggior accuratezza di quella realizzabile con i bits di una parola. Se N è un numero intero una forma generale per descrivere un qualsiasi termine è la seguente:

```
 1: PROGRAMMA CONVERGE;
 2: VAR R,EPS:REAL;
 3:    CONT:INTEGER;
 4:
 5: FUNCTION SERIE(X:REAL; VAR CNT:INTEGER):REAL;
 6: VAR TERM,SUM:REAL;
 7:    N:INTEGER;
 8: BEGIN
 9:    N:=1;
10:    TERM:=1;
11:    SUM:=1;
12:    REPEAT
```

```
13:       TERM:=TERM*X/N;
14:       SUM:=SUM+TERM;
15:       N:=N+1;
16:     UNTIL ABS(TERM)<=EPS*SUM
17:     SERIE:=SUM;
18:     CNT:=N;
19: END (*SERIE*);
20:
21: BEGIN (*programma principale*)
22:     EPS:=0.0001;
23:     WRITELN('CONVERGE');
24:     WRITELN('BATTI UN NUMERO REALE');
25:     READLN(R);
26:     WHILE ABS(R)<80.0 DO
27:     BEGIN
28:       WRITELN('SERIE(R)=',SERIE(R,CONT),
29:                           ',EXP(R)=',EXP(R) );
30:       WRITELN('CONT ERA:', CONT);
31:       WRITELN;
32:       WRITELN('ABBATTINE UN ALTRO');
33:       READLN(R);
34:     END;
35: END.
```

```
 1: Visualizzazione associata al programma CONVERGE
 2:
 3: CONVERGE
 4: BATTI UN NUMERO REALE
 6: 1.0
 7: SERIE(R)=2.718254, EXP(R)=2.718281
 8: CONT ERA:9
 9:
10: BATTINE UN ALTRO
11: 1E1
12: SERIE(R)=22025.43, EXP(R)=22026.46
13: CONT ERA:35
14:
15: BATTINE UN ALTRO
16: -1.0
17: SERIE(R)=3.678818E-1, EXP(R)=3.678793E-1
18: CONT ERA:9
19:
20: BATTINE UN ALTRO
21: 100.0
```

$$\frac{X^N}{N*(N-1)*(N-2)*\ldots*1}$$

ed ogni termine avrà un N incrementato di 1 rispetto al precedente. Un altro modo per rappresentare un generico termine è il seguente:

$$t_{(n)} = t_{(n-1)} * \frac{X}{n}$$

In altre parole, per formare un nuovo termine, si prende il vecchio e lo si moltiplichi per x/n. Si noti la somiglianza di questa definizione con le definizioni usate negli esempi delle procedure recursive nel capitolo 4.

Come l'elaborazione continua ogni termine successivo diviene più piccolo del precedente in quanto il denominatore (sotto la linea della frazione) va crescendo molto più in fretta del numeratore (sopra la linea). Il programma omette di ciclare quando trova un termine più piccolo di un valore considerato "stimatore di precisione". In questo programma lo stimatore è la variabile EPS che è posizionata nella linea 22. Il valore assegnato ad EPS, in questo esempio, è stato scelto abbastanza grande così che l'imprecisione del risultato può essere osservata. PASCAL ha una funzione interna per calcolare questa serie con la miglior precisione permessa dall'elaboratore. Questa funzione, chiamata EXP, è inclusa nell'istruzione WRITELN nella riga 29 per permettervi di notare la differenza nelle due elaborazioni. Le linee visualizzate dal programma CONVERGE, per 3 valori di R immessi da tastiera, sono mostrate dopo il programma con numerazione indipendente.

Il nome è stato scelto in quanto si dice che la serie "converge" verso il risultato corretto man mano che si aggiungono termini. Non è comunque possibile raggiungere il risultato con precisione assoluta in quanto questo implicherebbe un elaboratore con *infiniti* bits per ogni parola.

ESERCIZIO 5.5:
Rivedete il programma CONVERGE per fargli calcolare il valore della seguente serie:

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

dove la notazione dta per il denominatore di ogni termine va interpretata come segue:

3! = 3*2*1
5! = 5*4*3*2*1
7! = 7*6*5*4*3*2*1

e così via.
Perciò il metodo che potete usare per calcolare i termini di questa serie sarà molto simile al metodo usato in CONVERGE. Potete verificare se il risultato è corretto confrontando il vostro risultato con quello della funzione interna SIN(X). Per ottenere un valore più accurato usate un valore di EPS uguale o inferiore a 1E−6. Sollecitate il programma usando valori di R, immessi da tastiera, che vanno da −10 a +10 ed includete i seguenti:

0.0, 3.14159, 3.14159, 0.78539, −1.57079

11. Numeri casuali

In questa sezione vi presentiamo due programmi che pur essendo divertenti dovrebbero stimolare considerazioni molto serie. Entrambi i programmi usano una procedura il cui compito è quello di generare un "*numero casuale*" ogni qualvolta viene chiamata. Con "casuale" intendiamo un numero "estratto da un cappello" contenente una gran quantità di numeri appartenenti ad un certo insieme. Perciò ogni numero successivamente estratto non dovrebbe avere relazione alcuna con i numeri estratti precedentemente. In pratica non è possibile eliminare totalmente qualsiasi relazione per cui i numeri generati da un elaboratore sono spesso detti "pseudo-casuali". Sottoprogrammi per la generazione di numeri casuali sono di uso molto comune nella simulazione di sistemi fisici di qualsiasi tipo, nel cercare possibili (intenzionali) errori in sistemi di contabilizzazione ed in parecchi altri campi.

In entrambi i programmi la procedura CASUALI lavora sfruttando il fatto che dei bits di informazione vengono persi quando un numero è troncato per poterlo scrivere in una parola di memoria. In questo caso la funzione interna TRUNC è usata per ottenere solo la parte frazionale della variabile REAL SEME (linea 9). Questo è ottenuto sottraendo la parte INTEGER del valore di SEME ottenuto in riga 8. Il risultato della riga 9 è una frazione i cui valori sono un numero qualunque compreso tra 0 e 1.000000. La riga 10 produce quindi un numero quasi casuale che va da 0 a 1 meno della costante per cui è moltiplicato SEME. Nel programma CASUALGIRO, il prodotto ottenuto in questo modo è ridotto di 24.9 per far sì che il valor *medio* di tutti gli interi prodotti sia molto vicino allo zero.

Il programma CASUALGIRO genera il disegno mostrato in figura 5-3. In ogni passata del ciclo che va da linea 25 a linea 30, le posizioni X e Y della tartaruga sono variate con il valore generato dalle due chiamate di CASUALI. Il risultato è uno squinternato disegno che potrebbe essere fatto da una mosca ubriaca che gironzola su un pezzo di carta.

La figura 5-4 è il risultato di CASUALGIRO leggermente modificato cambiando i valori delle costanti usate in riga 8 e 14. In questo caso il valore usato in linea 14 era 7.5295141; i valori usati in linea 8 erano rispettivamente 3.1415917 e 2.7182813. L'intento nell'usare questi numeri complicati è quello di creare una complessa configurazione di bits che portino, verosimilmente, a perdere dei bits tutte le volte che viene calcolato un nuovo valore di SEME. Come potete vedere, il disegno ottenuto con le costanti usate per la figura 5-4 ha un'evidente tendenza a ripetere gruppi di numeri dopo breve tempo. Potete usare una funzione essenzialmente uguale a CASUALI su quasi tutti gli elaboratori, ma i valori delle costanti dovranno essere cambiati per assicurare una sequenza quasi casuale. I metodi per verificare il grado di casualità sono complicati e non rientrano negli scopi di questo libro.

```
 1: PROGRAMMA CASUALGIRO;
 2: CONST XMAX=480; YMAX=350;
 3: VAR SEME:REAL;
 4:    X,Y,ANGOLO:INTEGER;
 5:
 6: FUNCTION CASUALI:INTEGER
 7: BEGIN
 8:    SEME:=SEME*27.182813+31.415917;
 9:    SEME:=SEME-TRUNC(SEME);
10:    CASUALI:=TRUNC(SEME*50-24,9);
11: END;
12:
13: BEGIN (*PROGRAMMA PRINCIPALE*)
14:    SEME:=1.23456789; (*COMPLICATA CONFIGURAZIONE DI BITS*)
15:    PENCOLOR(WHITE);
16:    MOVETO(-XMAX,0);
17:    MOVETO(XMAX,0); (*ASSE X*)
18:    PENCOLOR(NONE);
19:    MOVETO(0,YMAX);
20:    PENCOLOR(WHITE);
21:    MOVETO(0,-YMAX); (*ASSE Y*)
22:    MOVETO(0,0);
23:    X:=0;
24:    Y:=0;
25:    WHILE (ABS(X)<XMAX) AND (ABS(Y)<YMAX) DO
26:       BEGIN
27:          MOVETO(X,Y);
28:          X:=X+CASUALI;
29:          Y:=Y+CASUALI;
30:       END;
31: END.
```

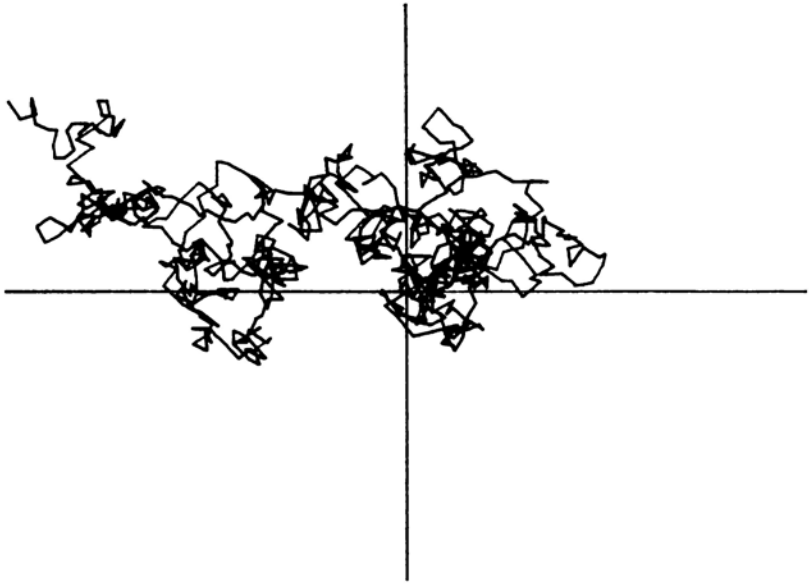


Figura 5-3

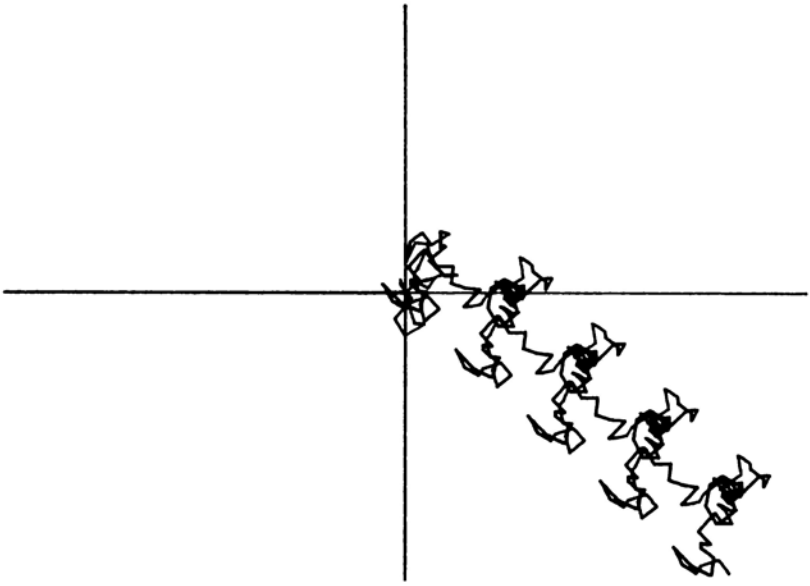


Figura 5-4

Il programma QUATTRO LET visualizza una sequenza (quasi) casuale di parole di 4 lettere. Quando lanciamo il programma sul nostro elaboratore, compaiono le seguenti linee:

```
ZXJF
KSYN
RXUF
PWTE
CKSY
ZEYF
TNQG
QIYX
GPQP
```

Da questo piccolo campionario, si può constatare che le parole familiari di 4 lettere (non tutte pronunciabili davanti alla mamma!) non possono essere generate casualmente come si potrebbe semplicisticamente pensare.

Se cambiate le costanti nella funzione CASUALI, ne risulterà una diversa sequenza di "parole". Quante parole sarebbero generate, secondo voi, prima che compaia una parola italiana riconoscibile? Se non ne avete idea,... forza con il programma!

```
 1: PROGRAMMA QUATTROLET;
 2: VAR SEME:REAL;
 3:    I,J:INTEGER;
 4:    CH:CHAR;
 5:
 6: FUNCTION CASUALI:INTEGER;
 7: BEGIN
 8:    SEME:=SEME*27.182813+31.415917;
 9:    SEME:=SEME-TRUNC(SEME);
10:    CASUALI:=TRUNC(SEME*26);
11: END;
12:
13: BEGIN (*PROGRAMMA PRINCIPALE*)
14:    SEME:=1.23456789; (*CONFIGURAZIONE DI BITS COMPLICATA*)
15:    WRITELN(QUATTROLET');
16:    REPEAT
17:       FOR I:=1 TO 20 DO
18:       BEGIN
19:          FOR J:=1 TO 4 DO
20:             WRITE(CHR(ORD('A')+CASUALI));
21:          WRITELN;
```

```
22:       END;
23:       WRITELN('PIGIA <ESC> PER FERMARE,',
24:                          '<SPAZIO> PER CONTINUARE');
25:       READ(CH);
26:     UNTIL CH=CHR(27(*ESC*));
27: END.
```

ESERCIZIO 5.6:
Scrivete e provate un programma che genera dei "titoli" casuali scegliendoli a caso e visualizzando una parola per ognuno dei tre elenchi seguenti.

PORTOGALLO	GOVERNO	CROLLA
GIAPPONE	VULCANO	ERUTTA
KISSINGER	MISSIONE	FALLISCE
ROSSI	POLITICA	AVANZA
LOMBARDIA	ECONOMIA	MIGLIORA
STUDENTE	CAPO	PROMOSSO
NAPOLI	AEREOPORTO	CHIUSO
ITALIANA	ARMA	FALLISCE
SEGRETO	EVIDENZA	SCOMPARE
CASALPUSTERLENGO	ATTACCO	SVELATO
ARABO	BLOCCO	NEGOZIATO

Il modo più semplice per far questo è di usare tre istruzioni CASE, ognuna delle quali assegna, ad una corrispondenza variabile STRING, una parola scelta da una dell'elenco. Se chiamate queste variabili S1, S2 ed S3 allora la seguente istruzione alla fine del ciclo si visualizzerà il titolo scelto:

```
WRITELN(S1, ' ', S2, ' ', S3)
```

La selezione da un elenco dovrebbe richiedere una sola chiamata a CASUALI. Dovrete modificare CASUALI per generare valori da 0 a 10 (o da 1 a 11) corrispondenti agli undici elementi di ogni elenco.

Problemi

PROBLEMA 5.1:
Convertite ognuna delle seguenti espressioni, date in forma algebrica, in <espressio-

ni aritmetiche > accettabili da PASCAL (NOTA: in questi esempi, ogni identificatore contiene solo una lettera):

$$\frac{L+\frac{R}{C}}{L-\frac{R}{C}}$$

$$AX^2+BX+C$$

$$1+\frac{AX+(B-A)X^2}{(1-X)^2}$$

$$\frac{P+Q}{XN+A^2}$$

$$\frac{4(X+Y)^2}{2X-Y}$$

PROBLEMA 5.2:
Scrivete i valori di I o R dopo l'esecuzione di ognuna delle seguenti istruzioni se l'istruzione stessa è "corretta" per PASCAL. Altrimenti segnalarne la non correttezza.

```
I := 10 DIV 2;
I := 11 DIV 2;
I := 25 MOD 10;
R := 1/(100000*100000);
I := 25 MOD 5;
I := 10 * 0.1;
R := TRUNC(10/3);
I := TRUNC(3.66667);
I := TRUNC(5/4);
R := 100 + 3*15;
R := 14/3 - ROUND(14/3);
R := 14/3 - TRUNC(14/3);
```

PROBLEMA 5.3:
Convertite i seguenti numeri binari in forma decimale:

1010, 10011, 110110, 110111, 111000, 101010

Convertite i seguenti numeri decimali in forma binaria:

16, 15, 17, 35, 31, 128, 255, 510

CAPITOLO 6

GESTIONE DI PROGRAMMI CON STRUTTURE COMPLESSE

1. Obiettivi

In questo capitolo iniziamo a chiedervi di usare un approccio ordinato per risolvere problemi con l'elaboratore. Questo è il primo capitolo in cui dovrete sintetizzare alcune specifiche del programma basate su una descrizione generale del problema.

1a. Apprendimento dei ruoli distinti di un algoritmo e dei dati associati che si combinano a formare un programma.

1b. Apprendimento dell'uso di Diagrammi di Struttura per descrivere algoritmi, e la loro stretta correlazione alle azioni di un programma.

1c. Apprendimento a suddividere la soluzione di un problema in diverse parti distinte che verranno prese in considerazione in sequenza.
Queste includono:
- descrizione concettuale del problema sulla carta
- descrizione sommaria di algoritmi di soluzione
- definizione della rappresentazione dei dati
- disegno dettagliato dell'algoritmo a stadi successivi
- stesura del programma
- introduzione del programma nell'elaboratore e compilazione
- messa a punto del programma

1d. Studio della(e) soluzione(i) di uno o più problemi concettualmente semplici che richiedono programmi di difficoltà media.

2. Premessa

Quando i programmi diventano più grossi e devono espletare compiti più comples-

si essi diventano anche più esposti a errori logici. Uno dei nostri compiti primari in questo capitolo sarà l'introduzione di un metodo per stendere diagrammi di Programmi Strutturati che aiutino a visualizzare le interferenze delle varie parti del programma. Altro compito sarà quello di farvi suddividere la soluzione di un problema con l'uso dell'elaboratore in quella di diversi problemi relativamente distinti. Dovreste iniziare a descrivere ciò che fate nel risolvere un problema come se fosse rappresentato da un diagramma dello stesso tipo di quelli che introdurremo in questo capitolo.

Nonostante la vostra familiarità con schemi a blocchi come mezzo per disegnare diagrammi che rappresentano programmi logici, faremo un uso molto scarso di tali schemi in questo libro. Non appena un programma diventa più grosso e complesso gli schemi a blocchi che lo descrivono possono diventare grossi e complessi allo stesso modo; anche schemi a blocchi relativamente semplici possono diventare così difficili da comprendere che l'usarli diventa una perdita di tempo. La figura 6-1 mostra un tale schema, senza la descrizione della logica specifica: anche per programmi semplici come quello descritto in tale schema è quasi impossibile verificare se il programma funziona correttamente in tutte le possibili situazioni. Per programmi più grossi lo schema a blocchi assomiglia ad una ciotola piena di spaghetti ed è comprensibile quanto una descrizione logica.

Come migliore alternativa introduciamo in questo capitolo il Diagramma di Struttura. Il Diagramma di Struttura aiuta un programmatore a visualizzare la struttura *"ad albero"* di un programma ben costruito. Poichè vi è un solo cammino per cui un programma può arrivare ad ogni elemento del diagramma, sono minimizzati gli errori che derivano da comunicazioni sovrapposte con altre parti del programma. Quando apprezzerete i pregi di una struttura ad albero per descrivere un programma, vi accorgerete che i programmi in PASCAL da soli sono sufficienti per visualizzare la struttura di programma.

3. Cos'è un Algoritmo?

Secondo il Dizionario di Webster, un *"algoritmo"* è uno speciale metodo per risolvere un certo tipo di problema. L'esempio più familiare di algoritmo nella vita di tutti i giorni è fornito dalle ricette di cucina, almeno nel senso che gli algoritmi sono scritti su carta. In realtà noi usiamo un algoritmo prestabilito per compiere quasi tutti i compiti noti sebbene raramente ci si fermi a pensare ai vari passi componenti tali algoritmi. Perché sia utile nel nostro contesto, dovremo approfondire un poco la definizione di algoritmo.

Un manuale recente definisce un "Algoritmo" come "un elenco di istruzioni per portare a termine un processo passo dopo passo". Tale definizione evita di specifica-

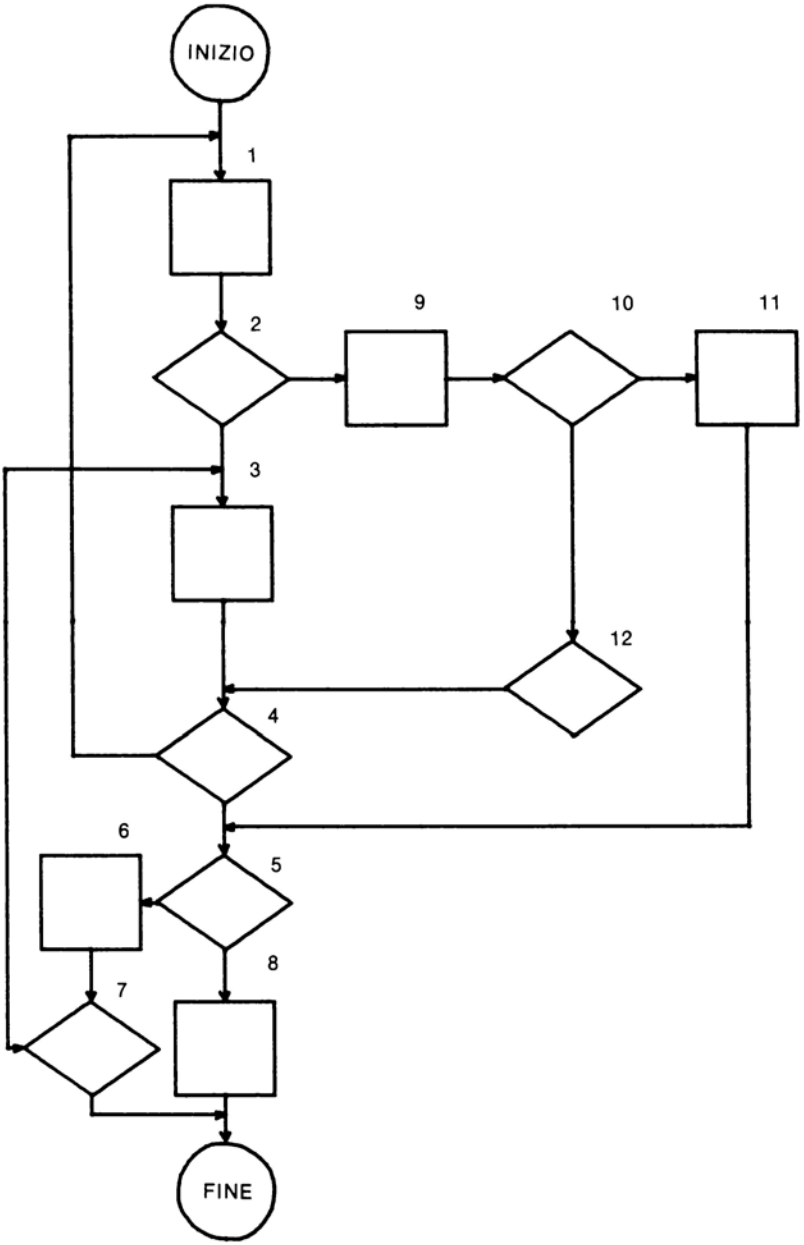


Figura 6-1

re il livello di dettaglio che deve essere usato nell'elenco. Infatti sarà utile iniziare con un elenco di istruzioni molto poco dettagliate e informali, e rifinire poi tali istruzioni costruendo una versione finale dettagliata dell'algoritmo sotto forma di programma. Di seguito diamo un algoritmo che potreste usare per uscire da un'autostrada.

1. Rallentamento al segnale di uscita della città in cui dovete recarvi
2. Imbocco corsia di decelerazione
3. Scelta porta d'uscita
4. Fermata
5. Consegna tagliando
6. Attesa cifra da pagare
7. Consegna denaro
8. Ripartenza

Si noti che questo algoritmo specifica sia le azioni, sia l''ordine in cui esse devono essere fatte.

Avete già visto che la natura dell'elaboratore digitale porta a specificare le soluzioni dei problemi usando algoritmi, cioè usando notazioni "algoritmiche". Poichè l'elaboratore può fare soltanto un piccolo passo per volta, si è stati portati a concentrarsi più sull'ordine di tali passi che sul capire come ogni passo è collegato in un tutt'uno con il problema. Oggi sappiamo che è spesso meglio invertire tale procedimento, decidendo l'ordine del procedimento dopo aver definito i passi maggiori del problema.

4. Livello di dettaglio

Quando si esprime un algoritmo come quello visto prima, ogni passo implica un qualche livello di aggregazione o astrazione. Ad esempio il passo #3 potrebbe essere ampliato come segue:

3. Scelta porta uscita
   - 3.1 Individuazione porte con segnale verde
   - 3.2 Contare macchine in coda per ogni porta
   - 3.3 Scelta della porta con il minor numero di macchine
   - 3.4 Verificare che la manovra sia possibile
   - 3.5 Portarsi in direzione della porta
   - 3.6 Accodarsi a pochi centimetri dall'ultima macchina
   - 3.7 Avvicinarsi lentamente alla porta.

Si può facilmente vedere come parecchi di questi passi di secondo livello possano essere ulteriormente approfonditi.

Per vedere come un diagramma di struttura si inserisca in questo schema, iniziamo con un problema più semplice. Di seguito diamo una descrizione di quello che dovrebbe essere il "percorso verde" per andare da Milano a Basilea (se preferite sostituite posti più vicini a voi). Dovreste attraversare le seguenti località:

Milano - Como - Lugano - Bellinzona - Airolo - Audermatt - Susten - Interlaken - Thun - Bern - Solothurn - Olten - Basilea

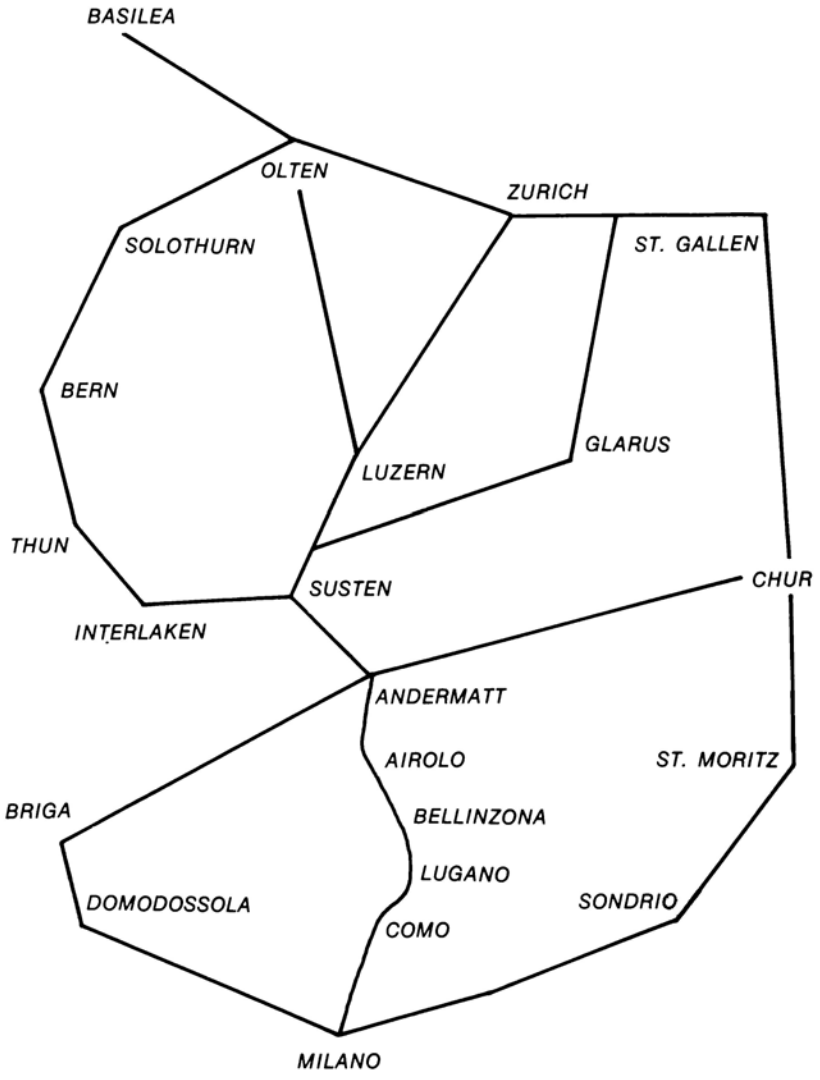


Figura 6-2

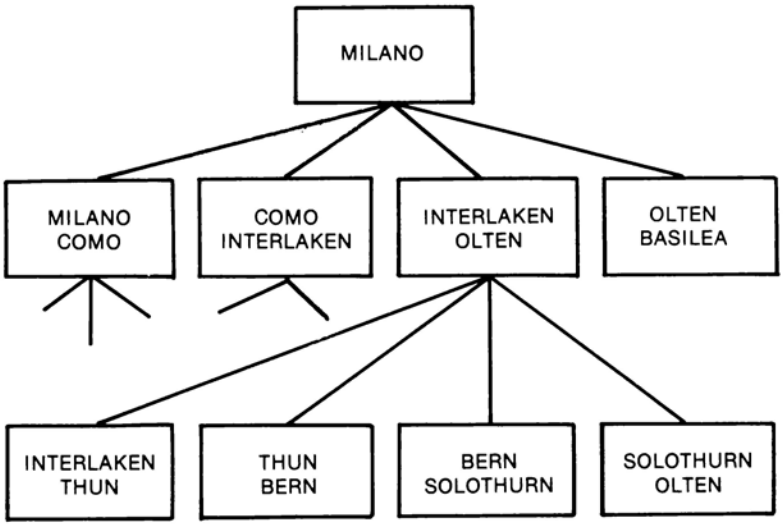


Figura 6-3

Coprite ora la pagina e vedete per quanto tempo ricorderete quei luoghi! Supponiamo ora di ridurre la descrizione, lasciando solo i più importanti punti della strada:

Milano - Como - Interlaken - Olten - Basilea

Potrete forse ricordare questo elenco senza troppi sforzi, anche se non sapete con precisione dove sono le posizioni di tutte quelle località. La figura 6-2 rappresenta una versione semplificata della carta stradale. La figura 6-3 mostra come possiamo considerare l'elenco sopra riportato come un secondo livello di dettaglio in relazione al problema originale di seguire la strada:

Milano - Basilea

Ciascun blocco di secondo livello può essere ulteriormente spezzato in livelli di maggior dettaglio, per esempio:

Interlaken - Olten

diviene:

Interlaken - Thun - Bern - Solothurn - Olten

e così via. Facendo veramente il viaggio, non avreste certo delle difficoltà nel ricor-

dare questo breve elenco di punti da toccare dopo aver raggiunto Interlaken. Potreste ancora controllare la vostra cartina, o anche chiedere alla polizia dopo aver raggiunto Olten prima di iniziare l'ultimo pezzo.

Alla base di tutto sta il fatto che la nostra mente non riesce a ritenere nello stesso tempo, e con la stessa chiarezza tutti i dettagli di un dato problema. Tuttavia noi riusciamo ad aggregare dai 5 ai 7 elementi per formare un'unica concettualizzazione ad un più alto livello di astrazione. Non abbiamo perciò difficoltà a trattare il concetto di un viaggio tra Milano e Basilea, ben sapendo che ci sono parecchi dettagli da considerare nell'intraprendere il viaggio ma non dovendovi far fronte quando si pensa allo stesso come un tutto uno. Abbiamo introdotto questo concetto quando, nel capitolo 2, abbiamo parlato della necessità delle procedure.

5. Diagrammi di struttura

Recepite le idee espresse nelle sezioni precedenti, possiamo ri-esprimere il nostro algoritmo di Fermata autostradale. Riduciamo dapprima il numero delle voci di dettaglio allo stesso livello:

1: Raggiungimento porta d'uscita
  1.1 Rallentamento e posizionamento sulla corsia di destra in vista dell'indicazione della "vostra" città.
  1.2 Imbocco corsia di decelerazione
  1.3 Scelta porta d'uscita
    1.3.1 Individuazione porte con segnale verde
    1.3.2 Conta delle macchine in coda per ogni porta
    1.3.3 Scelta della porta con il minor numero di macchine
    1.3.4 Verifica della fattibilità delle manovre
    1.3.5 Portarsi in direzione della porta
    1.3.6 Accodarsi a pochi centimetri dall'ultima macchina
    1.3.7 Avvicinarsi lentamente alla porta
  1.4 Fermata
2. Pagamento
  2.1 Consegna tagliando
  2.2 Attesa cifra da pagare
  2.3 Consegna danaro
3. Attesa resto
4. Ripartenza

La figura 6-4 mostra le stesse informazioni con un diagramma di struttura. (Sono debitore a Bob Doran e Graham Tate, della università di Masey, per un articolo di ricerca del giugno 72, "Un approccio alla programmazione strutturata", in cui descrivono l'idea dei diagrammi di struttura). Ci sono parecchie cose da notare:

a) Il diagramma è una struttura ramificata, simile ad un "albero" genealogico. In informatica tali strutture sono conosciute come alberi.

b) L'albero ha un'unica *"radice"*, ha rami che possono suddividersi in altri rami; alla fine dei rami ci sono le *"foglie"*. Sia i punti di diramazione che le foglie sono spesso chiamati *"nodi"*. Le foglie sono nodi "terminali" in quanto rappresentano la fine di un percorso che partendo dalla radice segue un particolare sistema di rami.

c) Come la maggior parte degli alberi che si vedono su pubblicazioni di informatica, la radice del nostro albero è in cima al diagramma. Questo non è un punto molto importante, ma all'inizio può confondere le idee.

d) C'è un solo percorso diretto da qualunque foglia alla radice, o viceversa. Non ci sono scorciatoie per andare direttamente da una foglia all'altra.

e) Le foglie possono essere trovate a diversi livelli.

f) L'ordine in cui l'algoritmo compie le azioni è generalmente da sinistra a destra, passando attraverso le foglie.

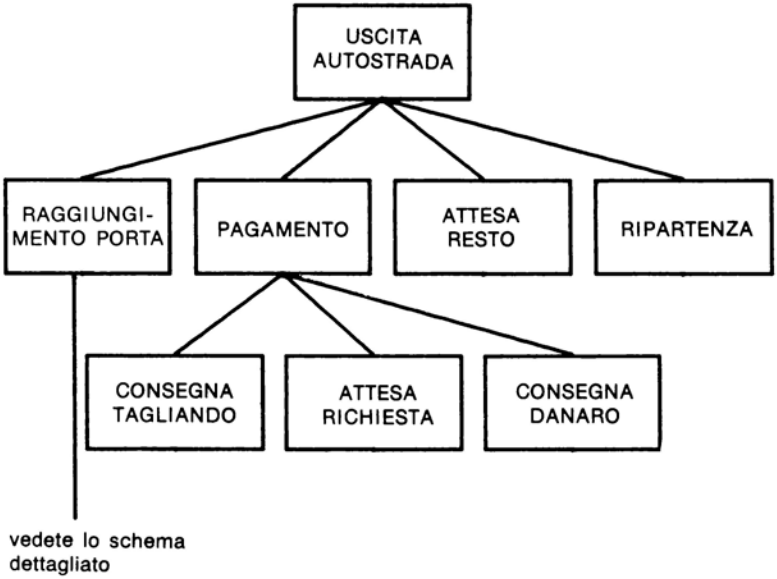
Avrete modo di leggere molto di più sulle strutture ad albero nel resto del libro. Gli alberi sono anche chiamati strutture *"gerarchiche"*.

Sin qui il nostro diagramma, e l'algoritmo che rappresenta, contiene solo nodi *azione*. Per esprimere l'ampia varietà di elaborazioni possibili su un elaboratore, è anche necessario esprimere quanto segue:

a) *Scelta* tra due o più alternative allo stesso nodo.

b) *Ripetizione* di una parte dell'algoritmo, solitamente cambiando in ogni ripetizione una o più quantità in relazione con l'elaborazione più ampia.

Come esempio di *scelta*, considerate di dover aggiungere, al punto 1.3.7. del nostro algoritmo "autostradale", una verifica sul fatto che una coda si svuoti più in fretta. Questo può essere rappresentato nel diagramma di struttura come in figura 6-5, o può essere rappresentato in forma lineare come:


USCITA AUTOSTRADA
RAGGIUNGI-MENTO PORTA
PAGAMENTO
ATTESA RESTO
RIPARTENZA
CONSEGNA TAGLIANDO
ATTESA RICHIESTA
CONSEGNA DANARO
vedete lo schema dettagliato

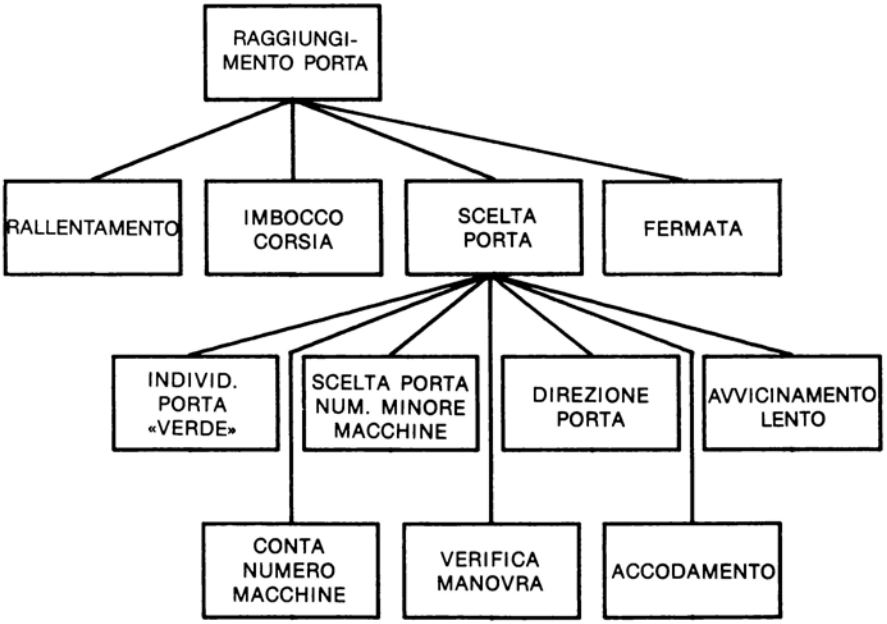

RAGGIUNGI-MENTO PORTA
RALLENTAMENTO
IMBOCCO CORSIA
SCELTA PORTA
FERMATA
INDIVID. PORTA «VERDE»
SCELTA PORTA NUM. MINORE MACCHINE
DIREZIONE PORTA
AVVICINAMENTO LENTO
CONTA NUMERO MACCHINE
VERIFICA MANOVRA
ACCODAMENTO

Figura 6-4

1.3.7 C'è una coda che si svuota più in fretta?
1.3.7.1 (sì) Va su quella coda
1.3.7.2 (no) Continua sulla tua

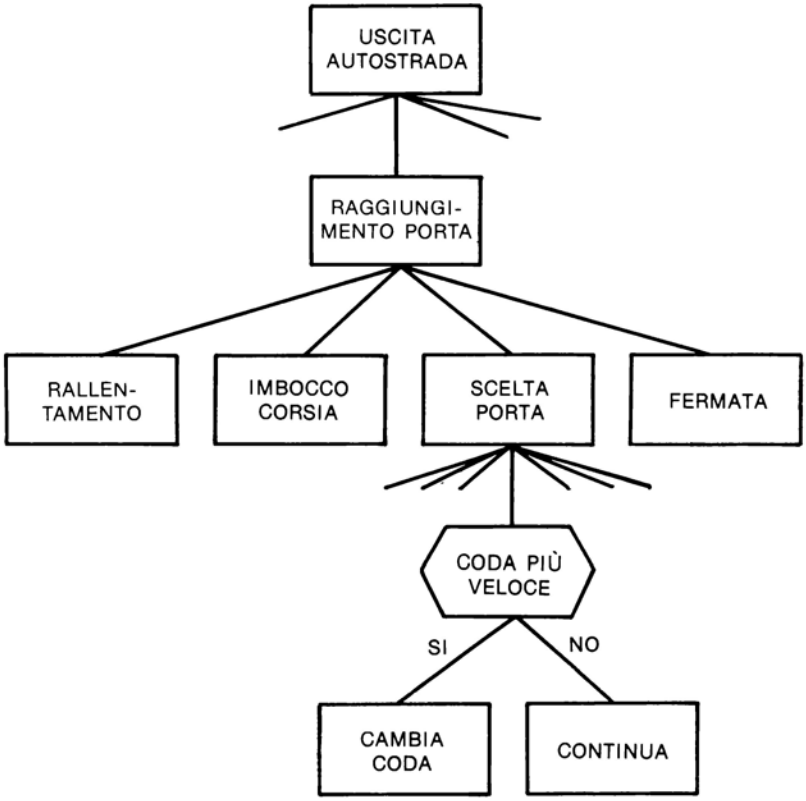


Figura 6-5

Come si vede dalla figura i blocchi che indicano una *scelta* sono rappresentati con blocchi romboidali.

Per vedere l'effetto di un blocco con scelta multipla, equivalente all'istruzione CASE, raffiniamo ulteriormente il nostro algoritmo al punto 2.3. Il punto che vogliamo ampliare è quello in cui, in base alla cifra richiesta, consegnamo danaro di vario ta-

glio. Mentre in figura 6-6 è rappresentato il cambiamento nel diagramma, qui di seguito riportiamo la notazione lineare:

2.3 Danaro Richiesto?
2.3.1 (<1000) Consegna 1000
2.3.2 (>1000 e <2000) Consegna 1000 più moneta
2.3.3 (>10000) Consegna 10000 + biglietti di taglio più piccolo
. . .
e così via

Ora per rendere l'idea della *ripetizione* nel diagramma di struttura, raffiniamo il punto 1.3.7.1. Supponiamo di aver davanti a noi un certo numero di macchine e di non volerci fermare lasciando avanzare la coda fino allo svuotamento (pericoloso se si hanno dietro autisti leggermente nervosi). Continueremo quindi ad avanzare di 3/4 metri e poi fermarci. Questa modifica è mostrata nel diagramma di figura 6-7 e nella tabella di struttura che segue:

1.3.7.1 Ripeti per il numero di macchine in coda
1.3.7.1. Avanza di pochi metri
1.3.7.2 Fermati

La struttura ripetuta può essere complicata quanto si vuole. Per rappresentare una ripetizione in un diagramma di struttura useremo un simbolo ovoidale. La ripetizione può continuare un numero fisso di volte come in questo esempio o potrebbe continuare fino a che una certa asserzione rimane TRUE (o FALSE). Per esempio:

1.3.7.1 Ripeti UNTIL sei davanti allo sportello
1.3.7.1.1 Avanza di pochi metri
1.3.7.1.2 Fermati

Non preoccupatevi troppo, per ora, del fatto di rispettare perfettamente il formalismo dei diagrammi di struttura. L'importante è capire come i diagrammi di struttura sono costituiti scomponendo il problema in passi separati, e come questi passi sono tra loro collegati. Generalmente un blocco di ripetizione dovrebbe contenere uno dei due punti seguenti:

a) Il numero di volte che l'iterazione deve essere eseguita, come l'istruzione FOR in PASCAL

b) L'espressione logica che determina per quanto va eseguita la ripetizione come nei costrutti PASCAL: REPEAT ... *UNTIL* e *WHILE* ... DO.

6. Sviluppo progressivo di Algoritmi

Lo sviluppo di algoritmi da usarsi su elaboratori e dei programmi che li "materializzano", è quasi sempre un processo di raffinare progressivamente fino a che è raggiunta una soluzione soddisfacente. Solo su problemi veramente semplici è probabile che voi riusciate a scrivere un algoritmo, o il programma equivalente, direttamente nella forma finale. I vari esempi dell'algoritmo "autostradale" mostra il processo di raffinamento progressivo. In questa sezione daremo alcune indicazioni delle tattiche che si sono rivelate più produttive nello sviluppo di programmi corretti.

Il problema della correttezza dei programmi è uno di quelli a cui gli esperti di elaboratori prestano più attenzione. La maggior parte dei grossi programmi attualmente in circolazione contengono almeno qualche errore logico. Forse anche voi siete stati vittima di tali errori se un grande magazzino, un concessionario automobilistico o qualche compagnia di carte di credito vi ha inviato una fattura sbagliata. Solitamente, se il programma è usato più di una volta, i responsabili del progetto passano più tempo a cercare e correggere gli errori di quanto ne hanno speso nel progetto stesso e nell'implementazione.

Molti programmi in uso sono stati scritti senza prestare attenzione ai concetti di struttura discussi in questo capitolo. Schematizzati sembrano più una pentola di spaghetti che un albero. Gli studiosi di elaboratori stanno cercando di trovare dei mezzi per usare l'elaboratore con lo scopo di provare la correttezza di algoritmi e programmi. Anche se costa un po' più di sforzo progettare algoritmi e programmi usando i concetti di struttura ad albero, è ormai chiaramente assodato che l'uso di questi concetti riduce di gran lunga lo sforzo necessario per sviluppare e mantenere programmi che non siano di dimensioni più che piccole.

Il metodo raccomandato (si veda "Notes on Structured Programming" di E.W. Diykstra nel libro "Structured Programming", Academic Press, 1972) è di iniziare con un diagramma di struttura a 2 livelli rappresentante solo una descrizione grossolana del problema che deve essere risolto. Il blocco di primo livello rappresenta la radice e descrive tutto il problema. Ciascuno dei blocchi di secondo livello dovrebbe rappresentare qualche sotto-sezione del problema: questi blocchi dovrebbero essere di complessità uniforme. Si dovrebbe rimandare la decisione sulle cose che devono fare i blocchi di terzo livello, o superiore, il più a lungo possibile. Il fatto è di spezzare il problema globale in 5/10 blocchi di secondo livello in modo da non avere, in una descrizione, più dettagli di quanti una persona riesca a tenere chiaramente sotto controllo. Ciascuno dei blocchi deve essere il più possibile indipendente dagli altri, per le stesse ragioni per cui ne abbiamo parlato a proposito delle intercomunicazioni tra procedure.

Continuando nel processo, si scompone quindi separatamente ogni blocco di se-

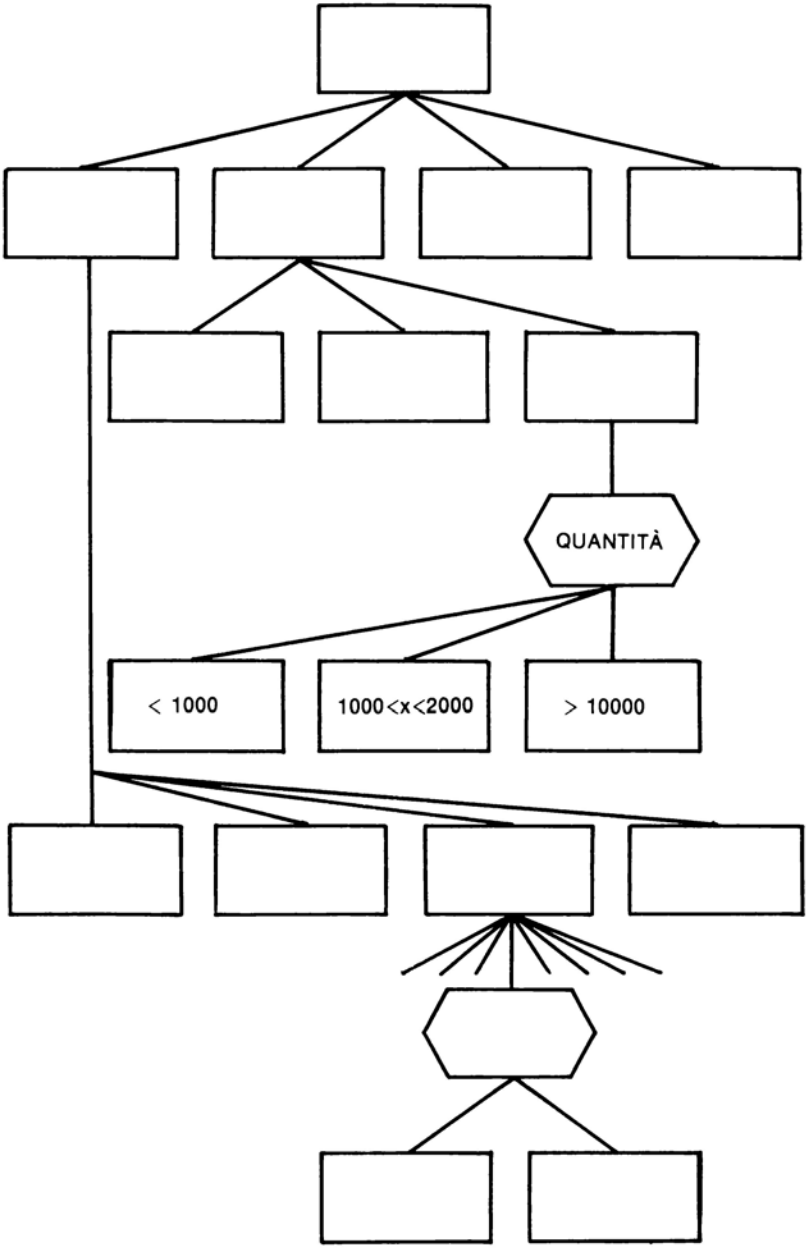


Figura 6-6

condo livello dell'algoritmo. Se ogni blocco di secondo livello viene scomposto in 5/10 blocchi di terzo livello, raggiungerete in fretta il punto in cui l'intero schema non può essere rappresentato in una pagina. Il modo migliore per affrontare questa situazione è quello di rappresentare su una pagina solo pochi livelli. Altre pagine possono essere usate per rappresentare la espansione dell'albero connesso ai nodi di più alto livello. Se non avete più spazio per disegnare blocchi addizionali in una pagina, è possibile sprecare una gran quantità di tempo stipando blocchi sempre più piccoli negli angoli. Quando vi accorgete di essere arrivati a questo punto, risparmierete tempo ridisegnando i diagrammi in un'altra pagina ed usando le pagine seguenti per i "sotto-algoritmi".

Finalmente raggiungerete un punto in cui il livello di dettaglio del diagramma di struttura è sufficiente per una descrizione concettuale dell'algoritmo, anche se i dettagli implementativi sono ancora impliciti. I programmi sono composti di sequenze di passi di elaborazione veramente piccoli. In generale non sarà necessario o desiderabile scomporre il vostro algoritmo al punto tale che ogni istruzione sia contenuta nel proprio blocco nel diagramma. Presenteremo un certo numero di esercizi più avanti nel capitolo, e nel libro, nei quali vi chiederemo di convertire diagrammi di struttura in programmi (e viceversa) con il livello di dettaglio delle istruzioni. La ragione principale di questo sta nel fatto che dovrete impratichirvi nel collegare la struttura di un programma al diagramma di struttura, e non nel fatto che pensiamo che continuerete a tracciare schemi allo stesso livello di dettaglio nei vostri sforzi futuri di risolvere problemi.

Avendo sviluppato il vostro algoritmo ad un ragionevole livello di dettaglio (non certo a trattare la trasmissione sinaptica, nel nostro algoritmo "autostradale"!), vi accorgerete di aver raggiunto un punto morto, vale a dire un punto in cui è logicamente impossibile continuare ad aggiungere dettagli seguendo la struttura dell'algoritmo già schematizzato. Certe volte, a questo punto, sarete tentati di violare le regole strutturali della rappresentazione ad albero per "fare il punto" velocemente. A lungo termine questo sarà quasi sempre un errore. Il metodo raccomandato è di ritornare ad un livello di struttura più vicino alla radice, rianalizzando il problema e avendo ben chiara la causa del vostro ritorno all'indietro. Quando abbiamo leggermente cambiato il problema dell'autostrada, ci siamo impegnati a tornare indietro ed a ridefinire il problema.

Se avete imparato a parlare una lingua straniera, avrete probabilmente già provato a fare questo processo di ritorno e ridefinizione in un simile contesto. La maggior parte dei linguaggi naturali seguono delle regole che permettono di schematizzare le frasi come alberi. Se parlate con qualcuno in una lingua che non è la vostra, vi capiterà, spesso di essere in una situazione in cui non riuscite ad esprimere un concetto che facilmente assocereste ad una parola della vostra lingua nativa. Per uscire da questo stallo voi, mentalmente, ritornate indietro e cercate un altro modo per esprimere lo

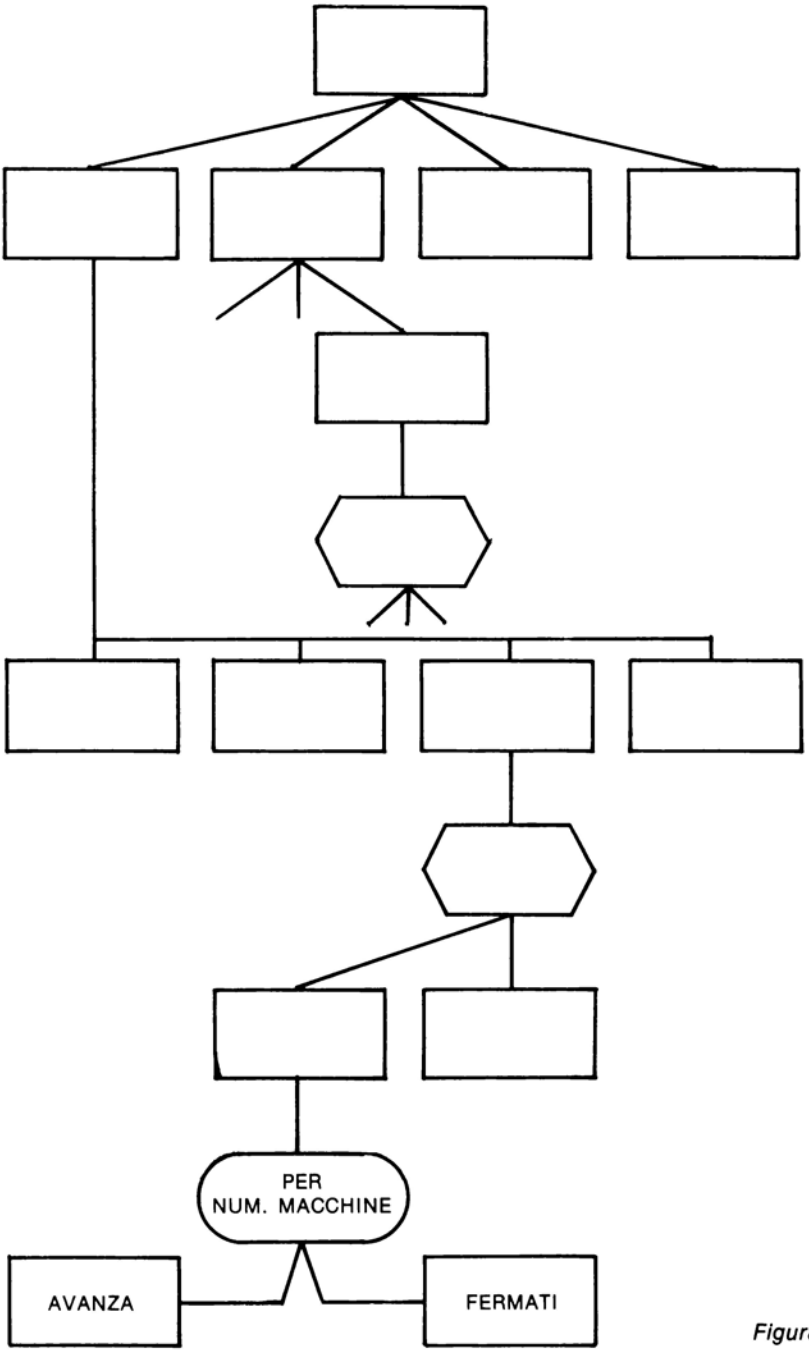


Figura 6-7

stesso concetto. Vedrete che lo stesso metodo è spesso di grande aiuto per evitare scomodi problemi nel progetto di algoritmi e programmi.

Nel gergo dell'informatica, il processo che consiste nel partire dalla radice dell'albero e nella progressiva definizione di livelli di maggior dettaglio, è conosciuto come programmazione con metodologia *"top-down* (dall'alto al basso). La ragione di questo termine dovrebbe ora esservi chiara. Il processo di ritorno indietro e di ridefinizione finchè non si giunge ad un progetto soddisfacente, è conosciuto come *"raffinamenti successivi"* di un algoritmo.

7. Diagrammi di struttura di qualche programma tipo

In questa sezione presentiamo dei diagrammi di struttura che descrivono alcuni dei programmi PASCAL tipo che sono stati presentati nella prima parte del libro. Dovreste far riferimento ai programmi PASCAL descritti da ogni diagramma e cercare di capire in dettaglio come il diagramma di struttura è collegato al programma. Riportiamo alcune considerazioni da tener presente:

a) In figura 6-8 abbiamo aggiunto delle linee tratteggiate per mostrare come possiate passare dalla struttura ad albero del diagramma all'ordine sequenziale dell'elaborazione di un programma PASCAL. La regola consiste nel partire dalla radice dell'albero, ed andare quindi sul nodo più a sinistra nel livello inferiore. Se questo nodo ha rami, considerate il nodo in cui siete come una nuova radice, e ripetete il processo da lì. Al raggiungimento di un nodo che non ha rami, il nodo va eseguito, e quindi l'elaborazione scatta al prossimo nodo a destra allo stesso livello. Se il nodo ha rami, consideratelo una radice e ripetete le operazioni; e così via... Come potete vedere, l'idea di ripetere il processo logico che si espleta al raggiungimento della radice, per ogni nodo che ha dei rami, è di per sè recursivo.

b) Nella maggior parte degli algoritmi è necessario inizializzare i valori delle varie <variabili> che sono usate. Spesso ci sono talmente tante inizializzazioni che da sole occupano un'intera pagina di diagrammi di struttura. La soluzione che abbiamo usato noi è illustrata nella figura 6-9. Nel rettangolo #2, sono inizializzate tre variabili semplici; questo non crea confusione, per quanto riguarda la struttura dello schema, poichè questi passi singoli sono compiuti in modo strettamente sequenziale. In effetti sarebbe possibile riorganizzare i passi aggregati in un unico blocco senza alterare la logica dell'algoritmo.

c) È spesso utile etichettare i blocchi in un diagramma di struttura in modo da convogliare la struttura ad albero dentro le etichette. Potete usare il sistema

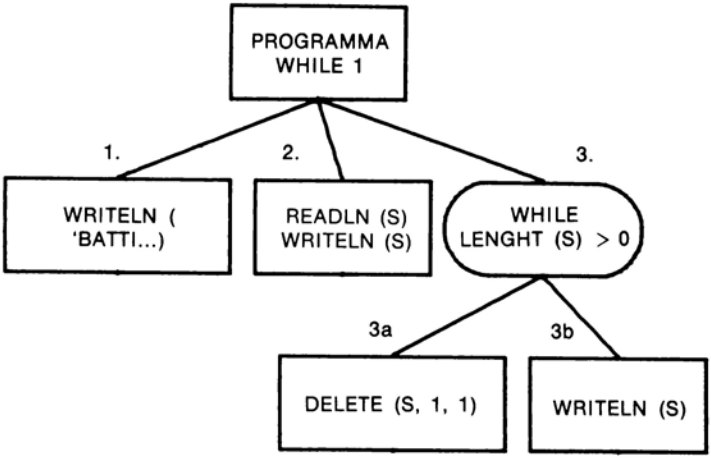


Figura 6-8

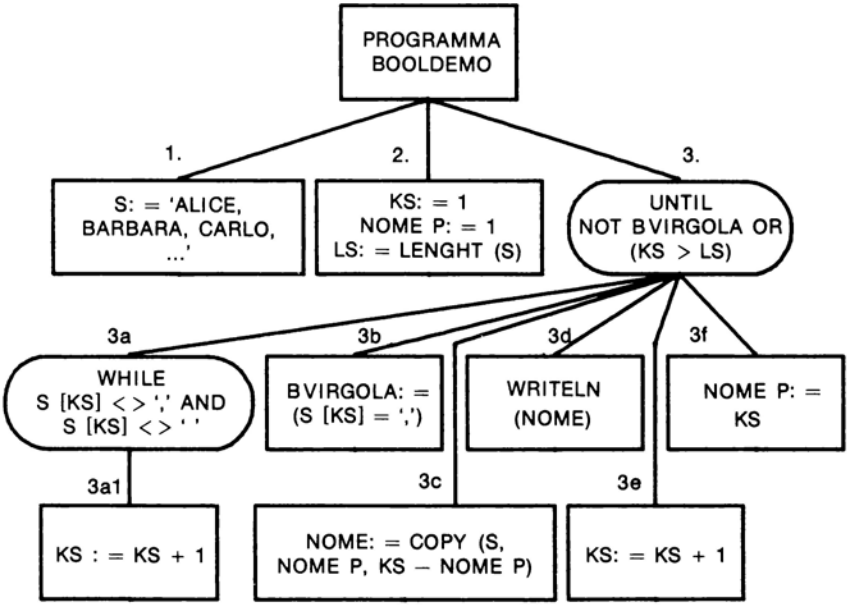


Figura 6-9

che abbiamo usato in questo capitolo nell'esempio dell'algoritmo "autostradale".
In questo sistema, simile al sistema Dewey-Decimale per organizzare i libri in una libreria, un intero seguito da un punto (".") indica la posizione del nodo in quel livello. Per esempio il nodo 2. è il secondo nodo del 1° livello. Il nodo 3.4 è il quarto nodo al secondo livello, nel gruppo di nodi che appartengono al terzo nodo del primo livello. Il sistema usato per etichettare i nodi nella figura 6-9 è equivalente al sistema Dewey-Decimale, ma usa alternativamente interi e lettere. Dipende dai vostri gusti personali usare un sistema o l'altro. Una volta capite le relazioni tra i nodi di un albero, e descritti in un diagramma di struttura, è ancora questione di gusto personale il fatto di voler rappresentare l'algoritmo con tabelle di struttura (in notazione lineare). Noi abbiamo usato tabelle simili alle tabelle di struttura nella sezione Obiettivi, in ogni capitolo del libro. Come potete notare i diagrammi e le tabelle di struttura sono perfettamente equivalenti nell'espressione di relazioni.

d) Quando un ramo di un blocco di *scelta* porta ad un gruppo di nodi, abbiamo l'equivalente della costruzione BEGIN...END di PASCAL (istruzione composta). Questo è mostrato nella figura 6-10. Il concetto di BEGIN ... END, per racchiudere un gruppo di istruzioni, è equivalente ad introdurre un livello fittizio in cui non ha luogo azione alcuna. Nella figura 6-10, i nodi 1b e 1b3a1 sono nodi "che non fanno niente" e sono posti nello schema per chiarire la struttura del diagramma. È spesso utile inserire o porre a lato di questi nodi, raccoglitori di blocchi, dei commenti.

e) Il concetto di procedura e di attivazione (chiamata) di una procedura, da un certo punto di vista si adatta bene ad essere espresso in un diagramma di struttura, da altri no. Quando un algoritmo è abbastanza complicato da non poter essere descritto in una pagina, potete "tagliare via" una parte dell'albero partendo da un qualsiasi nodo che abbia diramazioni. Questo nodo serve quindi come radice in un sotto-albero che può essere rappresentato con uno schema più piccolo su un'altra pagina. Lo stesso nome rimane nel diagramma "padre", ed avrà un nome che fa capire che il sotto-albero è effettivamente attaccato in quel punto. Questo concetto è chiaramente molto simile a quello di spezzare un programma PASCAL in procedure.

L'idea di usare blocchi di forme diverse per indicare concetti distinti in un diagramma di struttura, crea difficoltà all'aumentare del numero dei concetti. Poichè una procedura rappresenta un tipo particolare di *azione*, abbiamo scelto di rappresentare una procedura con un rettangolo con bordo doppio. Il blocco procedura è il nodo radice del diagramma di struttura rappresentante la procedura, come in figura 6-10 e 6-12. Il blocco procedura è un nodo ausiliario ogni volta che una procedura deve essere richiamata, come nel nodo 2a1a di

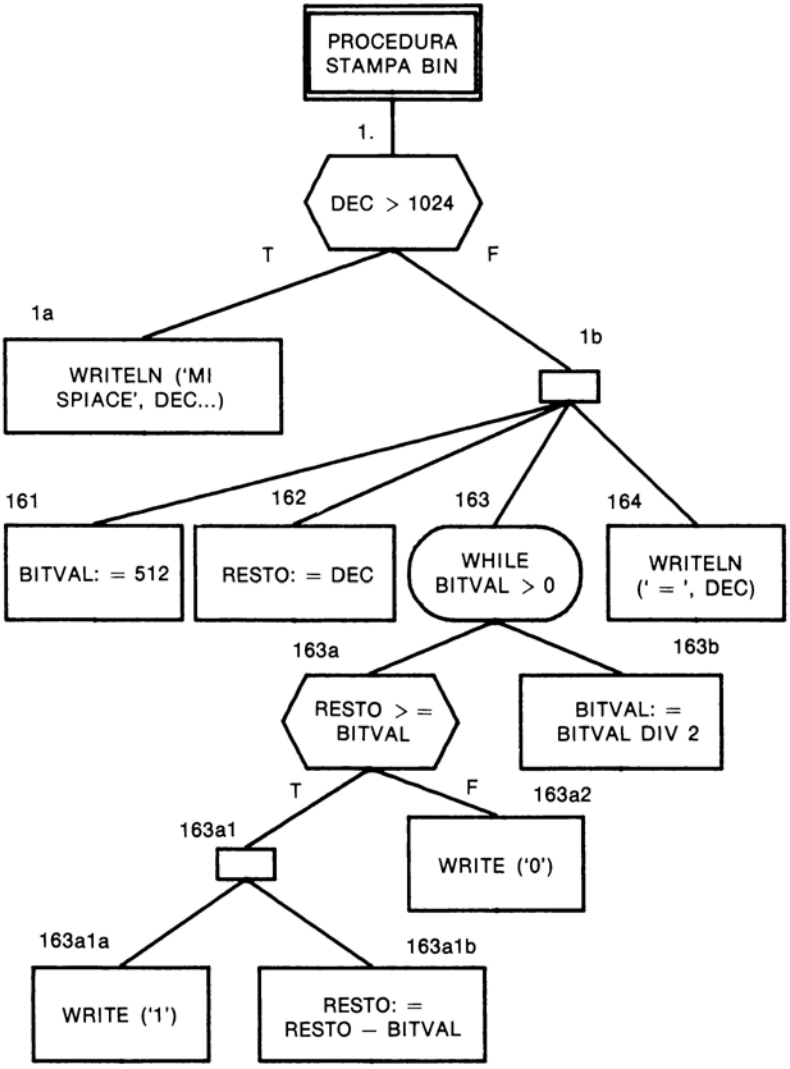

PROCEDURA
STAMPA BIN
1.
DEC > 1024
T
F
1a
WRITELN ('MI
SPIACE', DEC...)
1b
161
162
163
164
BITVAL: = 512
RESTO: = DEC
WHILE
BITVAL > 0
WRITELN
(' = ', DEC)
163a
163b
RESTO > =
BITVAL
BITVAL: =
BITVAL DIV 2
T
F
163a2
163a1
WRITE ('0')
163a1a
163a1b
WRITE ('1')
RESTO: =
RESTO – BITVAL

Figura 6-10

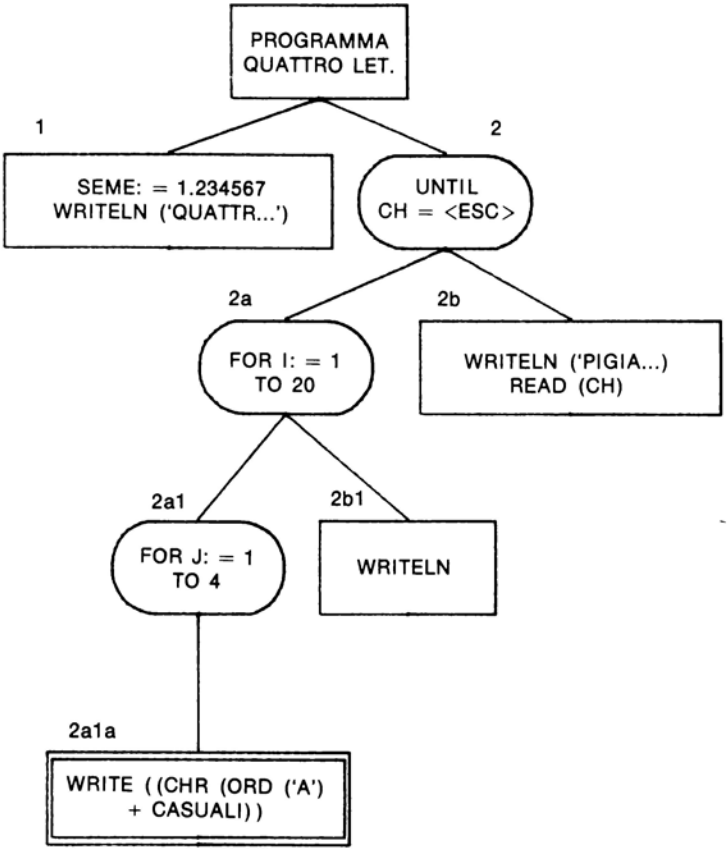


Figura 6-11

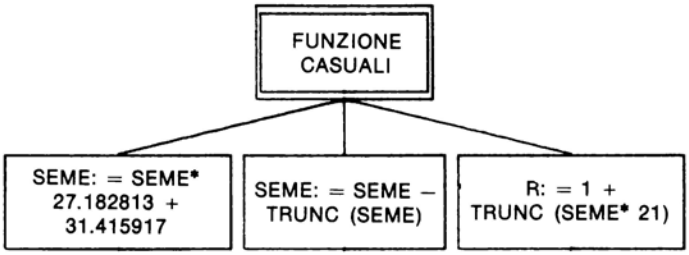


Figura 6-12

figura 6-11. In questo caso la "procedura" è in effetti una chiamata ad una funzione CASUALI, rappresentata *sottolineata* nel nodo 2a1a.

f) Notate che l'indentazione che abbiamo usato nei programmi e nelle tabelle di struttura, porta ad una relazione diretta con il concetto di *livello* in un diagramma di struttura. Maggiore è il livello, maggiore è il numero di colonne di indentazione. In pratica, un programma correttamente indentato ed una tabella di struttura sono parenti stretti.

Se capite il concetto di raffinamenti successivi, dovreste essere in grado di vedere il ruolo giocato dai commenti in un programma. Quando progettate un algoritmo al primo livello concettuale, scrivete una breve descrizione di cosa dovrebbe essere fatto in ogni sezione principale del vostro diagramma di struttura. Queste descrizioni possono e dovrebbero essere messe nel programma come commenti, messi là dove dovrebbe esserci il corrispondente nodo. Con una buona indentazione e commenti di questo tipo, un programma PASCAL può essere costruito come l'immagine del diagramma di struttura che rappresenta. Alla fine potreste essere capaci di rappresentare tale programma come fosse l'equivalente diagramma e potreste così, spesso, evitare la difficoltà di disegnare realmente il diagramma.

8. Soluzione di un problema basato su descrizione concettuale

Dovreste ora essere in grado di affrontare un problema di una certa qual maggior complessità, di quelli presentati fino ad ora. In questa sezione presenteremo un problema che sembra concettualmente semplice, cionondimeno richiede di essere pensato attentamente per essere tradotto in algoritmo ed infine in un programma.

Il problema consiste nello scrivere e verificare una funzione PASCAL che duplica l'azione eseguita dalla funzione interna POS per lavorare con variabili STRING. L'unica funzione interna per trattare stringhe che il vostro algoritmo dovrebbe sfruttare è LENGTH (vale a dire non usate CONCAT, COPY, DELETE, INSERT o, naturalmente, POS). Potete usare la funzione interna POS per verificare i vostri risultati.

Diamo una schematica descrizione di come lavora POS, facendo riferimento alla figura 6-13, come esempio. Data una stringa soggetto in una variabile stringa SOG, ed una configurazione che può essere pensata memorizzata in ELEM, possiamo usare le variabili INTEGER IE e IS per puntare alle locazioni in queste due stringhe. Partendo con ambedue i puntatori ad 1, IS è incrementato di un posto alla volta, ed in ogni locazione si verifica se il carattere a SOG[IS] è uguale al primo carattere di ELEM. Se no, IS è di nuovo incrementato di uno ed il ciclo continua. Se sono uguali, allora comincia un ciclo interno nel quale IS rimane fermo, ed IE avanza per far riferi-

mento ai successivi caratteri sia di SOG che ELEM. Questo ciclo interno continua fino a che sono trovati due caratteri diversi, oppure tutti i caratteri di ELEM sono stati confrontati con i corrispondenti in SOG. Se il ciclo interno non trova le stringhe uguali, allora il ciclo più esterno deve essere ripreso nel punto in cui si era fermato, con la ricerca di un nuovo accordo con il primo carattere di ELEM. Se non è trovata nessuna uguaglianza, prima che IS diventi troppo grande perchè l'accordo sia possibile, allora l'algoritmo riporta 0 come valore di POS. Se è trovata una stringa eguale, allora POS sarà uguale al valore di IS quando si è verificata l'uguaglianza sul primo carattere.

SOG LS = 25

1	2	3	4	5	6	7	8	9	10	11	12
I	–	F	I	G	L	I	–	T	U	O	I

ELEM LE = 3

1	2	3
G	L	I

IS	SOG [IS – 1 + IE]	ELEM [IE]	IE		UGUALE
1	I	G	1	<>	F
2	–	G	1	<>	F
3	F	G	1	<>	F
4	I	G	1	<>	F
5	G	G	1	=	T
5	L	L	2	=	T
5	I	I	3	=	T
			4 > LE		

Nota: Il carattere '–' sta per una spaziatura

Figura 6-13

Il primo passo per capire come risolvere questo problema potrebbe essere quello di creare un'immagine mentale di ciò che succede aiutandovi con un disegno, tipo quello delle due stringhe SOG e ELEM in figura 6-13. Poi dovreste cominciare a definire le variabili che pensate di usare nel programma. Assunti certi valori per queste variabili, dovreste scrivere i valori, per le variabili che cambiano, in punti abbastanza caratteristici dell'algoritmo concettuale per cominciare a vedere certe strutture ripetitive in ciò che si sta facendo. La tavola in figura 6-13 è stata preparata per questo scopo. Dovreste percorrere questa tabella passo-passo per verificare che rappresenta proprio l'azione descritta per la funzione POS.

Una volta che avete una chiara descrizione concettuale dell'algoritmo, ed un'idea generale su quali variabili usare per memorizzare i dati, potete tracciare una prima descrizione dell'algoritmo. La figura 6-14 è una parziale soluzione al problema nella forma di un grezzo diagramma di struttura. Abbiamo disegnato i vari blocchi del diagramma nell'ordine mostrato dai numeri cerchiati. Voi potete vedere il problema in modo diverso e disegnare questi stessi blocchi, od altri, in modo un po' diverso. La nostra idea base è di usare una variabile booleana UGUALE per controllare le iterazioni. All'inizio si assume che UGUALE è FALSE per il ciclo esterno. Quando il carattere iniziale di ELEM è trovato in SOG, UGUALE è posta provvisoriamente a TRUE mentre è in esecuzione il ciclo interno. L'assegnamento nel blocco (6) riporta UGUALE a FALSE non appena il carattere in SOG non è uguale al corrispondente in ELEM. Lascerà UGUALE TRUE se tutti i caratteri sono uguali fino a che IE è maggiore della lunghezza di ELEM, LE. Quindi terminano entrambi i cicli, e nel blocco (7) può essere verificata la condizione che ha posto termine al ciclo esterno. Non sono stati dettagliati la inizializzazione ed i controlli su IS ed IE.

Una volta che il diagramma di struttura dell'algoritmo è stato tracciato a questo grezzo livello di progetto, bisogna entrare nei particolari. L'algoritmo deve, naturalmente, essere progettato per adattarsi ad una gran varietà di casi speciali ritornando sempre dei valori corretti. Come aggiungete dettagli, dovreste tornare indietro allo stadio dei calcoli carta-penna per essere sicuri che tutti i passi del *"modello"* concettuale dell'algoritmo operano ancora come previsto. A questo punto potreste trovare dei problemi, e potreste dover tornare indietro e riprogettare una parte dell'algoritmo per farlo lavorare correttamente.

Quando pensate di avere un algoritmo abbastanza dettagliato in modo da essere sicuri che lavora correttamente, è ora di cominciare la traduzione del diagramma di struttura in un programma PASCAL. Notate che nel nostro esempio dovrete usare un identificatore diverso da POS per poter usare la funzione interna POS per verificare i vostri risultati.

Potete ora battere il programma e memorizzarlo su disco con l'aiuto dell'editor e potete tentare di compilare il programma. Dovrete probabilmente togliere un piccolo

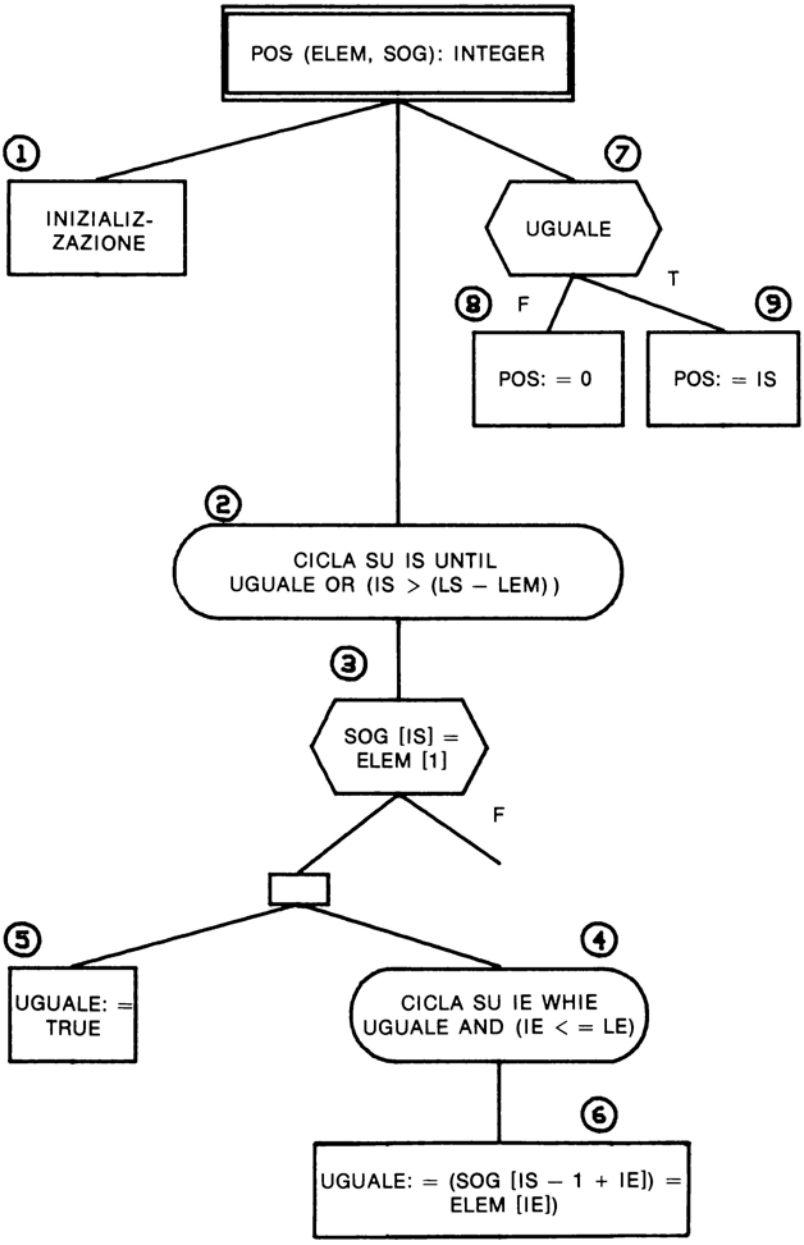

POS (ELEM, SOG): INTEGER
1
INIZIALIZ-
ZAZIONE
7
UGUALE
T
8
F
9
POS: = 0
POS: = IS
2
CICLA SU IS UNTIL
UGUALE OR (IS > (LS – LEM))
3
SOG [IS] =
ELEM [1]
F
5
UGUALE: =
TRUE
4
CICLA SU IE WHIE
UGUALE AND (IE < = LE)
6
UGUALE: = (SOG [IS – 1 + IE]) =
ELEM [IE])

Figura 6-14

numero di errori di sintassi. Se trovate, a questo stadio, un alto numero di errori di sintassi, risparmierete sicuramente tempo rivedendo con cura il programma sulla carta prima di procedere.

E finalmente ecco lo stadio della verifica logica e della correzione. Non è sufficiente avere il programma che gira e dà risultati corretti per un insieme di dati di verifica. Dovreste aggiungere, temporaneamente, istruzioni per tracciare l'esecuzione ed essere quindi sicuri che l'esecuzione del programma si accorda strettamente con i dati tabellari che voi avevate preparato per il modello concettuale, come in figura 6-13. Poi, se tutto è andato bene fino a questo punto, dovreste preparare dei dati per provare il programma nei vari casi particolari che possono nascere. Potrete ritenere il programma ragionevolmente privo di bachi quando opera correttamente per tutti i possibili casi che voi avete trovato.

ESERCIZIO 6.1:
Completate il progetto e la verifica della funzione sostitutiva POSizione come descritta in questa sezione. Come minimo dovrete verificare che funziona correttamente nei seguenti casi:

a) Configurazione esistente nel soggetto
b) Configurazione non esistente nel soggetto
c) Configurazione in posizione sinistra (estrema)
d) Configurazione in posizione destra (estrema)
e) Configurazione lunga 1 carattere in ogni posizione
f) Configurazione lunga almeno 3 caratteri
g) Soggetto lungo 1 carattere
h) Soggetto più corto della configurazione (no eguaglianza)
i) Soggetto o configurazione vuoti (no eguaglianza)

9. Due problemi sfida

In questa sezione vi presentiamo 2 problemi sufficientemente difficili per cui la loro soluzione vi costerà un bel po' di tempo. Ormai conoscete tutti gli strumenti di programmazione che PASCAL vi offre per trattare questi problemi. Ciò che caratterizza questi problemi rispetto a quelli analizzati fino ad ora è che essi richiedono molta riflessione per creare l'algoritmo necessario in un modo relativamente semplice.

Il problema delle Torri di Hanoi è un classico della scienza degli elaboratori. Può essere risolto con un programma non più lungo di 40 righe, con un'unica istruzione per linea. Una soluzione ragionevole dovrebbe includere anche linee di commento, e delle procedure per la tracciatura in fase di correzione. Il problema dei Billardi è un

esempio di un gran numero di disegni "animati" che potreste creare con l'elaboratore. Se il microelaboratore che state usando permette che la Tartaruga disegni una linea nera (BLACK) sulle linee bianche (WHITE), vi è possibile cancellare ogni percorso della palla prima di fare il prossimo, dando perciò l'illusione di una palla che si muove. Questo problema non è concettualmente difficile quanto le Torri di Hanoi, ma il programma che implementa i disegni animati dei Biliardi sarà più lungo di 100 linee. Se trovate che uno qualsiasi di questi problemi è per voi troppo difficile a questo stadio, lasciatelo perdere fino a che, avendo progredito nel testo, avrete più esperienza nella soluzione dei problemi. L'esperienza acquisita risolvendoli con successo vi aiuterà nell'uso futuro dell'elaboratore.

ESERCIZIO 6.2: Torri di Hanoi

Forse avete già visto questo problema sotto forma di gioco di pazienza venduto nei grandi magazzini. Il problema è illustrato in figura 6-15. Vi sono consegnati tre sostegni (le "torri") ed alcuni dischi possono essere infilati dall'alto in uno qualsiasi dei sostegni. Ogni disco ha diametro diverso.

All'inizio tutti i dischi sono nella posizione di sinistra che potremmo chiamare 'A'. Il problema consiste nello spostare tutti i dischi nella posizione destra, che possiamo chiamare 'C', in modo che non succeda mai che un disco stia sopra ad un altro di diametro inferiore. Il problema può essere risolto per un qualunque numero di dischi inizialmente in 'A'. La figura 6-15 illustra i vari passi necessari nel caso di 2, 3 e 4 dischi.

Per due dischi, il posto intermedio ('B') è usato temporaneamente per salvare il disco più piccolo; questo permette al disco più grande di essere messo in 'C'. Lo postamento può quindi essere completato muovendo il disco più piccolo da B a C, e mettendolo di nuovo sul disco più grande.

Per tre dischi, è necessario muovere i due più piccoli sulla torre di mezzo ('B') per poter spostare in 'C' il disco più largo. Per muovere i due dichi sopra in B, usate la stessa logica vista per il problema dei due dischi, con B e C scambiati. Dopo che il disco più grande è stato spostato in C, i due rimanenti su B possono essere spostati in C, usando A come deposito temporaneo del disco più piccolo.

Per quattro dischi, si spostano dapprima i 3 sopra in B. Il più grande è quindi messo in C e infine i tre più piccoli sono spostati in C. Questo è fatto usando lo stesso algoritmo dei tre dischi, ma con i ruoli delle torri A e B scambiati.

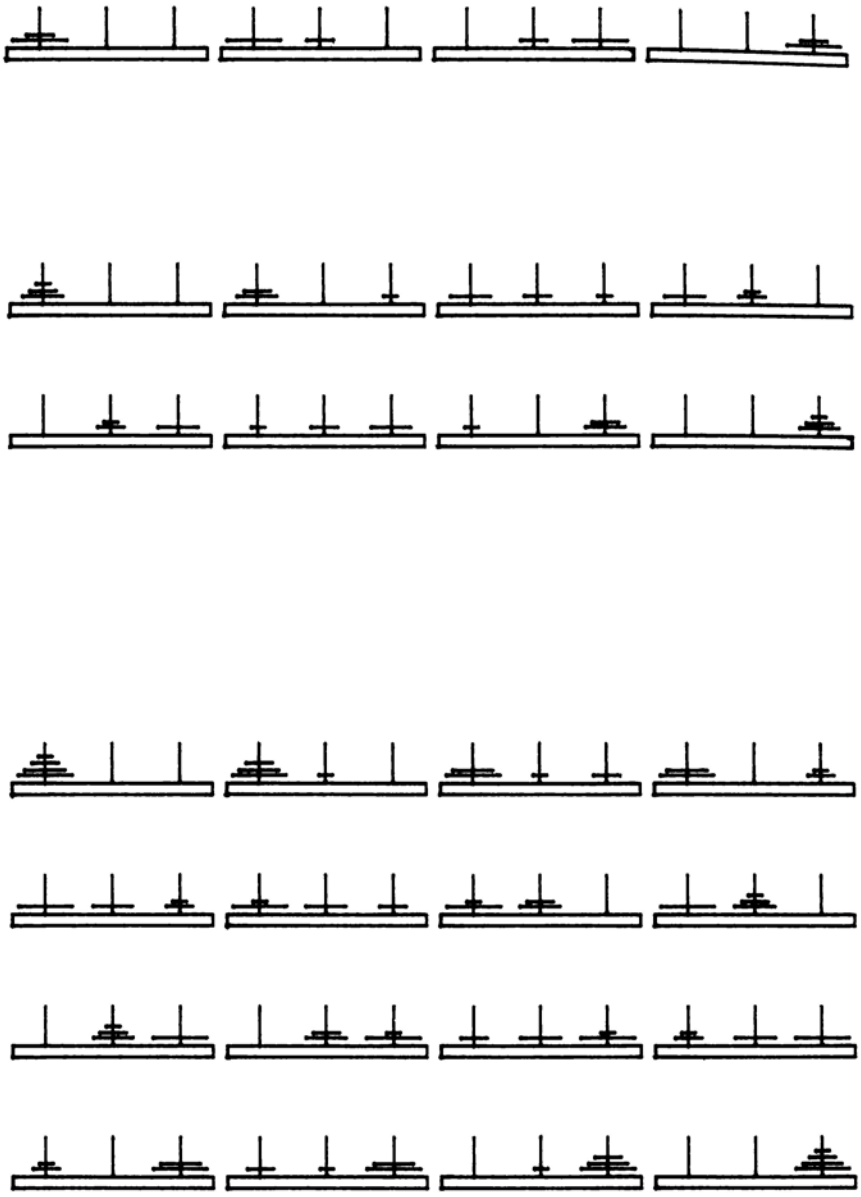


Figura 6-15

Ormai dovreste aver notato che c'è una configurazione *recursiva* nel modo di risolverlo. Per risolvere il problema per N dischi, lo si risolve semplicemente per N—1 dischi usando le torri di destinazione alternate. Il disco più grande può quindi essere spostato. Infine la soluzione di N—1 dischi è ripetuta con lo scambio di ruoli di A e B.

La figura 6-16 suggerisce un modello concettuale per un sotto-algoritmo HANOI che può essere usato come base per una procedura recursiva dello stesso nome. In questo caso facciamo riferimento a sostegni Sorgente (SOR), Destinazione (DES) ed alternato (ALT) al posto di A, B e C, in quanto l'ultimo cambia ruolo in successivi livelli di recursione. CON rappresenta il conto dei dischi che devono essere messi quando HANOI è chiamata. La figura 6-16b rende l'algoritmo più esplicito in termine di chiamate di sotto-algoritmi (procedure).

Per rappresentare le torri abbiamo usato tre variabili STRING A, B e C. I dischi sono rappresentati dai caratteri '1', '2', '3' e '4' ed il carattere rappresenta la dimensione del disco. L'assenza di un disco è rappresentata da un carattere "spazio". Le Torri B e C all'inizio sono "piene" di spazi. La torre A, nel problema a 4 dischi, è inizializzata a '4321'. È conveniente usare tre contatori NA, NB ed NC per rappresentare, in ogni istante, i dischi sulla torre corrispondente. Sarebbe possibile usare, al posto dei contatori, la funzione LENGTH.

Con questa introduzione dovreste riuscire a completare il problema. La procedura TRACCIA può essere usata per visualizzare (su una linea, ogniqualvolta TRACCIA è chiamata) il contenuto di ciascuna torre tutte le volte che HANOI gira, e quando il programma termina. Solo come ulteriore sfida dovreste cercare di fare un "bel" disegno delle torri sul vostro schermo.

ESERCIZIO 6.3: il gioco dei Biliardi
Come avrete probabilmente visto in posti pubblici, dei piccoli elaboratori sono ormai molto usati per simulare "giochi televisivi". Con un elaboratore dotato di uno schermo grafico che permette, selettivamente, di rappresentare e cancellare figure dallo schermo, sono praticamente infinite le possibilità di programmare simili giochi. Questo problema — il gioco dei biliardi — dà un esempio di tali giochi.

La figura 6-17 mostra il disegno "animato" che il programma che scriverete per questo esercizio dovrebbe creare. È mostrato su uno schermo che non ha la possibilità di cancellazione selettiva per cui si vedono le successive posizioni della palla. Il programma dovrebbe

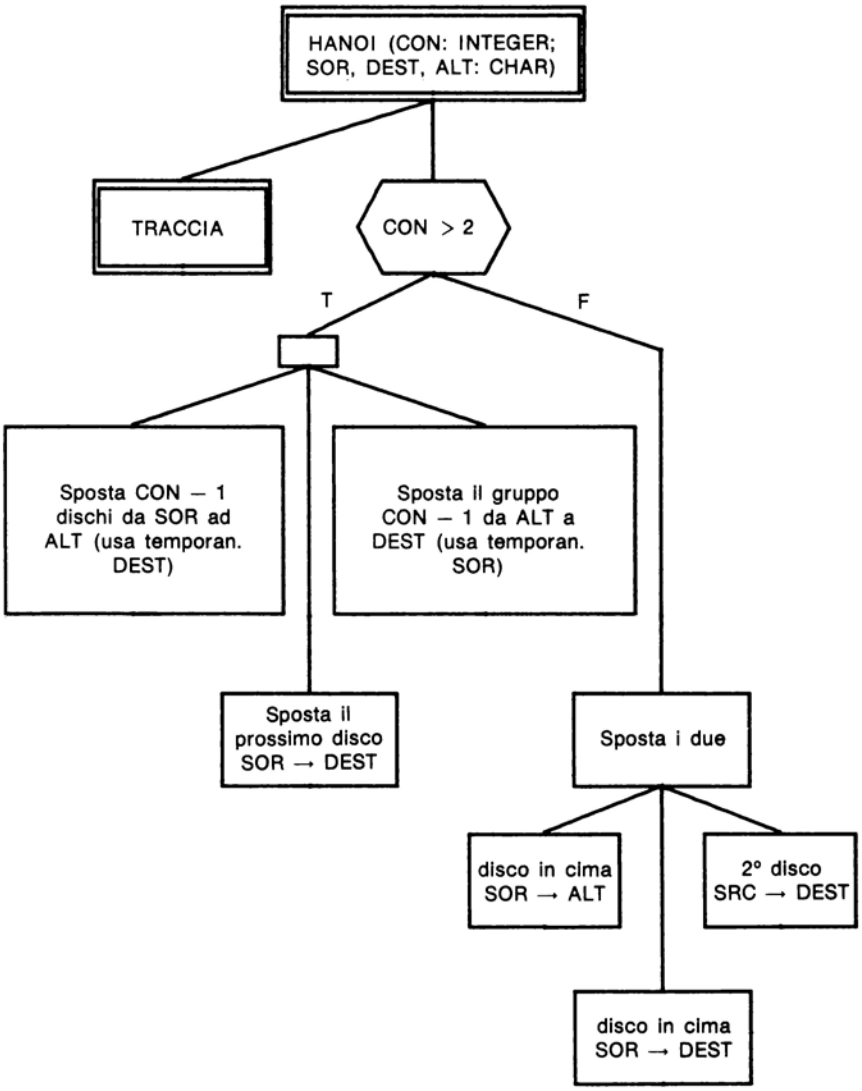

HANOI (CON: INTEGER;
SOR, DEST, ALT: CHAR)
TRACCIA
CON > 2
T
F
Sposta CON – 1
dischi da SOR ad
ALT (usa temporan.
DEST)
Sposta il gruppo
CON – 1 da ALT a
DEST (usa temporan.
SOR)
Sposta il
prossimo disco
SOR → DEST
Sposta i due
disco in cima
SOR → ALT
2° disco
SRC → DEST
disco in cima
SOR → DEST

Figura 6-16a

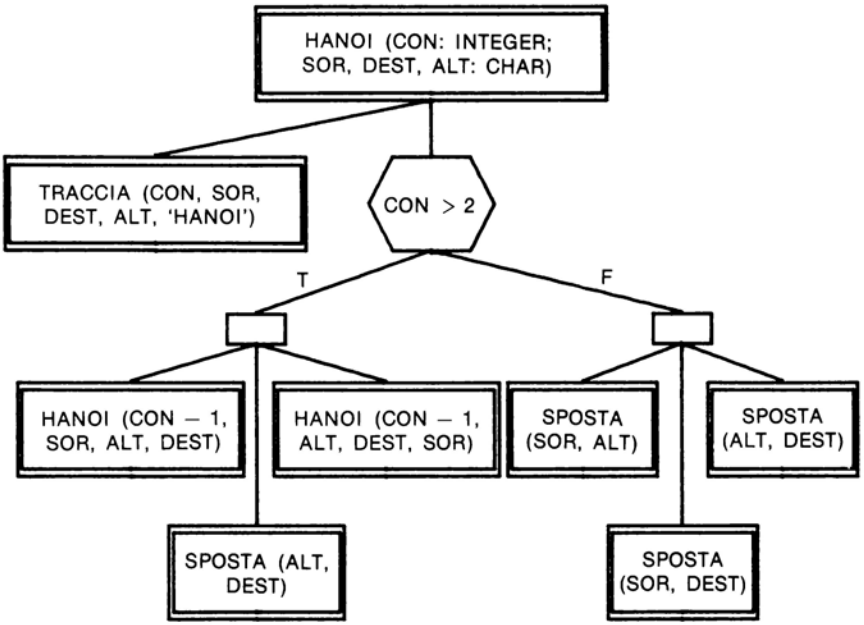


Figura 6-16b

riuscire a trattare qualunque direzione iniziale del moto; direzione immessa via tastiera, in gradi. Quando una palla "colpisce" una sponda viene "riflessa". L'angolo compreso tra la direzione originale del moto e la sponda ha lo stesso valore dell'angolo formato dalla direzione finale riflessa e la sponda del lato opposto del punto di riflessione. Per esempio, se la palla colpisce la sponda superiore con un angolo di 45 gradi, uscirà con –45 gradi quando è riflessa; se colpisce la sponda destra con un angolo di 45 gradi, verrà riflessa a 135 gradi.

Il programma dovrebbe rilevare quando la palla cade in una delle 6 "buche" e proclamare un "vincitore". Altrimenti la palla rallenta nei successivi rimbalzi ed infine si ferma in una nuova posizione. Il programma dovrebbe quindi richiedere una nuova direzione per la prossima "giocata".

La figura 6-18 dà un primo diagramma di struttura del programma che usiamo per risolvere questo problema. Una possibile complicazione è il calcolo di NUOVOX e NUOVOY dopo ogni passo nel ciclo che simula il moto della palla. Ciò può essere fatto nel modo seguen-

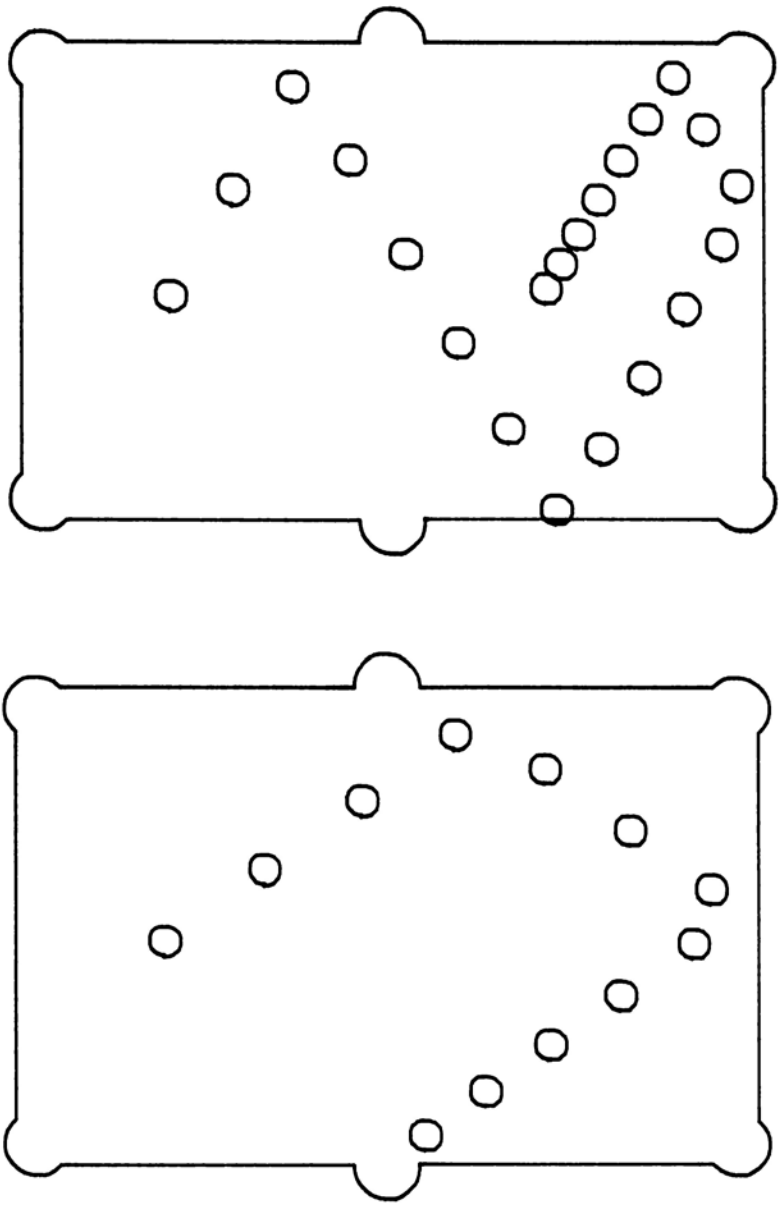


Figura 6-17

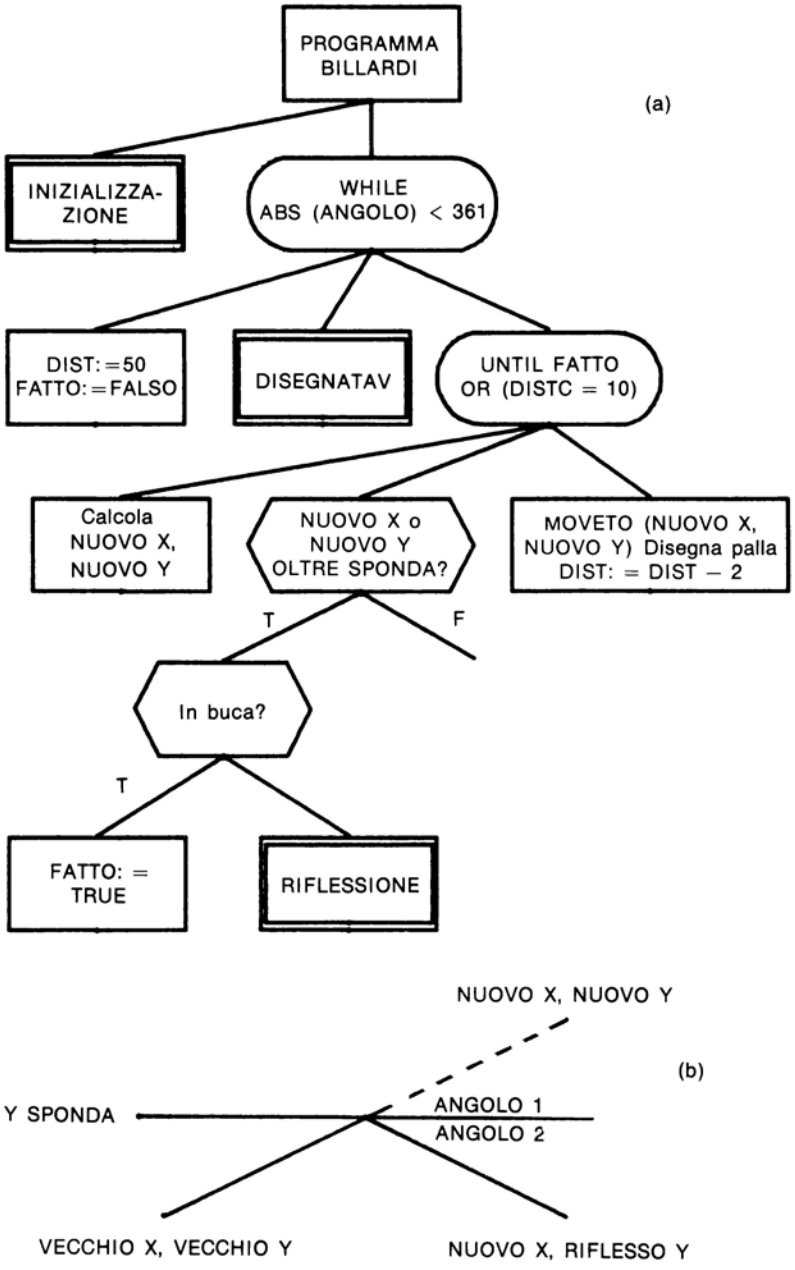

PROGRAMMA BILLARDI
(a)
INIZIALIZZA-ZIONE
WHILE ABS (ANGOLO) < 361
DIST: =50 FATTO: =FALSO
DISEGNATAV
UNTIL FATTO OR (DISTC = 10)
Calcola NUOVO X, NUOVO Y
NUOVO X o NUOVO Y OLTRE SPONDA?
MOVETO (NUOVO X, NUOVO Y) Disegna palla DIST: = DIST – 2
T
F
In buca?
T
FATTO: = TRUE
RIFLESSIONE
NUOVO X, NUOVO Y
(b)
Y SPONDA
ANGOLO 1
ANGOLO 2
VECCHIO X, VECCHIO Y
NUOVO X, RIFLESSO Y

Figura 6-18

te: assumendo che NUOVOX, NUOVOY, e NUOVOANGOLO siano variabili INTEGER, allora

```
WHEREAMI (NUOVOX, NUOVOY, NUOVOANGOLO)
```

assegnerà il valore corrente descrivente la posizione della tartaruga a queste 3 variabili usate come parametri variabile. WHEREAMI è una procedura interna per lavorare con i grafici tartaruga.

Fate riferimento alla parte (b) della figura 6-18 per vedere una rappresentazione geometrica che mostra la riflessione della palla di biliardo che batte contro la sponda alta. Se il centro della palla parte dalla posizione (VECCHIOX, VECCHIOY), un MOVE(DIST) nella direzione originale ANGOLO1 farà sì che la tartaruga arrivi in (NUOVOX, NUOVOY) che è oltre le sponde. Notate che DIST potrebbe essere talmente piccolo che (NUOVOX, NUOVOY) non necessariamente sarebbe un punto fuori dallo schermo. Nell'esercizio mostrato, NUOVOX sarà lo stesso ottenuto usando WHEREAMI, dopo MOVE(DIST), con PENCOLOR(NONE) abilitato. Ma NUOVOY deve essere convertito per RIFLESSY, e il valore di Y che risulta dalla riflessione della sponda. Questo può essere ottenuto considerando che RIFLESSY è *sotto* la sponda quanto NUOVOY è sopra. Perciò:

```
RIFLESSY := 2*YSPONDA – NUOVOY;
ANGOLO2 := – ANGOLO1
```

produrrà il valore di Y ed il nuovo angolo, MOVETO e TURNTO serviranno per stabilire la nuova posizione della tartaruga dopo la riflessione. Regole geometriche come queste si applicano a tutte e tre le altre sponde. Notate che RIFLESSIONE non deve tracciare le linee mostrate nella parte (b) della figura 6-18. Deve solo trovare la nuova posizione della tartaruga, cosicchè possa essere disegnata la prossima posizione della palla.

Problemi

PROBLEMA 6.1:
Scrivete un programma PASCAL completo per eseguire l'algoritmo della figura 6-19. (SQRT(X) è una funzione interna che calcola la radice quadrata di X). ESCI è booleano; gli altri identificatori sono interi. Per coloro che sono orientati alla matematica dirò che questo algoritmo calcola i primi 100 numeri primi. Per gli altri diò che la solu-

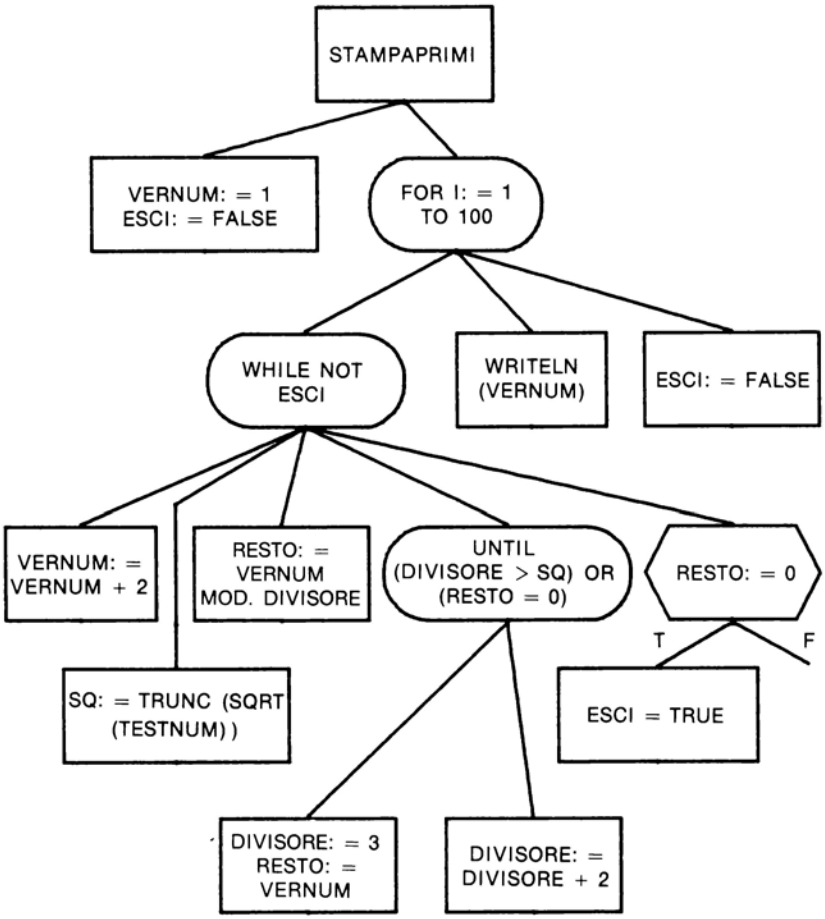

STAMPAPRIMI
VERNUM: = 1
ESCI: = FALSE
FOR I: = 1
TO 100
WHILE NOT
ESCI
WRITELN
(VERNUM)
ESCI: = FALSE
VERNUM: =
VERNUM + 2
RESTO: =
VERNUM
MOD. DIVISORE
UNTIL
(DIVISORE > SQ) OR
(RESTO = 0)
RESTO: = 0
T
F
SQ: = TRUNC (SQRT
(TESTNUM))
ESCI = TRUE
DIVISORE: = 3
RESTO: =
VERNUM
DIVISORE: =
DIVISORE + 2

Figura 6-19

zione di questo problema non richiede che voi sappiate per cosa saranno usati questo algoritmo o l'equivalente programma.

PROBLEMA 6.2:
Scrivete un programma PASCAL completo che rappresenti l'algoritmo mostrato in figura 6-20. Tutte le variabili sono interi. Per scrivere il problema non è necessario che voi sappiate per cosa possa essere usato l'algoritmo.

PROBLEMA 6.3:
Disegnate dei diagrammi di struttura per rappresentare CONTAPAROLE (capitolo 4 sezione 6) e RCONT (Capitolo 4, sezione 8).

PROBLEMA 6.4:
Disegnate un diagramma di struttura per rappresentare il programma SUALBERO (Capitolo 4, sezione 10).

PROBLEMA 6.5:
Disegnate un diagramma di struttura per il programma DECBIN (Capitolo 5, sezione 8).

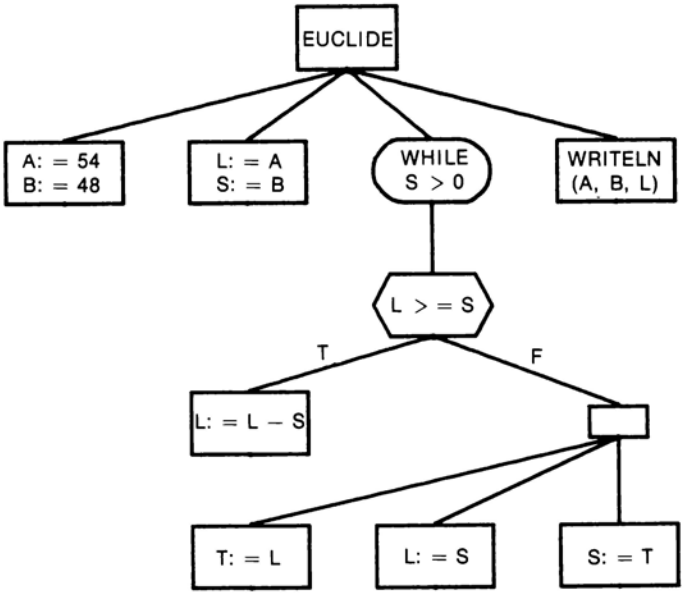


Figura 6-20

CAPITOLO 7

IMMISSIONE DATI

1. Obiettivi

Questo capitolo è dedicato all'immissione dei dati nel programma da dispositivi esterni, in modo particolare di dati inseriti dalla tastiera abbinata al vostro elaboratore.

1a. Apprendimento dell'uso delle istruzioni READ e READLN per la manipolazione di numeri, singoli caratteri, e stringhe. Uso di READ con e senza eco automatico di ogni carattere inserito dalla tastiera.

1b. Controllo delle iterazioni di un programma che legge una sequenza di valori la cui lunghezza non può essere prevista al momento della scrittura del programma. Uso delle funzioni interne EOF e EOLN.

1c. Apprendimento della convalidazione dei valori immessi per assicurarsi della loro correttezza o perlomeno dell'appartenenza a quella gamma di valori che può essere manipolata dal programma senza portare a dei risultati abnormi.

1d. Sviluppo di programmi che risolvano problemi specifici richiedenti l'immissione di valori dalla tastiera.

2. Premessa

Abbiamo fin qui evitato molte delle complicazioni associate alla comunicazione delle informazioni in un programma per mezzo di dispositivi esterni. Non tutti i programmi sono progettati per elaborare dati esterni, ma la maggior parte di essi lo è. Fino a non molto tempo fa, lo strumento più comunemente usato per l'*"immissione"* dei dati in un programma era il lettore di schede perforate. Recentemente, gran parte degli utenti di grossi elaboratori ha optato per l'uso di cassette magnetiche, cartucce, dischi o terminali a distanza per la comunicazione di dati in grossi elaboratori. Sul mi-

croelaboratore usato nello studio di questo libro, userete molto probabilmente, una tastiera su terminale CRT, oppure una tastiera direttamente collegata all'elaboratore, per l'immissione dei dati.

Indipendentemente dal tipo di strumento usato per l'immissione dei dati, il metodo standard per la comunicazione di questi al vostro programma, si serve dell'istruzione READ, o dell'altra, strettamente affine, che è READLN. Su alcuni elaboratori e nel caso di alcuni linguaggi di programmazione, l'uso dell'istruzione READ diventa il comando, al sistema, di analizzare fisicamente i dati contenuti in un *"record"* completo. Per esempio, tutte le 80 colonne di una normale scheda perforata possano contenere dei dati, che di solito sono fra loro indipendenti. Tutta la scheda viene fisicamente analizzata al momento dell'esecuzione di un'istruzione READ, e ciascuno dei gruppi indipendenti di dati riportati sull' *"immagine di scheda"* viene interpretato da un *"formato"* a parte. In PASCAL, ciascuna istruzione READ o READLN è progettata per interpretare uno o più gruppi di dati, inseriti dal dispositivo di immissione, senza bisogno di una rigida osservanza del record.

La gestione del *"formato"* usato in alcuni linguaggi di programmazione è in realtà un insieme di procedure interne che permettono la traduzione dei caratteri manipolati da molti dei dispositivi *"periferici"* (cioè esterni) nella forma binaria *"interna"*, nella quale vengono manipolati e memorizzati i valori Interi e Reali. In PASCAL, la traduzione dipende dal <tipo> di una variabile nominata in un'istruzione READ o READLN. Analogamente, nell'emissione con l'uso di istruzioni WRITE o WRITELN, viene attuata la traduzione inversa.

In alcuni sistemi di elaborazione, il concetto di separati record di dati non esiste. I dati vengono invece considerati come un flusso continuo di caratteri. Questo concetto è difficilmente visualizzabile nel caso di schede perforate, ma è naturalmente applicato alla comunicazione da un *"terminale"* di tipo *"conversazionale"*, cioè un dispositivo che vi permette di "conversare" con l'elaboratore. Questi terminali sono detti anche *"interattivi"* in quanto l'utente può interagire abbastanza direttamente con l'elaboratore, piuttosto che attraverso un uso più indiretto di schede per l'immissione e di listati su stampanti, per l'emissione. Le istruzioni READ e WRITE di PASCAL, permettono di lavorare sia con registri di lunghezza determinata, che con flussi di dati di lunghezza variabile. Ciò vi permette una naturale flessibilità nella manipolazione di diversi formati di dati con l'uso di semplici istruzioni. Tuttavia questa flessibilità vi potrà occasionalmente richiedere la scrittura di un programma PASCAL leggermente più lungo per eseguire cose che potrebbero essere fatte in modo più breve e semplice con l'uso dei formati in altri linguaggi.

Molti dei programmi che manipolano dati di immissione devono essere scritti per far fronte ai cambiamenti nell'ammontare dei dati da un'elaborazione all'altra. Ciò significa che il vostro programma deve essere progettato in modo da poter stabilire

quando è stato letto l'ultimo gruppo di dati, cioè quando la fine del *"flusso"* di immissione è stato raggiunto. Il termine "flusso" si riferisce al meccanismo di trasferimento dei dati da e a qualsiasi dispositivo esterno. L'origine di questo termine risale all'uso di schede perforate, dove l'analogia con un flusso convenzionale di schede è evidente. Questa terminologia è ora generalizzata e si riferisce a qualsiasi dispositivo esterno, che sia una tastiera, un'unità video, una linea di comunicazione, ed altri.

Quando un programma impegnato nella lettura di dati, raggiunge l'ultimo gruppo di dati disponibile, si dice che il programma ha raggiunto la condizione di *"fine del flusso"* per il dispositivo associato. Nel caso di lettura di schede perforate, la fine del flusso o condizione *"EOF"* si presenta quando è stata letta l'ultima scheda. Nel caso di immissione da tastiera è necessario stabilire in base a delle convenzioni, in che modo potete segnalare all'elaboratore il raggiungimento della condizione EOF. Alcuni sistemi non prevedono un esplicito meccanismo interno per il riconoscimento della condizione di EOF. In questi casi è necessario che il programmatore sistemi il programma in modo che la comparsa di speciali valori all'interno del flusso di dati sia interpretata come raggiungimento della condizione EOF.

Faremo degli esempi di entrambi queste tecniche in questo capitolo.

3. Differenze fra i diversi sistemi di immissione/emissione

A causa delle tante differenze nelle caratteristiche fisiche fra i diversi dispositivi di immissione, dovete essere a conoscenza di alcune, dettagliate, differenze fra i metodi descritti in questo capitolo e quelli che potrete incontrare su altri elaboratori. Questo capitolo è la revisione sostanziale di un equivalente capitolo originariamente scritto per descrivere l'immissione per mezzo di schede perforate, su grossi elaboratori. Tutti i programmi richiesero una revisione fondamentale per renderli adatti all'uso della tastiera per l'immissione, e di un'unità video per la corrispondente emissione. Queste differenze fanno parte del "mondo reale" della programmazione che devono essere accettate da chiunque lavori regolarmente con gli elaboratori.

In questo capitolo vi daremo un piccolo assaggio del tipo di differenze che possono sorgere. Vi mostreremo come READ possa essere usato sia in una situazione in cui ciascun carattere inserito dalla tastiera abbia un'"eco" immediata sull'unità video, che in una in cui non si avrà nessuna eco. Avrete forse notato che il programma editor, da voi usato nella preparazione dei programmi PASCAL, usa l'ultimo metodo per la maggior parte di comandi. La possibilità senza eco permette un controllo esplicito di ciò che apparirà sul video in risposta alla battitura di ciascun carattere.

Come potrete vedere, entrambi i metodi comportano delle complicazioni che bisogna capire per poter scrivere dei programmi per raffinate operazioni di immissione.

Una delle differenze principali fra il lavorare con *"lotti"* di schede perforate per

l'immissione di dati, e con l'immissione via tastiera è data dal modo con cui il programmatore fornisce all' *"utente"* del programma il mezzo per comprendere le azioni del programma a stadi determinati. Nel caso delle schede perforate, o caso "lotto", le schede vengono sottoposte all'elaborazione in un'unica volta, ed i risultati, stampati, dell'emissione possono non essere disponibili per alcune ore. Anche se l'emissione fosse disponibile dopo pochi minuti dall'immissione delle schede, l'utente non ha la possibilità di interagire con il programma mentre sta girando. Quindi, qualsiasi informazione necessaria all'utente per determinare l'azione del programma dopo che l'elaborazione è stata completata, dovrà essere ricercata sui listati mentre il programma sta girando. Per questo motivo è buona regola nella correzione di un programma da lanciare in ambiente elaborazione a lotti, quella di inserire diverse istruzioni WRITELN causanti la stampa di copie di dati letti nel programma.

Nell'ambito interattivo, con immissioni dalla tastiera ed emissioni dal dispositivo video, il sistema farà eco di ciascun carattere inserito dalla tastiera facendolo apparire automaticamente sullo schermo, a meno che siano state prese delle speciali precauzioni per sopprimere l'eco cosicchè un programma scritto per fare l'eco sul dispositivo di emissione standard con istruzioni WRITE, come nell'ambiente a lotti, produrrà confusione sullo schermo, in quanto ciascuna voce inserita apparirà *due volte* (una volta con l'eco durante la battitura, un'altra volta quando visualizzata dall'istruzione WRITE).

Un'altra differenza è data dal fatto che l'utente di un programma interattivo dovrebbe essere tenuto informato di quando il programma aspetta l'immissione di dati dalla tastiera.

Se il programma raggiunge un'istruzione READ senza visualizzare nessuna informazione, smetterà di attendere l'immissione, come programmato, ma l'utente non capirà cosa stia succedendo!

Questa situazione non si presenta nel caso di un programma a schede, in quanto l'addetto alla perforazione delle schede deve essere a conoscenza in anticipo dei dati aspettati dal programma e di come questi dati debbano essere riportati sulle schede. Alcuni sistemi interattivi ideati per utenti ancora inesperti (scrive) WRITE automaticamente un segno di due punti (':'), o un altro carattere quale "suggerimento" all'utente, ogni volta che viene eseguita un'istruzione READ. Questo impedisce al programmatore di avere un controllo completo di ciò che appare sullo schermo. Nel nostro sistema PASCAL, non esiste nessuna scrittura automatica di caratteri di suggerimento, e dovrete quindi usare delle istruzioni WRITE per far apparire uno di questi caratteri.

4. Istruzioni READ e READLN

Per comprendere il funzionamento dell'immissione dati, ci dobbiamo interessare

alla *"semantica"* delle istruzioni di immissione, piuttosto che alla sintassi. In apparenza, la sintassi delle istruzioni READ e READLN è molto simile a quella di WRITE e WRITELN. Purtroppo non esiste un metodo abbastanza chiaro e conciso per descrivere le azioni svolte da READ e READLN, cioè la loro semantica, nello stessm senso con cui un diagramma fa una descrizione delle regole sintattiche. In questa e nelle prossime due sezioni, riporteremo alcuni esempi brevi che illustrino il funzionamento delle possibilità di immissione PASCAL. Il resto del capitolo è dedicato alla presentazione ed analisi di diversi programmi che usano l'immissione di dati.

Nella discussione che segue, sarà assunto che queste dichiarazioni si applicano a:

```
VAR CH:CHAR;
    I,J,K,L:INTEGER;
    R,Q:REAL;
    S:STRING;
```

Quando l'esecuzione di un programma raggiunge un'istruzione READ o READLN, il programma si ferma temporaneamente ed aspetta i dati, conformemente alla variabile nominata nell'elenco parametri, inseriti da tastiera. Per esempio:

```
READ(CH)
```

farà sì che il successivo carattere inserito dalla tastiera venga assegnato quale valore alla variabile CH.

```
READ(S)
```

aggiungerà tutti i caratteri inseriti alla fine di S fino a quando viene battuto il successivo <RET>. Se cercate di inserire in S più caratteri di quanto ne siano permessi dalla sua lunghezza dichiarata, di solito 80 caratteri, il sistema continuerà ad accettarli ed a farne eco ma essi non saranno assegnati alle locazioni di S. Nel caso commetteste un errore durante l'inserimento dei caratteri in S, potrete correggerlo usando il *"backspace"* o tasto <BS>, oppure il tasto <DEL> (<RUBOUT>). <BS> cancellerà un carattere da S ogni volta che viene premuto e il corrispondente carattere sparirà dallo schermo. <DEL> elimina l'intera linea, permettendovi di cominciare da capo la battitura di una nuova linea.

In entrambi i casi il numero di caratteri assegnati ad S verrà ridotto del numero di caratteri eliminati.

```
READ(I)
```

aspetterà la battitura di una <costante intera>. Se, mentre il programma è fermo sull'istruzione READ(I), voi inserite qualsiasi carattere che non sia una cifra (da '0' a

'9') oppure battete uno < spazio > bianco, il programma terminerà in modo abnorme. Dopo la battitura della prima cifra, READ(I) assumerà che il primo carattere non - cifra inserito significherà la fine dell'intero voluto. Questo si applica a < BS > e < DEL > come a qualsiasi altro carattere che non sia una cifra. Cosicchè non potete correggere un errore inserendo un intero con l'uso di questi tasti. (È possibile sistemare un programma in modo che accetti degli interi quale immissione, e che permetta l'uso di < BS > e < DEL > per correggere gli errori, ma questo argomento è al di fuori dello scopo di questo libro).

Per esempio, se il vostro programma contenesse le seguenti istruzioni:

```
READ(I)
READ(J);
READ(K);
READ(L);
```

e voi battete:

```
  273 4 15<RET>
58
```

il risultato sarebbe: I=273, J=4, K=15, L=58, dopo che tutte le istruzioni hanno completato la loro esecuzione.

< RET > dopo "15" è facoltativo e sta semplicemente a significare l'uso del tasto di ritorno per poter battere all'inizio della linea successiva. Le istruzioni READ (< intero >) interpretano l'uso di < RET > come equivalente alla battitura di uno < spazio >.

Se al momento in cui queste quattro istruzioni stanno aspettando l'immissione voi inserite qualsiasi carattere *che non sia uno < spazio >, < RET > o una cifra,* preceduto da una cifra, il programma terminerà in modo abnorme. Per esempio, la battitura:

```
273, 4, 15, 58
```

farà sì che ad I venga assegnato correttamente 273, ma il programma non terminerà correttamente nel caso di READ(J) a causa della virgola (',') battuta immediatamente dopo "273".

Come abbiamo già visto, la sintassi di READ è simile a quella dell'istruzione WRITE. In particolare è possibile avere un elenco parametri passato a READ. Così:

```
READ(I, J, K, L)
```

è l'equivalente della sequenza delle quattro istruzioni READ mostrate più sopra.

Per la lettura di valori REAL:

```
READ(R)
```

accetterà qualsiasi <costante> REAL come quelle illustrate nel capitolo 5, sezione 6 di questo libro. Prima della battitura di qualsiasi altro carattere, potrete inserire oltre a <spazio> e <RET> anche i caratteri '+', '—' '.' ed 'E' nei punti illustrati dal diagramma sintattico nella figura 5-2. La violazione delle regole sintattiche descritte da questo diagramma farà terminare il programma in modo abonorme. In tutti i punti dove la sintassi permette un'uscita dal diagramma, quando non c'è più la necessità di altri caratteri, sarà lecito inserire qualsiasi carattere diverso da quelli illustrati dalla sintassi per indicare la fine del numero. Cioè:

```
READ(R,Q)
```

accetterà:

```
—1.0E3 0.123
```

assegnando —1.0E+3 (—1000.0) a R, e + 1.23E—1 a Q.

È lecito *mischiare* delle richieste di dati READ dei diversi <tipi> discussi in questo capitolo ma ciò deve essere fatto con una certa attenzione. Supponete ad esempio che il vostro programma contenga le istruzioni:

```
READ(I, CH, S)
```

e che voi inseriate ciò che segue:

```
5064Luigi,Aldo  123—22—6720<RET>
```

dove, come sempre "<RET>" significa una pressione del tasto di ritorno. Ad I sarà assegnato il valore intero 5064, a CH sarà assegnato 'L', e ad S sarà assegnata la stringa 'Luigi,Aldo 123—22—6720', dove LENGTH(S) ritornerà il valore 23 che include anche i 3 spazi bianchi che seguono 'Aldo'.

Se avessimo voluto dare un ordine diverso ai diversi <tipi> di variabili qui considerate facendo precedere S ad una delle altre variabili contenute nell'istruzione READ, sarebbe stato necessario battere la <stringa> completa su di una linea, terminarla con <RET> e continuare poi con gli altri dati sulla linea successiva.

L'istruzione:

```
READLN
```

fa sì che il programma ignori qualsiasi immissione fino a quando non venga battuto

< RET >. READLN significa "READ a Line". Le azioni di READ e READLN possono essere combinate come nel seguente esempio:

```
READLN(I, J)
```

che è l'equivalente di:

```
READ(I);
READ(J);
READLN
```

In altre parole, l'attesa della battitura del tasto < RET > avviene *dopo* che i dati sono stati assegnati a tutte le variabili nominate quali parametri. Così:

```
READLN(I, J, K, L)
```

con l'immissione di:

```
12 34 5< RET >
678< RET >
```

causerà il completamento dell'istruzione READLN solo quando il *secondo* < RET > è stato premuto, in quanto il primo < RET > verrà ignorato ed interpretato come se fosse uno < spazio > nella ricerca del valore di L (678).

5. EOF e EOLN

"EOF" significa "End - Of - File". *"EOLN"* significa "End - Of - Line". Entrambe rappresentano il nome di funzioni Booleane interne che possono essere utili nella scrittura di programmi interattivi. Entrambe queste funzioni sono concettualmente più adattabili all'ambito dei lotti di schede perforate, o all'uso di altri dispositivi di immissione che memorizzano effettivamente flussi di dati divisi in record.

Quando il vostro programma viene lanciato, EOF sarà inizialmente FALSE. Resterà FALSE fino a quando non terminerete un'istruzione READ o READLN con la pressione di < EXT > (o < ENTER >), o con la battitura di 'C' *mentre mantenete premuto il tasto < CTRL >, cioè il tasto "controllo".* Questa particolarità può essere usata in qualsiasi dei seguenti modi:

```
WHILE NOT EOF DO          REPEAT
```

```
BEGIN
  . . .
  qualsiasi istruzione        qualsiasi istruzione
  . . .                       . . .
END                         UNTIL EOF
```

<CTRL–C> è equivalente al tasto <EXT> (End of Text), chiamato alle volte <ENTER>. Un'iterazione o l'altra terminerà alla prima occasione dopo la battitura di <EXT> mentre è in esecuzione un READ o READLN all'interno del ciclo. Notate che nel caso il ciclo contenesse più di un'istruzione READ o altre istruzioni influenzate da READ, dovrete includere un'istruzione che inizi così:

```
IF NOT EOF THEN...
```

per prevenire elaborazioni indesiderate di istruzioni all'interno del ciclo dopo la battitura di <EXT> o <CTRL–C>.

Concettualmente EOLN è simile a EOF, ad eccezione che EOLN è fissato TRUE con la battitura di <RET> al termine di una istruzione READ. *Il READ successivo riporterà EOLN a FALSE.*

Considerate il seguente esempio:

```
READ(I);
IF EOLN THEN istruzione–1
        ELSE istruzione–2
```

Se voi battete:

```
123<RET>
```

allora l'istruzione–1 sarà eseguita. Però se voi battete:

```
123 <RET>
```

allora verrà eseguita l'istruzione–2 poiché lo <spazio> battuto prima di <RET> farà terminare READ. L'operazione di READ (<variabile reale>) avrà caratteristiche simili.

READ(CH) sarà soddisfatto dalla sola battitura di <RET>, *e fisserà CH=' '*, ma fisserà anche EOLN a TRUE. Il successivo READ(CH) riporterà EOLN di nuovo a FALSE!

READ(S) permette la battitura di <RET> in qualsiasi momento come è stato illu-

strato più sopra. Immediatamente dopo il termine di READ(S) EOLN dovrebbe sempre essere TRUE.

6. I flussi INPUT e OUTPUT

Il sistema PASCAL prevede dei modi di lettura (READing) di informazioni da un dispositivo diverso da una tastiera, e dei modi di scrittura (WRITEing) di informazioni su un dispositivo diverso dal video. Finora vi abbiamo evitato le complicazioni di linguaggio associate a queste possibilità. Mentre lavorate con gli esercizi di questo libro potrete ignorare, in molti casi, queste complicazioni. Esistono tuttavia due azioni che si possono dimostrare molto convenienti e che richiedono un minimo di conoscenza dei *"flussi"* standard INPUT e OUTPUT.

Il nostro sistema PASCAL, così come molti altri, presume che le informazioni da introdurre nel programma con l'uso dell'istruzione READ, provengano da un dispositivo di immissione "standard". Si presume inoltre che tutte le istruzioni WRITE facciano riferimento al dispositivo di emissione standard. È possibile cambiare sia READ che WRITE in modo che facciano riferimento ad un dispositivo differente dandone il nome del flusso, per esempio:

```
READ(<nomeflusso>, I, J, K)
```

e

```
WRITE(<nomeflusso>, R, L, CH)
```

Fin qui avevamo lasciato vuoto "<nomeflusso>", cosicchè il compilatore assumeva che volevamo il dispositivo di immissione e quello di emissione standard. Comunque, le due seguenti istruzioni sono esattamente equivalenti:

```
READ(INPUT, I, J, R);
READ(I, J, R)
```

Analogamente:

```
WRITE(OUTPUT CH, S);
WRITE(CH, S)'
```

inoltre:

```
EOF(INPUT) equivale a EOF
EOLN(INPUT) equivale a EOLN
```

Lo scopo di questi chiarimenti sta nel fatto che potreste desiderare di usare le procedure interne *"RESET"* o *"PAGE"* che richiedono la dichiarazione esplicita del nome del flusso. Se, dopo aver fissato EOF a TRUE con la pressione del tasto <ETX> o <CTRL–C>, desideraste iniziare la lettura in una parte successiva del programma, ciò potrà essere fatto con:

```
RESET(INPUT)
```

Nel caso desideraste avere il video completamente sgombro così da poter iniziare la visualizzazione di nuove informazioni su un "foglio pulito", lo potrete fare con:

```
PAGE(OUTPUT)
```

Ulteriori discussioni sui flussi è al di fuori dello scopo di questo libro.

7. Il Programma tipo MEDIA

Partendo da questa sezione presenteremo diversi esempi di programmi illustranti l'immissione dei dati. La sola descrizione scritta non sarà sufficiente per farvi capire quello che sta avvenendo mentre il programma gira, per questo vi suggeriamo di implementare ogni programma sul vostro elaboratore. Lanciate poi il programma e cercate di capirne l'azione confrontando attentamente con il programma PASCAL stampato.

Il programma MEDIA viene presentato in due forme per permettere il confronto di due metodi capaci di portare a termine un ciclo che legge valori di dati in una sequenza di lunghezza indefinita. Entrambe le forme calcolano il valore medio corrispondente ad una colonna di interi.

La maggior parte del lavoro è svolto nell'esecuzione ripetuta delle linee comprese fra la 25 e la 34 di MEDIA, e le linee dalla 25 alla 32 di EOLNMED. All'inizio di ogni linea di immissione, il programma *"suggerisce"* che è attesa l'immissione visualizzando il carattere '>'. Il listato vi mostra poi il risultato della battitura di un numero immediatamente dopo la visualizzazione del carattere di suggerimento. L'istruzione READ assegna il numero inserito ad X, che viene poi aggiunto a SOMMA. La variabile che conta gli interi N, è aumentata di 1 per mantenere traccia del numero di valori di cui viene fatta la media.

WRITELN visualizza poi il valore di N, ed il corrente valore di SOMMA. Notate l'uso della funzione DIMEN all'interno dell'elenco parametri di WRITELN. Visualizzando (6–DIMEN(X)) spazi bianchi, le colonne mostranti i valori di N e SOMMA sono allineate verticalmente sullo schermo, indipendentemente dalla lunghezza (entro certi limiti) dell'intero inserito.

In MEDIA il ciclo terminerà quando viene immesso un numero di valore assoluto uguale a 10.000 o superiore. La funzione interna ABS ritorna un valore positivo avente la grandezza dell'espressione Intera o Reale usata quale suo parametro effettivo.

Questo vi illustra uno dei possibili modi di arrestare l'iterazione, che funziona quando sia possibile specificare un valore di dati esterno alla gamma dei normali dati aspettati dal programma per l'elaborazione.

In EOLNMED, il ciclo termina quando viene battuto un numero immediatamente seguito da <RET>. Senza l'uso del flusso *"KEYBOARD"* e dell'associata istruzione di eco "WRITE", la visualizzazione ottenuta dovrebbe differire da quella mostrata per MEDIA in quanto l'ultima linea visualizzante i valori di N e SOMMA risulterebbe interrotta dall'azione di <RET> al momento dell'eco. Questi valori non risulterebbero quindi propriamente allineati nelle colonne. Il flusso KEYBOARD ha caratteristiche simili a INPUT, eccetto che nessuno dei caratteri battuti avrà eco sullo schermo.

```
 1: PROGRAMMA MEDIA;
 2: VAR N,X,SOMMA:INTEGER;
 3:    MED:REAL;
 4:
 5: FUNCTION DIMEN (X:INTEGER):INTEGER;
 6: VAR DM, AX:INTEGER
 7: BEGIN
 8:    AX:=ABS(X);
 9:    IF AX<10 THEN DM:=1
10:    ELSE
11:       IF AX<100 THEN DM:=2
12:       ELSE
13:          IF AX<1000 THEN DM:=3
14:          ELSE
15:             DM:=4;
16:    IF X<0 THEN DM:=DM+1; (*per segno meno*)
17:    DIMEN:=DM
18: END (*DIMEN*);
19:
20: BEGIN (*programma principale*)
21:    WRITELN('MEDIA');
22:    SOMMA:=0;
23:    X:=0;
24:    N:=0;
25:    REPEAT
26:       WRITE('>'); (*suggerimento per immissione*)
27:       READ(X);
28:       IF ABS(X)<1000 THEN
```

```
29:       BEGIN
30:         SOMMA:=SOMMA+X;
31:         N:=N+1;
32:         WRITELN(' ':6-DIMEN(X),N:2, ' ':2,'SOMMA=',SOMMA);
33:       END;
34:     UNTIL ABS(X)>=10000;
35:     MED:=SOMMA/N
36:     WRITELN; WRITELN;
37:     WRITELN('MEDIA=',MED, 'PER ',N,' VOCI');
38: END.
```

```
 1: Visualizzazione associata al programma MEDIA
 2:
 3: MEDIA
 4: >6      1          SOMMA=6
 5: >-8     2          SOMMA=-2
 6: >15     3          SOMMA=13
 7: >20     4          SOMMA=33
 8: >19999
 9:
10: MEDIA=8.25 PER 4 VOCI
```

```
 1: PROGRAMMA EOLNMED;
 2: VAR X,N,SOMMA:INTEGER;
 3:     MED;REAL;
 4:
 5: FUNCTION DIMEN(X:INTEGER):INTEGER;
 6: VAR DM,AX:INTEGER;
 7: BEGIN
 8:     AX:=ABS(X);
 9:     IF AX<10 THEN DM:=1
10:     ELSE
11:       IF AX <100 THEN DM:=2
12:       ELSE
13:         IF AX<1000 THEN DM:=3
14:         ELSE
15:           DM:=4;
16:     IF X<0 THEN DM:=DM+1; (*per il segno -*)
17:     DIMEN:=DM;
18: END (*DIMEN*);
19:
20: BEGIN (*PROGRAMMA PRINCIPALE*)
21:     WRITELN('MEDIA');
```

```
22:    SOMMA:=0;
23:    X:=0;
24:    N:=0;
25:    REPEAT
26:       WRITE('>'); (*suggerimento per immissione*)
27:       READ(KEYBOARD,X);
28:       WRITE(X);
29:       SOMMA:=SOMMA+X;
30:       N:=N+1;
31:       WRITELN(' ':6-DIMEN(X),N:2,' ':2,'SOMMA=',SOMMA);
32:    UNTIL EOLN;
33:    MED:=SOMMA/N;
34:    WRITELN; WRITELN;
35:    WRITELN('MEDIA=',MED, ' PER ',N,' VOCI');
36: END.
```

Ricordatevi che <RET> viene interpretato come uno <spazio> per la lettura di una variabile INTEGER, così da evitare qualsiasi problema di conclusione anomala del programma in linea 27 quando <RET> viene inserito. READ in linea 27 non terminerà fino a quando non venga battuta almeno una cifra, questo per la ragione appena citata.

Una terza alternativa potrebbe essere quella di usare EOF invece di EOLN. La battitura di <ETX> o di <CTRL–C> dopo '>' avrebbe fissato EOF a TRUE, ma avrebbe anche assegnato un valore indefinito a X nell'istruzione READ. Per evitare problemi con tentativi di sommare valori indefiniti di X a SOMMA, sarebbe necessario controllare le linee da 29 a 31 di EOLNMED (modificato ad usare EOF invece di EOLN) all'interno di un'istruzione IF NOT EOF THEN... Esenzialmente la stessa tecnica viene usata in diversi degli esempi dei programmi successivi.

8. Programma tipo FARESTO

Lo scopo principale di questo programma è di illustrare l'uso di parecchi valori di dati immessi su un'unica linea di immissione. Il programma simula, in modo schematico, quello che avviene alla cassa di un grande magazzino o al momento del pagamento del conto in un ristorante. Generalmente vi presenterete con spese, sommanti a diverse migliaia di lire, più alcune frazioni di mille lire espresse in pezzi da cento lire etc. Spesso vi troverete a pagare con soli biglietti di carta, consegnando alla cassiera uno o più "biglietti" da mille, cinquemila, diecimila lire. Questo programma accetta sia il pagamento espresso con un numero intero di lire, che con la denominazione dei biglietti espressi solamente in migliaia di lire.

Il programma vi richiede dapprima l'immissione dei pagamenti, poi l'immissione dei soldi sotto forma di serie di costanti intere separate da spazi bianchi, tutto su un'unica linea. Quale esempio guardate le linee 4, 7 e 10 nella pagina a parte mostrante le linee visualizzate da questo programma. Su queste linee l'elaboratore ha visualizzato "CONTO"; l'ammontare in lire con il relativo simbolo e inoltre la legenda "SOLDI". L'utente batte gli altri numeri, terminando la linea con <RET> dopo aver inserito la denominazione dell'ultimo biglietto di lire. Il programma visualizza quindi una seconda linea mostrante il numero di biglietti da mille dovuti al cliente quale resto.

```
 1: PROGRAMMA FARESTO;
 2: VAR CONTO,SOLDI,PAGATO,RESTO:INTEGER;
 3:
 4: PROCEDURE VISUAL;
 5: VAR DIECIMILA,DUEMILA,MILLE,CINQUECENTO,CENTO:INTEGER;
 6: BEGIN
 7:   DIECIMILA:=0; DUEMILA:=0; MILLE:=0; CENTO:=0;
 8:   WRITE('RESTO:');
 9:   RESTO:=PAGATO-CONTO;
10:   WHILE RESTO>=100 DO
11:     BEGIN RESTO:=RESTO-100; DIECIMILA:=DIECIMILA+1; END;
12:   WHILE RESTO>=20 DO
13:     BEGIN RESTO:=RESTO-20; DUEMILA:=DUEMILA+1; END;
14:   WHILE RESTO>=10 DO
15:     BEGIN RESTO:=RESTO-10; MILLE:=MILLE+1; END;
16:   IF RESTO>=5 THEN
17:     BEGIN RESTO:=RESTO-5; CINQUECENTO:=1; END
18:     ELSE
19:       CINQUECENTO:=0;
20:   WHILE RESTO >=1 DO
21:     BEGIN RESTO:=RESTO-1; CENTO:=CENTO+1; END;
22:   WRITELN(' DIECIMILA:',DIECIMILA, 'DUEMILA:',DUEMILA,
23:             ; MILLE:',MILLE, 'CINQUECENTO:', CINQUECENTO,
24:             ;CENTO:'CENTO);
25:   WRITELN;
26: END (*VISUAL*);
27:
28: BEGIN (*PROGRAMMA PRINCIPALE*)
29:   WRITELN('FARESTO');
30:   REPEAT
31:     WRITE('CONTO:');
32:     READ(CONTO);
33:     IF NOT EOF THEN
```

```
34:       BEGIN
35:         WRITE(' L',(CONTO DIV 100),'.',
36:         (CONTO MOD 100) );
37:         PAGATO:=0;
38:         WRITE(' SOLDI:');
39:         WHILE NOT EOLN DO
40:         BEGIN
41:           READ(SOLDI); (*dimensione di ogni biglietto*)
42:           PAGATO:=PAGATO+SOLDI*100; (*in centinaia di lire*)
43:         END;
44:         VISUAL;
45:       END;
46:     UNTIL EOF;
47: END.
```

```
 1: Visualizzazione associata con il programma FARESTO
 2:
 3: FARESTO
 4: CONTO:23700 LIT23700 SOLDI:50.000
 5: RESTO: DIECIMILA:2; DUEMILA:3, MILLE:0, CINQUECENTO:0,
    CENTO:3
 6:
 7: CONTO:23700 LIT23700 SOLDI:24.000
 8: RESTO: DIECIMILA:0, DUEMILA:3, MILLE:0, CINQUECENTO:0,
    CENTO:3
 9:
10: CONTO 59100 LIT 59100 SOLDI:100.000
11: RESTO: DIECIMILA:4, DUEMILA:0, MILLE:0, CINQUECENTO:1,
    CENTO:4
12:
13: RESTO:
```

Il programma può essere compreso con l'aiuto dei diagrammi strutturali nelle figure 7-1a e 7-1b. Dopo la verifica, in 1, che è realmente il programma FARESTO che sta girando il ciclo del programma principale è interno a 2. Viene visualizzato dapprima il suggerimento per "CONTO": ed il programma aspetta la battitura di un intero. Essendo stato programmato che il ciclo verrà terminato dall'uso del meccanismo <ETX> per fare EOF TRUE, è necessario assicurarsi che il messaggio di suggerimento iniziale non sia seguito immediatamente dal cambiamento in EOF. Se EOF non è ancora TRUE, visualizza il conto. Il programma pone quindi PAGATO a zero, e visualizza un suggerimento per i pezzi da diecimila con cui si paga (il tutto in 2b2). L'iterazione 2b3 legge quindi l'ammontare e lo somma a PAGATO usando come base le 100 lire; questa iterazione termina quando è battuto <RET> immediatamente dopo l'ultima cifra intera dell'elenco. È infine chiamata la procedura VISUAL che genera la linea con il dettaglio del resto.

La parte (b) del diagramma mostra un metodo per calcolare il numero di pezzi da diecimila, duemila, mille e cento che devono essere resi come resto. L'algoritmo mostrato è molto simile a quello che dovrebbe usare un impiegato nel contare il resto. L'idea base è quella di rendere biglietti o monete della più alta pezzatura disponibile che siano però come totale inferiori al resto che deve ancora essere dato.

Su grossi elaboratori l'iterazione con WHILE non è necessariamente il miglior modo per eseguire l'elaborazione, in quanto sarebbe stato possibile sostituire l'iterazione con operazioni DIV e MOD. Certi mini e microelaboratori, come forse quello che state usando mentre studiate su questo testo, non hanno istruzioni hardware per eseguire direttamente la divisione o la moltiplicazione; queste operazioni richiedono perciò l'uso di procedure interne costituite da una sequenza di addizioni o sottrazioni con un metodo uguale a quello che voi usereste con carta e penna. Questo particolare programma è tanto corto che non dovreste accorgervi della differenza nei tempi di esecuzione di grosse macchine rispetto a piccole, usando questi due metodi:

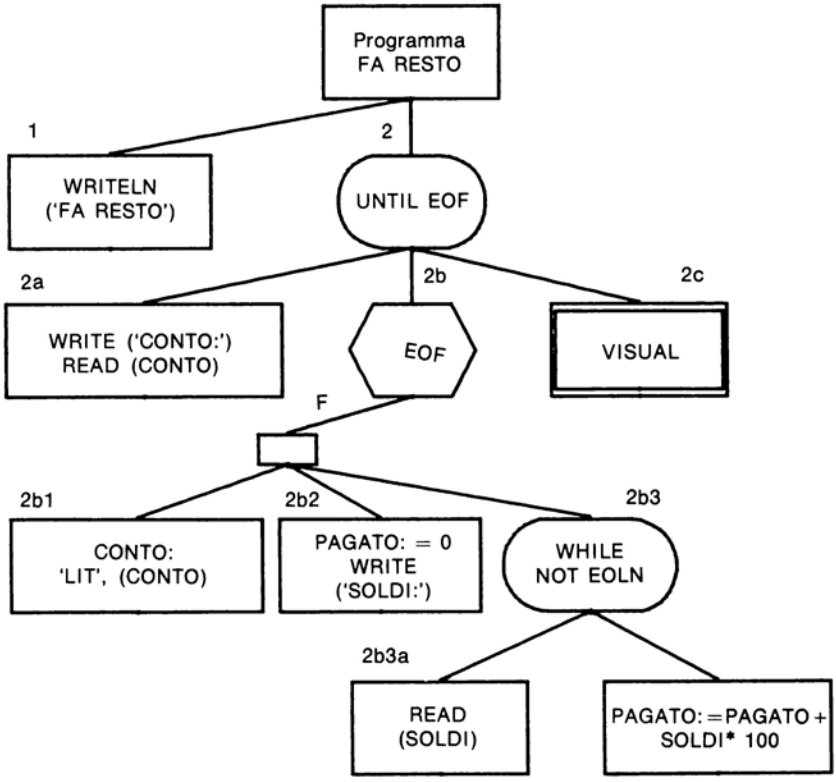


Figura 7-1a

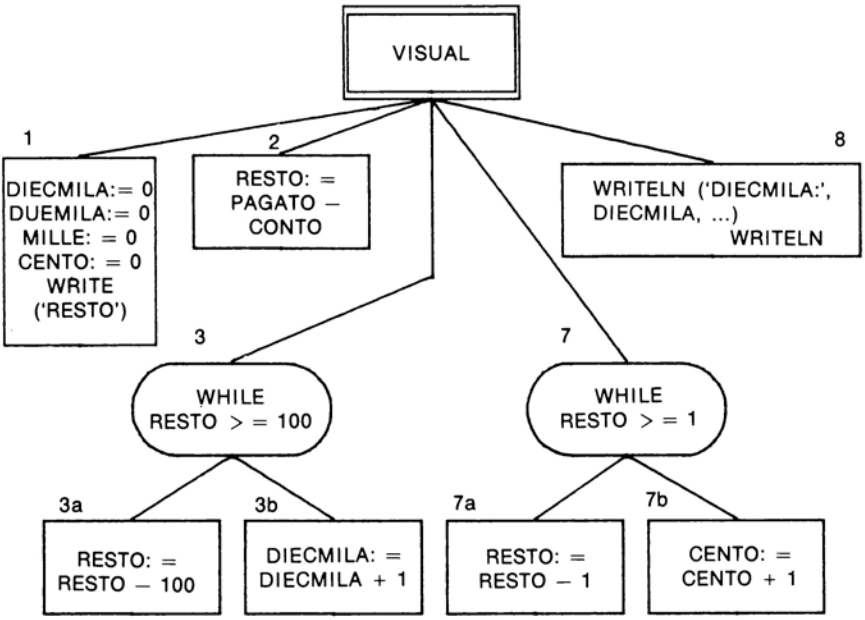


Figura 7-1b

9. Programma tipo NODISTURBI

Questo programma mostra parte del processo spesso usato per preparare indici e titoli di articoli pubblicati nella letteratura scientifica. Lo stesso processo, chiamato scrittura di indice per mezzo di "Key Word In Context" (KWIC), è usato anche da storici e linguisti nello studio della letteratura non scientifica. L'idea dell'indice KWIC è di stampare ogni titolo, spostato a destra o a sinistra, in modo tale che la parola chiave (KEYWORD) termini allineata al centro della riga. Ogni titolo è stampato una volta per ogni parola chiave, ogni volta spostato a destra o sinistra di una quantità diversa in modo che la parola chiave selezionata compaia sempre al centro della pagina. I titoli spostati per i diversi documenti sono quindi ordinati, cosicchè possono essere stampati in ordine alfabetico in accordo con le parole chiave scelte. Questo permette ad un lettore di scandire l'elenco come un indice, mettendo ogni parola chiave nel proprio contesto (titolo).

```
1: PROGRAMMA NODISTURBI;
2: VAR RUMORE,S:STRING;
3:     P,I:INTEGER;
```

```
 4: BEGIN
 5:   WRITELN('NODISTURBI');
 6:   REPEAT
 7:     WRITE('>');
 8:     READLN(S);
 9:     IF NOT EOF THEN
10:     BEGIN
11:       FOR I:=1 TO 7 DO
12:       BEGIN
13:         CASE I OF
14:           1: RUMORE:='IL ';
15:           2: RUMORE:='LA ';
16:           3: RUMORE:='LE ';
17:           4: RUMORE:='GLI ';
18:           5: RUMORE:='IN ';
19:           6: RUMORE:='DEI ';
20:           7: RUMORE:='COEN ';
21:         END (*CASE*);
22:         P:=POS(RUMORE,S);
23:         IF P>P>0 THEN DELETE(S,P,LENGTH(RUMORE));
24:       END;
25:       WRITELN(S);
26:       WRITELN;
27:     END (*SE NON EOF*);
28:   UNTIL EOF;
29: END.
```

Se l'indice KWIC deve essere utile, il programma non deve disturbare l'indice con parole "rumore" che se è vero che contribuiscono alla leggibilità non portano però nessuna informazione reale. Ad esempio, nessuna delle parole brevi che si vedono a destra delle istruzioni di assegnamento da riga 14 a riga 20 del programma è di alcun interesse nel costruire l'indice. Perciò un programma che deve preparare un indice KWIC deve poter riconoscere le parole di disturbo ed ignorarle nel resto del lavoro.

Ecco come potrebbe apparire un indice KWIC veramente corto fatto con alcuni titoli campioni con questo programma:

```
                    parole chiave

                          ANALISI   DEI CROMOSOMI CON L'ELABORATORE
                          ANALISI   DEI SISTEMI DI TRASPORTO URBANO
                                    ANALISI DEI CROMOSOMI CON L'ELABO
ANALISI DEI CROMOSOMI CON L'ELABORATORE
```

IL PIÚ VELOCE ELABORATORE
L'UTILIZZO DEGLI ELABORATORI NELL'ISTRUZIONE
'UTILIZZO DEGLI ELABORATORI' NELL'ISTRUZIONE

Per mantenere la stampa compatta, e per rimanere nel numero di colonne prescritte, alcuni titoli sono troncati nella coda o nella testa. Solitamente la stampa dovrebbe essere più larga di quella esemplificata qui, e dovrebbe includere un codice di riferimento che indica dove possono essere trovate le informazioni dettagliate a cui fa riferimento quel titolo.

Dopo aver visualizzato il suo nome (linea 5 del programma), il programma presenta un '>'come abilitazione all'immissione. Attende quindi che venga battuta una stringa. La stringa appare dapprima così come è immessa e poi sulla riga sottostante dello schermo senza le parole "rumore". Questo programma è stato scritto solo per esemplificare l'operazione di rimozione delle parole disturbanti, e non fa nessun altro lavoro per la costruzione dell'indice KWIC. Se fate un programma per la scrittura degli indici secondo il metodo KWIC dovrete anche scrivere una procedura che tolga i disturbi. Il programma NODISTURBI può essere usato nel corso del progetto della procedura come uno strumento di test. Una volta verificato che è corretto il programma potrebbe essere convertito in una procedura ed essere aggiunto al programma KWIC in fase di sviluppo.

Questo programma è un esempio del blocco di un'iterazione tramite la battitura di <ETX> o <CTRL–C> come primo carattere dopo il carattere di suggerimento. Ecco un esempio di come può apparire lo schermo mentre usate il programma:

```
NODISTURBI
>ANALISI DEI TRASPORTI URBANI
ANALISI TRASPORTI URBANI

>ANALISI DEI CROMOSOMI CON ELABORATORE
ANALISI CROMOSOMI ELABORATORE

>USO DEI VIDEI CON ELABORATORE
USO VIDEI ELABORATORE
```

10. Programma tipo DEVOCALIZZA

Lo scopo principale di questo programma tipo è di esemplificare la lettura carattere per carattere, da un blocco di dati immessi, usando READ. Un secondo scopo è quello di far vedere i tipi di alterazione ad un testo che un linguista può operare per

capire come le persone estraggono informazioni dalla lettura di un testo. In questo caso il programma toglie tutte le vocali dai dati immessi. Avendo letto una riga nella forma italiana, non dovreste avere eccessive difficoltà nel "leggere" il testo senza vocali: Potreste far questo senza aver letto la versione italiana normale?

```
 1: PROGRAMMA DEVOCALIZZA;
 2: VAR CH:CHAR;
 3:   S,TEMP:STRING;
 4: BEGIN
 5:   WRITELN('DEVOCALIZZA');
 6:   WRITE('>');
 7:   READ(CH);
 8:   WHILE NOT EOF DO
 9:   BEGIN
10:     S:='';
11:     TEMP:=' '; (*un carattere*)
12:     WHILE NOT EOLN DO
13:       BEGIN
14:         IF (CH<>'A') AND (CH<>'E') AND (CH<>'I')
15:       AND (CH<>'0') AND (CH<'U') THEN
16:         BEGIN
17:         TEMP[1]:=CH;
18:         S:=CONCAT(S,TEMP);
19:         END;
20:       READ(CH);
21:     END;
22:     WRITELN(S);
23:     WRITELN;
24:     WRITE('>');
25:     READ(CH); (*annulla EOLN*)
26:   END (*While not eof*);
27: END.
```

```
 1: Visualizzazione associata a DEVOCALIZZA
 2:
 3: >PARIEMI CHE 'L SUO VISO ARDESSE TUTTO,
 4: PRM CH 'L S VS RDSS TTT,
 5:
 6: >E LI OCCHI AVEA DI LETIZIA SÌ PIENI,
 7: L CCH V D LTZ S PN,
 8:
 9: >CHE PASSARMEN CONVIEN SENZA COSTRUTTO.
10: CH PSSRMN CNVN SNZ CSTRTT.
```

Le linee visualizzate da questo programma (1÷10) sono prese dal Paradiso, canto XXIII, versetti 22-24.

In questo programma, all'interno dell'iterazione controllata dalle funzione EOLN e che parte da linea 12 viene letto un carattere alla volta. Le consonanti sono concatenate in una variabile stringa S fino a che è battuto < RET >, il quale fa sì che EOLN sia TRUE fino a che viene eseguita una nuova READ. I caratteri immessi appaiono sullo schermo così come sono battuti. In riga 22 il contenuto di S è visualizzato sulla riga seguente dello schermo. Alla fine è battuto < ETX > che rende vero EOF e fa terminare il ciclo esterno.

11. Programma tipo-VERIFICADATA

Questo programma è presentato per illustrare la tecnica di *"convalida"* dei dati immessi. Nei programmi gestionali di taglia appena ragguardevole si è soliti fare delle verifiche sui valori dei dati immessi per essere sicuri che siano validi. Queste verifiche sono generalmente necessarie tutte le volte che i dati da immettere sono preparati da persone, siano essi impiegati a tempo pieno o persone che lo fanno una tantum. I non addetti ai lavori frequentemente fanno degli errori perché sbagliano nel leggere o capire le istruzioni per riempire le maschere. Ma anche il miglior impiegato che lavora all'immissione dati fa degli errori su circa 1% dei documenti che prepara per l'elaboratore usando perforatori di schede o macchine simili.

In parecchi casi, l'elaborazione di dati errati può portare svariati problemi alle persone interessate dall'elaborazione e per le quali si immettono i dati. Avrete probabilmente sentito parlare di persone alle quali si rifiuta di pagare un assegno in quanto ci sono dei record sbagliati nell'archivio clienti della banca. Allo stesso modo i grandi magazzini emettono fatture sbagliate, le università rifiutano studenti autorizzati, persone innocenti sono accusate di crimini, e così via. Spesso questi errori succedono perché chi scrisse il programma non lo rese sufficientemente intelligente da trovare degli errori banali nei dati preparati dagli impiegati per l'immissione.

```
1: PROGRAMMA VERIFICADATA;
2: VAR CH:CHAR;
3:    MESE,GIORNO,ANNO:INTEGER;
4:
5: PROCEDURE RICHIAMO(S:STRING; X:INTEGER);
6: BEGIN
7:    WRITELN(' ***',S,' ERRATO:',X,CHR(7(*SUONO*)));
8: END;
9:
```

```
10: FUNCTION DENTROBARRA:BOOLEAN;
11: BEGIN
12:   READ(CH);
13:   DENTROBARRA:=(CH='/');
14:   IF CH< >'/' THEN
15:     WRITELN(' *** CI SI ASPETTA UNA BARRA:'CH,
    CHR(7(*SUONO*)));
16:
17: END (*DENTROBARRA*);
18:
19: PROCEDURE METTIMESE(M:INTEGER);
20: VAR S:STRING;
21: BEGIN
22:   CASE M o F
23:       1: S:='GENNAIO';
24:       2: S:='FEBBRAIO';
25:       3: S:='MARZO';
26:       4: S:='APRILE';
27:       5: S:='MAGGIO';
28:       6: S:='GIUGNO';
29:       7: S:='LUGLIO';
30:       8: S:='AGOSTO';
31:       9: S:='SETTEMBRE';
32:      10: S:='OTTOBRE';
33:      11: S:='NOVEMBRE';
34:      12: S:='DICEMBRE';
35:   END (*CASE*);
36:   WRITE(S);
37: END (*METTIMESE*);
38:
39: BEGIN (*PROGRAMMA PRINCIPALE*)
40:   WRITELN('VERIFICADATA');
41:   CH:=' ';
42:   WHILE CH < > '#' DO
43:   BEGIN
44:     WRITE('MESE/GIORNO/ANNO:');
45:     READ(MESE);
46:     IF (MESE<=0) OR (MESE)>13 THEN
47:       RICHIAMO('MESE',MESE) ELSE
48:       BEGIN
49:         IF DENTROBARRA THEN
50:         BEGIN
51:         READ(GIORNO);
```

```
52:           IF (GIORNO<=0) OR (GIORNO>=32) THEN
53:             RICHIAMO('GIORNO',GIORNO) ELSE
54:             BEGIN
55:                 IF DENTROBARRA THEN
56:                 BEGIN
57:                   READ(ANNO);
58:                   IF (ANNO<=10) OR (ANNO>=69) THEN
59:                   RICHIAMO('ANNO',ANNO); ELSE
60:                   BEGIN
61:                     METTIMESE(MESE);
62:                     WRITELN(' ',GIORNO,', ',ANNO+1900);
63:                   END (*anno*);
64:               END (*seconda barra*)
65:             END (*giorno*)
66:           END (*prima barra*)
67:         END (*Mese*);
68:       REPEAT
69:         WRITELN;
70:         WRITE('>');
71:         READ(CH); (*è possibile '#' indicante fine flusso*)
72:         IF (CH<>' ') AND (CH<>'#') THEN
73:           WRITE('--> SPAZIO O '#' ATTESI',
74:                   CHR(7(*BEL*)));
75:       UNTIL (CH=' ') OR (CH='#');
76:     END (*CH<>'#'*);
77: END.
```

```
 1: Visualizzazione associata al programma VERIFICADATA
 2:
 3: VERIFICADATA
 4: MESE/GIORNO/ANNO:6/15/57 GIUGNO 15,1957
 5:
 6: >MESE/GIORNO/ANNO:6/15/1957 *** ANNO ERRATO:1957
 7:
 8: >MESE/GIORNO/ANNO:15/ *** MESE ERRATO:15
 9:
10: >MESE/GIORNO/ANNO:12/21/10 *** ANNO ERRATO:10
11:
12: >MESE/GIORNO/ANNO:9/22/75 *** ANNO ERRATO:75
13:
14: >MESE/GIORNO/ANNO:6/37 *** GIORNO ERRATO:37
15:
16: >MESE/GIORNO/ANNO:6- *** CI SI ASPETTA UNA BARRA:-
```

```
17:
18: >6 ---->SPAZIO o '#' ATTESI
19:
20: >MESE/GIORNO/ANNO:12/31/64#DICEMBRE 31,1964
```

Questo programma tipo è stato scritto per mostrare le verifiche tipiche di validità che si possono voler fare su dati consistenti in date. La forma generale che ci si aspetta da una data è la seguente:

< mese >/< giorno >/< anno >

dove < mese >, < giorno >, < anno > sono interi espressi con una cifra o due cifre. Nel nostro caso la verifica è per date di nascita ragionevoli per studenti che si immatricolano nel 1981. Riteniamo che un ragazzo nato dopo il 1969 sia troppo giovane e che una persona nata prima del 1910 sia troppo anziana. Naturalmente anche qualunque data tra questi due limiti può essere errata, ma questo errore andrebbe trovato con tutt'altro metodo che una semplice verifica sul fatto che un anno sia "ragionevole".

Le verifiche su < mese > e < giorno > sono più ovvie. Il programma, dal momento che vogliamo mantenere la logica semplice, non tien conto del fatto che i giorni dei mesi sono variabili. Ogni variazione rispetto al formato sarà giudicata un dato errato, anche se una persona può capire la variazione senza difficoltà.

11a. Rilevazione di fine flusso senza EOF

Solo per amor di chiarimento, scegliamo di supporre che la funzione EOF non era disponibile per controllare le iterazioni del ciclo principale di questo programma (da linea 42 a linea 76). Sebbene PASCAL mette a disposizione una funzione interna EOF, certi sistemi o linguaggi non lo fanno. In tali casi, in forme più o meno simili, è usato il metodo mostrato in questo programma.

L'idea è di alzare una *"bandiera"*, costituita da un qualche dato conosciuto che non può logicamente essere frammischiato ai valori dei dati che il programma deve elaborare. In questo caso abbiamo scelto il carattere '#' come *"carattere di fuga"*. Assumiamo inoltre che nel programma venga sempre immesso almeno un dato. Dopo aver letto ogni data, ed averne verificata la validità, il programma visualizza un carattere di suggerimento '>' e quindi controlla se è stato battuto '#'. Se per terminare l'elaborazione di una data è battuto < RET >, READ in linea 71 assegnerà uno spazio bianco a CH, poichè il sistema deve "guardare avanti" quando legge il numero precedente per trovare un carattere non-cifra. Ricordate che il programma riceve il carattere spazio quando è battuto < RET >.

In linea 16 dello schermo si vede che è stato battuto '—' invece di '/'. La funzione DENTROBARRA attiva una segnalazione acustica per richiamare l'attenzione dell'operatore sul fatto che è stato commesso un errore. Il tentativo di introdurre una cifra all'inizio della riga 18 provoca un altro messaggio di errore dalle righe 73 e 74 del programma. Il controllo che ci sia uno spazio oppure '#' a questo punto assicura che l'operatore non si lascerà sfuggire la mancanza della barra nella data, e che ricomincerà a battere la data dall'inizio. Sulla linea 20 dello schermo si vede che è stato battuto un unico spazio, permettendo così al programma di tornare al ciclo principale dove è di nuovo visualizzato: "MESE/GIORNO/ANNO". Infine il carattere '#' è usato nella riga 20 dello schermo per terminare il programma alla fine dell'ultima data. Questa data è verificata prima che il programma raggiunga la fine del ciclo principale.

Vi suggeriamo di provare questo programma sul vostro elaboratore. Non sono state fatte tutte le possibili verifiche. Inoltre il programma, così come è stato scritto, non può far fronte ad un certo numero di errori che spesso le persone fanno e che provocano una fine anormale del programma. Se per esempio il primo carattere battuto quando ci si aspetta un intero non è una cifra, il programma termina in modo anormale. Se il vostro sistema PASCAL è stato programmato per gestire questa possibilità, è meno generale e più "testone" di quanto sia desiderabile (in questo caso). Vi è tuttavia possibile scrivere uno scansare degli interi più "intelligente" che permette altri errori senza che il programma venga bloccato.

11b. Struttura del programma

Questo programma controlla la validità di ogni parte di un elemento di informazione partendo da sinistra. Se la parte analizzata non è valida, viene visualizzato un messaggio d'errore ed azionato il "segnale acustico" di richiamo mediante il carattere di controllo < BEL >, il cui valore decimale è 7. Non essendo possibile mettere direttamente il carattere < BEL > in una < costante stringa > quotata, l'istruzione WRITELN di linea 7 converte il valore decimale nel corrispondente carattere, usando la procedura CHR per la conversione di tipo. Dopo aver trovato un errore in una data, viene abbandonato ogni ulteriore esame su di essa e si attende che l'operatore batta un'altra data.

La figura 7-2 mostra questo programma sotto la forma di un diagramma di struttura. Come potete notare il programma è troppo complesso perché il diagramma di struttura possa occupare soltanto una pagina. In questo caso abbiamo lasciato che il programma principale fosse un po' più grande del massimo di circa 25 righe che di solito cerchiamo di rispettare in ogni < blocco >. Questo permette di controllare la validità di ogni parte di un elemento di informazione in un'unica istruzione IF nidifica-

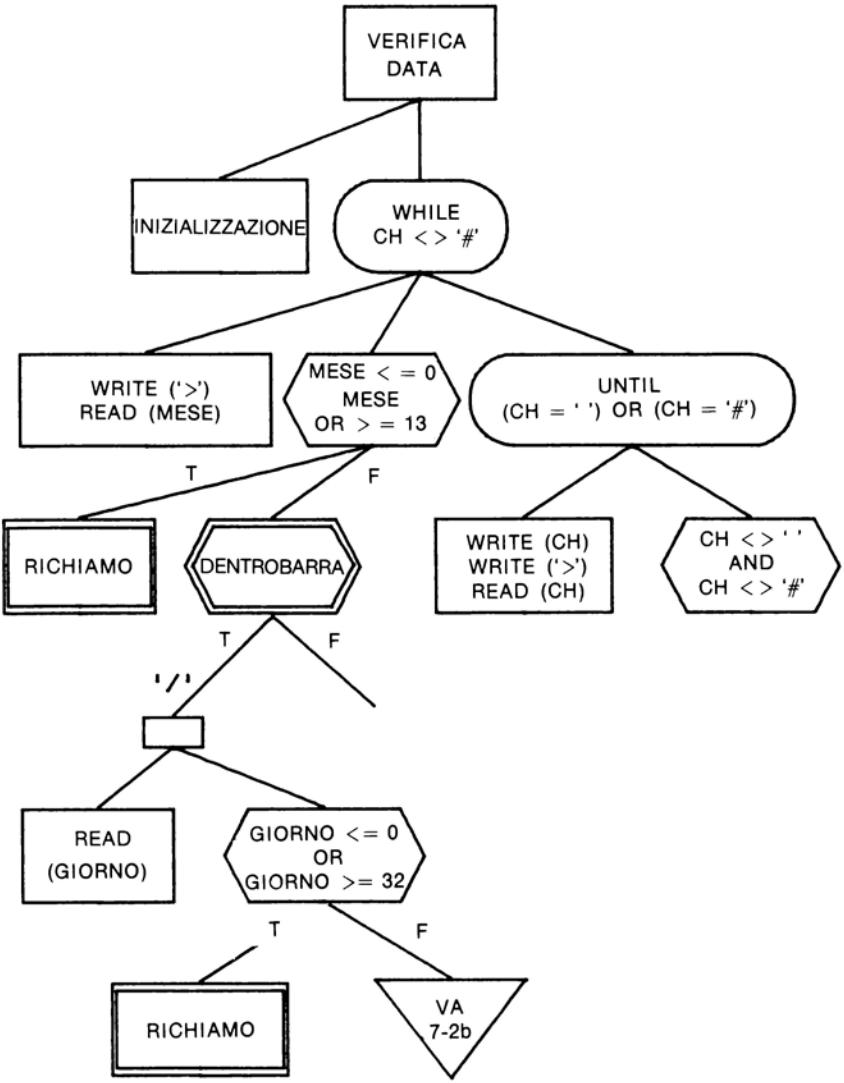

VERIFICA DATA
INIZIALIZZAZIONE
WHILE CH < > '#'
WRITE ('>') READ (MESE)
MESE < = 0 MESE OR > = 13
UNTIL (CH = ' ') OR (CH = '#')
T
F
RICHIAMO
DENTROBARRA
WRITE (CH) WRITE ('>') READ (CH)
CH < > ' ' AND CH < > '#'
T
F
'/'
READ (GIORNO)
GIORNO <= 0 OR GIORNO >= 32
T
F
RICHIAMO
VA 7-2b

Figura 7-2a

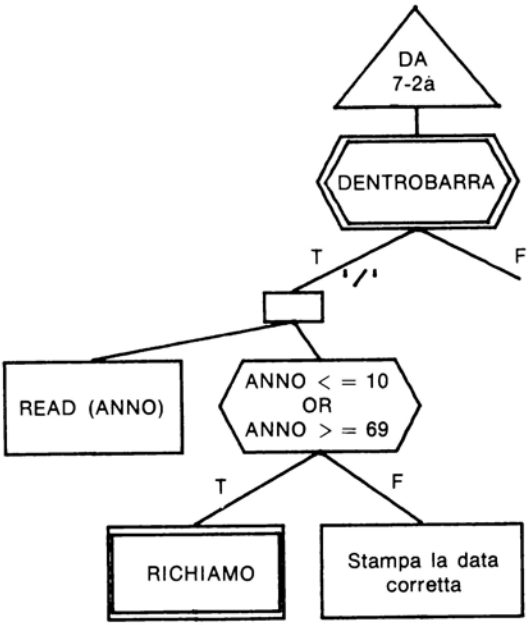


Figura 7-2b

ta. Si può discutere sul fatto che il programma sia più leggibile in questo modo o dichiarando in una procedura separata la parte più interna dell'istruzione IF nidificata. Come vi abbiamo mostrato, la struttura del programma è molto più facile da seguire quando si usa l'indentazione per mostrare il livello di ramificazione nella struttura. Anche i commenti che seguono ogni END aiutano notevolmente nell'evitare errori.

ESERCIZIO 7.1:
Le seguenti sono le specifiche di un programma che calcola il punteggio medio ottenuto dai giocatori in uno sport tipo il bowling. Ogni giocatore può guadagnare fino a 10 punti in una stagione, ma non tutti i giocatori guadagneranno punti in ogni partita. (Modificheremo un poco le regole di questo gioco in modo che il programma possa essere utilizzato per un largo numero di sport).

Il programma deve accettare, per ogni riga di input, il nome di un giocatore cui farà seguito una sequenza di punteggi, fino a 10, corrispondenti alle partite individuali. In uscita dovrebbe essere visualizzata una linea per ogni riga immessa, e tale linea dovrebbe contenere

il nome del giocatore, il numero di partite effettuate e il punteggio medio in quelle partite. Una delle ragioni che possono spiegare la ripetizione di alcune informazioni sulla riga visualizzata è quella di fornire un controllo ridondante sulla correttezza dei dati ricevuti dal programma.

Usate 10 linee per l'ingresso contenenti dati di controllo da voi costruiti. Il formato di tali linee dovrebbe essere simile al seguente:

```
Cecconi, Gianni 150 115 175 140 112 145 160 148 203
Pesarini, Franca 125 148 135 120 110 190 115 140
```

Questo esercizio consiste nel progettare un programma che esegua le azioni, precedentemente descritte, nel preparare un diagramma di struttura che mostri, ad un livello di dettaglio ragionevole, come il programma dovrebbe lavorare, e nel mettere a punto tale programma sull'elaboratore usando dati di controllo.

ESERCIZIO 7.2:
Scrivete un programma PASCAL che legga dati, riga dopo riga, da una tabella simile a quella mostrata più sotto. Dopo aver letto tutte le righe della tabella, il programma dovrebbe visualizzare il numero della riga in cui la somma dei valori risulti massima, il valore di tale somma, e il contenuto della linea trovata in questo modo.

Dati campione:

```
91  46  55
43  59  83
64  47  45
94  25  91
51  24  96
```

Il programma dovrebbe poter gestire un numero variabile di linee in ingresso. L'uscita dovrebbe avere approssimativamente il seguente formato:

```
SOMMA MASSIMA: 210 NELLA RIGA NUMERO: 4
CONTENUTO: 94 25 91
```

ESERCIZIO 7.3:
Scrivete e mettete a punto un programma PASCAL che legga testi italiani da una tastiera, contando il numero di volte che capita ciascu-

na delle cinque vocali ('A', 'E', 'I', 'O' o 'U'). Il programma dovrebbe poter gestire un numero variabile di linee in ingresso. Prendete il testo di questo esercizio come dati di prova. Potete semplificare il problema battendo soltanto lettere maiuscole. Dopo l'introduzione del testo, il programma dovrebbe visualizzare separatamente il contatore per ogni vocale.

ESERCIZIO 7.4:
Scrivete e mettete a punto un programma PASCAL che legga un numero variabile di righe, dove ciascuna riga contiene un nome di studente e una sola votazione di una scala da 0 a 100. Il nome può riempire non più di 30 colonne, il programma dovrebbe poi richiedere, sulla stessa linea visualizzata, la votazione. Il programma dovrebbe poi controllare la votazione, come potrebbe essere fatto in una procedura di validazione, alla ricerca di votazioni minori di 65 o maggiori di 100. Se la votazione è minore di 65, il programma dovrebbe visualizzare il messaggio "*** RESPINTO ***" nella riga in ingresso ed azionare il "segnale acustico" dell'elaboratore. Se la votazione fosse superiore a 100 o inferiore a 0 il programma dovrebbe visualizzare un messaggio d'errore ed azionare il segnale acustico.

Mentre accetta i valori in ingresso, il programma dovrebbe valutare la media di tutte le votazioni valide introdotte. Dopo l'immissione dell'ultimo stato, il programma dovrebbe visualizzare un messaggio contenente la valutazione media di tutti gli studenti introdotti. Il programma dovrebbe poter gestire un numero qualunque di studenti, *compreso nessuno studente*, senza finire in maniera anomala.

ESERCIZIO 7.5:
Come parte della sua attività di elaborazione dei dati, una compagnia di credito deve controllare un numero di carta su un documento in ingresso per controllarne la validità. Il numero di carta è formato da 10 cifre decimali; in aggiunta a tale numero vi sono due cifre supplementari ottenute sommando le prime 10 cifre e prendendo poi il resto della divisione di tale somma per 11 (undici). Per esempio:

0123456789 01

La somma delle prime dieci cifre è 45, e

(45 MOD 11) = 1

Scrivete e mettete a punto un programma PASCAL che legga numeri

di carte di credito codificate secondo questo schema (detto "somma di controllo") e verifichi in ogni caso che la somma di controllo sia corretta. Se risultasse non corretta il programma dovrebbe visualizzare un messaggio d'errore ed azionare il segnale acustico dell'elaboratore. Il programma dovrebbe poter gestire un qualunque numero di linee in ingresso.

Nota: Il valore binario di una cifra tipo '2' è ORD('2'). Cioè l'intero V, corrispondente al carattere CH, il cui valore è CH, è una cifra che può essere ottenuta da:

```
V := ORD(CH) - ORD('0')
```

dove CH è una variabile di <tipo> CHAR. Controllate il programma con la linea vista sopra, che si sa essere corretta, e con la seguente, non corretta:

```
0742267205 05
```

ESERCIZIO 7.6:
Un compito spesso svolto dai programmi per l'edizione di testi, che preparano testi per le pubblicazioni su quotidiani e libri, è quello di "aggiustare" la lunghezza di una riga in modo che tocchi sia il margine sinistro che quello destro. Questo è ottenuto inserendo tra le parole spazi vuoti supplementari, per esempio:

```
ADESSO I GABBIANI DEVONO ESSERE TUTTI A DORMIRE
```

diventa

```
                                                              I
ADESSO I GABBIANI DEVONO ESSERE TUTTI A DORMIRE I
```

dove il carattere 'I' simula il margine destro.

Scrivete un programma PASCAL che legga testi (non contenenti caratteri di punteggiatura) da una tastiera e inserisca spazi per ottenere l'aggiustamento destra-sinistra mostrato prima. Il numero di spazi fra le parole nella riga risultante non dovrebbe variare di più di 1. Ad esempio, sulla riga precedente 1 o 2 spazi fra le parole sono corretti nella linea risultante, mentre 1, 2, 3, 4 e 5 spazi tutti sulla stessa linea non sarebbero stati corretti. Potete assumere che la riga originaria in ingresso contiene esattamente uno spazio separante ogni coppia di parole.

Suggerimento: Contate il numero di spazi fra le parole nella riga originaria e il numero di spazi da inserire. Copiate ora uno per volta i caratteri nella nuova riga in uscita. Quando trovate uno spazio nella linea in ingresso, inserite uno o più spazi supplementari nella nuova riga. Il numero di spazi da introdurre in questo modo non si dividerà esattamente fra le parole della riga. Usate DIV per ottenere il numero di spazi da introdurre fra ogni coppia di parole e MOD per ottenere il resto che può essere distribuito ad uno spazio per volta fino all'esaurimento.

Problemi

PROBLEMA 7.1:
Disegnate i diagrammi di struttura per i programmi esemplificativi MEDIA, DISTURB e DEVOCALIZZA.

CAPITOLO 8

STRUTTURE DATI BASE I
VETTORI

1. Obiettivi

Questo è il primo di tre capitoli in cui verranno trattati tipi di dati strutturati. Questo capitolo introduce la struttura "*vettore*", usata per trattare molti elementi, tutti dello stesso <tipo>, sotto lo stesso <identificatore>.

1a. Apprendimento a lavorare con vettori ad una dimensione; usarli per salvare i dati immessi, così come sono introdotti, per un uso futuro da parte del programma.

1b. Lavoro con vettori a due dimensioni; usarli per trattare tabelle di dati.

1c. Apprendimento ad usare vettori a tre o più dimensioni.

1d. Scrittura e messa a punto di programmi implicanti vettori.

2. Premessa

Così come è utile unificare molte azioni separate, ma correlate, in un programma o in una procedura sotto uno stesso nome, allo stesso modo è utile riunire vari elementi in una unità singola, detta '*struttura di dati*'. Mentre molte strutture di dati possono essere considerate come gerarchiche (sono cioè "alberi"), non per tutte è così. In questo libro propedeutico, saranno trattate solo strutture gerarchiche di dati.

Un "*vettore*" è una struttura dati contenente due o più elementi tutti dello stesso <tipo>. Ogni elemento del vettore è individuato dalle istruzioni del programma usando l'<identificatore> di vettore e il numero (i) "indice", di posizione dell'elemento all'interno del vettore stesso. Avete già usato vettori di un <tipo> speciale, precisamente le variabili STRING. I componenti (spesso chiamati "*elementi*") di una variabile STRING sono tutti del <tipo> CHAR e i loro indici sono predefiniti tra 1 e 80. Mostreremo in questo capitolo come definire un vettore contenente elementi di un qual-

siasi <tipo> già introdotto. Nel prossimo capitolo si vedrà come definire <tipi> più complicati, che, volendo, potranno essere strutturati in vettori.

I vettori sono usati ogni qual volta è conveniente o necessario che il programma decida quale elemento scegliere da un certo gruppo.

3. Vettori inerenti all'hardware

Abbiamo visto nel capitolo 5 che ogni <identificatore>, più precisamente una <variabile> di <tipo> REAL, INTEGER, CHAR o BOOLEAN corrisponde al nome di una locazione nella memoria dell'elaboratore. Un vettore usa un <identificatore> per individuare un gruppo di locazioni di memoria. Nella forma più semplice un vettore è un gruppo contiguo di parole nella memoria centrale dell'elaboratore. La figura 8-1 mostra una piccola parte della memoria centrale sotto forma di "mappa".

Si può supporre che la memoria possa contenere almeno 4000 parole, numerate 0, 1, 2, 3,... Abbiamo mostrato solo la parte di queste parole che vanno dalla locazione 3745 alla 3760. All'interno di questa piccola parte o "*intervallo*" ("range") mostriamo sei parole individuate dall'unico <identificatore> SCORE. Ogni parola del gruppo di sei è identificata non solo dal nome "PUNTI", ma anche da un numero indicante l'ordine con cui appare nel gruppo. Tale gruppo è un "*vettore*" e deve essere dichiarato prima di essere usato perchè si tratta di <identificatori> non riservati in PASCAL.

4. Variabili con indice

Nella maggior parte dei casi, un vettore può essere usato ovunque sia utile avere una <variabile> con componenti dello stesso <tipo>. Possiamo ad esempio assegnare ad X (variabile semplice intera) un valore maggiore di 1 rispetto al terzo elemento di PUNTI come segue:

```
X=PUNTI[2] + 1
```

oppure possiamo assegnare un nuovo valore alla quarta locazione della tabella PUNTI.

```
PUNTI[3] :=X + Y
```

dove Y è anch'esso una variabile intera. In entrambi i casi, l'identificatore PUNTI appare come una "*variabile con indice*". Questo termine ci riporta all'uso dei matemati-

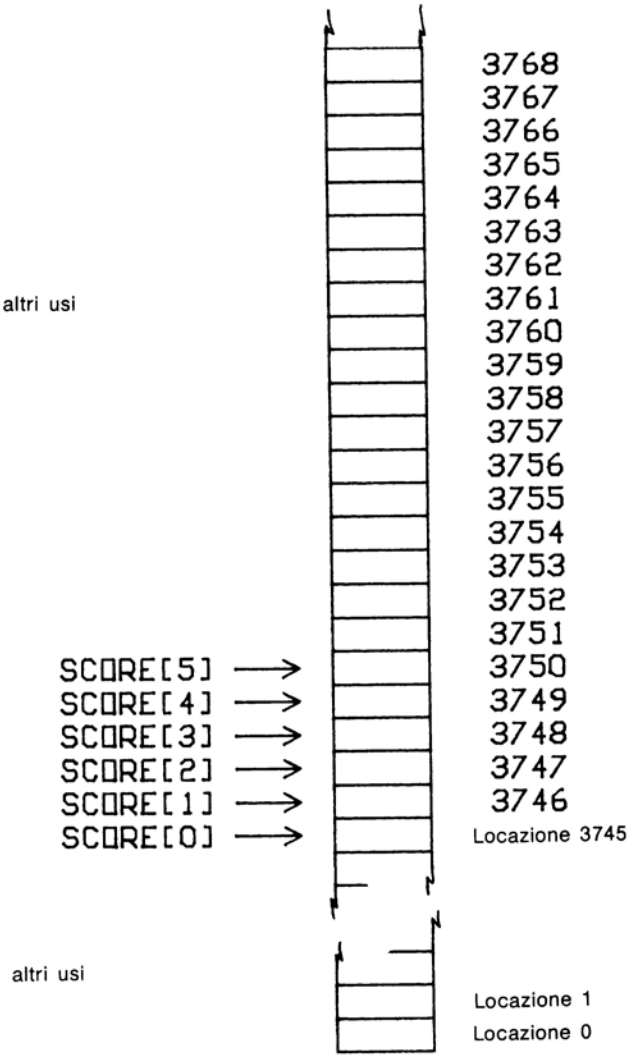

altri usi
3768
3767
3766
3765
3764
3763
3762
3761
3760
3759
3758
3757
3756
3755
3754
3753
3752
3751
SCORE[5]
SCORE[4]
SCORE[3]
SCORE[2]
SCORE[1]
SCORE[0]
3750
3749
3748
3747
3746
Locazione 3745
altri usi
Locazione 1
Locazione 0

Figura 8-1

ci di porre "*indici*" alle variabili per individuare la loro posizione in una sequenza. Potremmo ad esempio usare una variabile T per indicare l'ora del giorno. I valori successivi di T sarebbero:

$T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7\ \ldots$

Poichè gli indici (1, 2, 3, 4...) sono difficili da trattare sulla tastiera di un dispositivo d'ingresso (e anche difficili da rappresentare su di una sola scheda perforata), la sequenza precedente sarà rappresentata in PASCAL come segue:

```
T[1] T[2] T[3] T[4] T[5] T[6] T[7] ...
```

dove i valori successivi della variabile saranno memorizzati in locazioni adiacenti del vettore T.

L'utilità dei vettori sarebbe però estremamente limitata se gli indici dovessero essere <costanti intere>. PASCAL permette invece di usare come indici una qualsiasi <espressione aritmetica> il cui valore sia un intero. È ad esempio perfettamente accettabile la:

```
PUNTI[X + (Y * Z) DIV 3]
```

Il compilatore in questa situazione genera dapprima una sequenza di istruzioni in linguaggio macchina che calcolino il valore dell'espressione.

```
X + (Y * Z) DIV 3
```

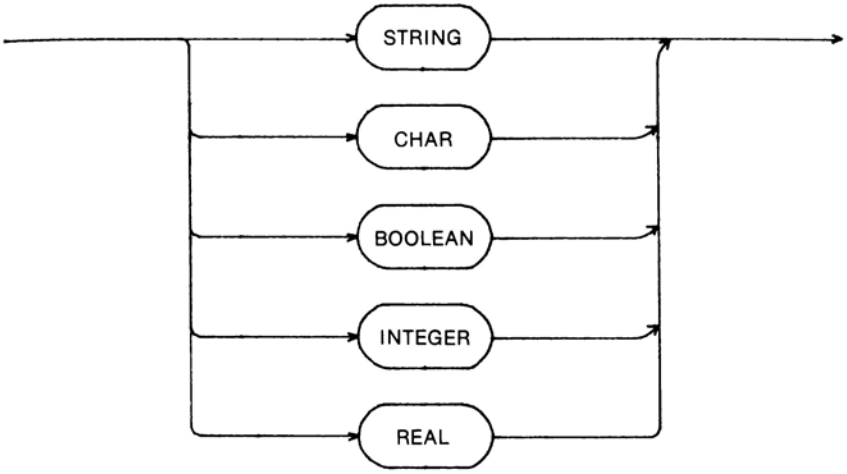
e quindi utilizza tale valore come indice con cui individuare l'elemento del vettore. Anche ALGOL e PL/1 permettono l'utilizzo di espressioni come indici. FORTRAN, COBOL e BASIC sono invece più restrittivi.

5. Dichiarazione di variabili vettori

La figura 8-2 mostra le regole sintattiche per la dichiarazione di variabili vettori. Di seguito diamo qualche esempio di dichiarazione corretta:

```
VAR TARA: ARRAY ([1..10] OF STRING;
  (*tabella contenente 10 elementi STRINGA*)
  (*numerati da 1 a 10                    *)
AC: ARRAY[0..79] OF CHAR;
  (*80 elementi di <tipo> CHAR, da 0 a 79*)
```

<tipo semplice>



<tipo>

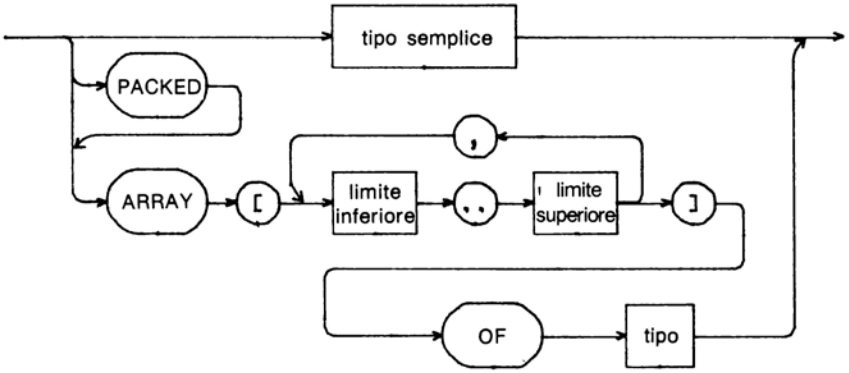


Figura 8-2

```
ABOOL: ARRAY[-5..+4] OF BOOLEAN;
  (*10 elementi booleani, da -5 a +4*)
ADOPPIO: ARRAY [1..10] OF
           ARRAY [0..4] OF INTEGER;
  (* 10 elementi numerati da 1 a 10, in cui
     ogni elemento è esso stesso una
     tabella di 5 elementi INTERI*)
ARDOPPIO: ARRAY [1..10] OF
            ARRAY[0..4] OF REAL;
  (*simile alla ADOUBLE, ma ad elementi
     REAL*)
```

La sintassi mostrata in Figura 8-2 è leggermente più esplicita di quella in Appendice E. In figura 8-2, sia il <limite inferiore>, sia il <limite superiore> sono del tipo <costante intera>. Il valore dell'indice usato per individuare un elemento nel vettore deve essere non minore del <limite inferiore> e non maggiore del <limite superiore>. Come vedremo nel Capitolo 9 vi sono vari modi di definire i limiti di un vettore. Il nostro scopo nell'utilizzare qui esplicitamente il <limite inferiore> e il <limite superiore> è quello di evitare confusioni non necessarie a questo punto.

6. Utilizzo di vettori a una dimensione

Il programma SPORTPUNTI legge due elementi da ognuna delle linee immesse. Il primo elemento è un numero a tre cifre rappresentante il punteggio ottenuto in uno sport, il secondo è il cognome del giocatore che lo ha ottenuto. Scopo del programma è quello di scegliere il giocatore che ha ottenuto il punteggio più alto, e visualizzare tale punteggio insieme al nome del giocatore.

```
 1: PROGRAMMA SPORTPUNTI;
 2: VAR SA: ARRAY[1...10] OF STRING;
 3:       IA: ARRAY[1..10] OF INTEGER;
 4:       K,LN MASS,KMASS:INTEGER;
 5:       CH:CHAR;
 6: BEGIN
 7:    K:=1:
 8:    MASS:=0;
 9:    REPEAT
10:       WRITE('PUNTEGGIO:');
11:       READ(IA[K]);
12:       IF NOT EOF THEN
```

```
13:       BEGIN
14:         WRITE('NOME:');
15:         READ(SA[K]);
16:         LN:=LENGTH(SA[K]);
17:         IF IA[K]>MASS THEN
18:         BEGIN
19:           MASS:=IA[K];
20:           KMASS:=K;
21:         END;
22:         K:=K+1;
23:       END (*non EOF*);
24:     UNTIL EOF OR (K > 10);
25:     WRITELN;
26:     WRITELN('MIGLIOR PUNTEGGIO:',IA[KMASS], ' ',SA[KMASS])
27: END.
```

Con i seguenti dati in entrata:

```
PUNTEGGIO:190 NOME:Gonzales
PUNTEGGIO:150 NOME:Jones
PUNTEGGIO:135 NOME:Alberti
PUNTEGGIO:160 NOME:De Angeli
PUNTEGGIO:203 NOME:Anelli
PUNTEGGIO:115 NOME:Raimondi
PUNTEGGIO:148 NOME:Rossi
PUNTEGGIO:175 NOME:Aschieri
PUNTEGGIO:<ETX>
```

il programma dovrebbe visualizzare in uscita:

```
MIGLIOR PUNTEGGIO:203 Anelli
```

La notazione <ETX> non è in realtà visualizzata, ma è mostrata per indicare dove il tasto (i) di Fine Flusso è (sono) pigiato (i).

Poichè non si conosce in anticipo l'identità del giocatore col punteggio più alto, è necessario prevedere nel programma un'area di memoria temporanea. Soltanto dopo la lettura di tutti i nomi e i punteggi (linea 13) sarà possibile determinare quale giocatore abbia il punteggio più alto. Usiamo le tabelle SA e IA per salvare temporaneamente i nomi e i punteggi durante la lettura. Quando è noto quale giocatore abbia il punteggio più alto, l'uscita desiderata è visualizzata alla linea 26 del programma selezionando i dati corrispondenti al giocatore dalle tabelle IA per quanto riguarda il punteggio, e SA per il nome.

Un'annotazione da fare riguardo a questo programma è che, sulla maggior parte degli elaboratori, lo spazio nella memoria centrale per la memorizzazione temporanea dei dati è generalmente ristretto. Se la lista elaborata da questo esempio di programma fosse di migliaia di nomi, sarebbe impossibile memorizzare tutti i nomi e i punteggi assieme nella memoria centrale. Quando questo succede, si deve tipicamente usare un dispositivo di memoria ausiliaria, come un disco magnetico, per trattare il superamento di capacità della memoria centrale. Maggiori dettagli su questo argomento esulano dagli scopi di questo libro.

Il ciclo di ripetizione di questo esempio si interrompe quando diventa vera una qualsiasi delle due condizioni. Poichè in ogni dichiarazione di vettori sono previsti soltanto 10 elementi, il ciclo deve interrompersi dopo la lettura da parte del programma della decima coppia di valori. Nell'esempio mostrato le coppie di dati in ingresso erano meno di 10 e il programma è terminato per la lettura della condizione di Fine Flusso.

7. Vettori di caratteri impaccati — Due dimensioni

Dobbiamo ormai aver capito che una variabile STRING è realmente un tipo speciale di vettore. Abbiamo aggiunto le variabili STRING al linguaggio PASCAL standard usato internazionalmente per avere il modo di trattare senza complicazioni inutili dati non numerici nei primi stadi di questo libro. Potete dichiarare una tabella molto simile ad una variabile STRING come segue:

```
VAR PCA: PACKED ARRAY [1..80] OF CHAR;
```

La parola riservata *"PACKED"* indica che nella memorizzazione dei caratteri del vettore deve essere usato meno spazio possibile. La rappresentazione interna di un carattere usa soltanto 8 bits di memoria. Se la parola di memoria dell'elaboratore è di 16 bits, come è probabile se usate un microelaboratore, allora due caratteri possono essere "impaccati" in una parola di memoria. (Nota: Sebbene molti microelaboratori siano basati su microprocessori a 8 bits, il nostro sistema PASCAL opera come se avessero parole di memoria di 16 bits).
Su elaboratori più grandi, in cui la parola di memoria ha più bits, è talvolta possibile impaccare 6 o 8 caratteri in una parola.

Nell'esempio SPORTPUNTI discusso nella sezione precedente, trascurammo la difficoltà presentata dal mischiare interi e informazioni di testo sulla stessa linea d'ingresso. Quando l'istruzione READ incontra una <variabile> di <tipo> STRING, tutte le rimanenti informazioni sulla linea sono assegnate a quella variabile. Questo ci obbliga a porre le informazioni numeriche a sinistra delle informazioni di testo, in modo

che i numeri possano essere convertiti nella forma interna binaria essendo assegnati alle variabili intere.

Supponiamo ora di avere buone ragioni per voler porre sulla linea visualizzata prima le informazioni di testo, seguite da una serie di numeri, per esempio:

```
Gonzales 190 150 178 135 163
```

Se usiamo schede perforate in ingresso, potremmo essere obbligati a usare un metodo simile a questo. Potremmo riservare un certo numero fisso di colonne sulla scheda per i nomi, diciamo 15 spazi. Il programma SPORTPUNTI2 potrebbe allora essere usato per leggere questi dati, calcolare il punteggio medio di ogni giocatore, e identificare il giocatore col più alto punteggio *medio*. Una delle principali differenze tra questo programma e quello che probabilmente avete scritto nell'Esercizio 7-1 è l'uso in questo programma di vettori per la memorizzazione temporanea. Ponendo le seguenti visualizzazioni in ingresso.

```
NOME:GONZALES,MARIA PUNTI:125 148 135 120 110 190
  MEDIA:138.00
NOME:JONES,BILL PUNTI:150 115 175 140 112 145 160
  MEDIA:142.43
NOME:ALBERTI,ROSANNA PUNTI:135 205 121 143 97 168
  MEDIA:144.83
NOME:DE ANGELI,MARIALUISA PUNTI:126 149 115 162 157 188
  MEDIA:149.50
NOME:ANELLI,LUIGI PUNTI:194 139 173 152 212 177<ETX>
  MEDIA:174.50
```

al termine del programma si avrà la seguente visualizzazione:

```
ANELLI,LUIGI 194 139 173 152 212 177 MEDIA:174.50
```

Ancora una volta <ETX> non è realmente visualizzato, ma mostra dove dovrebbe essere inserito il tasto di Fine-Flusso.

```
1: PROGRAMMA SPORTPUNTI2;
2: VAR NOMI: ARRAY[1..10]OF
3:             PACKED ARRAY[1..15] OF CHAR
4:   PUNTEGGI: ARRAY[1..10] OF
5:             ARRAY[0..9] OF INTEGER;
6:   SOMMA,IMASS,I,J,K:INTEGER;
7:   MED,MASSMED:REAL;
8:   CH:CHAR;
```

```
 9:    CHBUONO:BOOLEAN;
10:  BEGIN
11:    MASSMED:=0; I:=1;
12:    REPEAT
13:      WRITE('NOME:');
14:      J:=1;
15:      REPEAT
16:        READ(CH);
17:        CHBUONO:=((CH>='A') AND (CH<='Z')) OR (CH=',');
18:        IF CHBUONO THEN
19:           BEGIN NOMI[I,J]:=CH; J:=J+1; END;
20:      UNTIL (NOT CHBUONO) OR (J >15);
21:      WHILE J<=15 DO
22:      BEGIN
23:        NOMI[I,J]:=' ';
24:        J:=J+1;
25:      END;
26:      SOMMA:=0; K:=0;
27:      WRITE('PUNTI:');
28:      WHILE (NOT EOLN) AND (K < 9) DO
29:      BEGIN (*leggi e salva i punti per questo giocatore*)
30:        K:=K+1;
31:        READ(PUNTI[I,K]);
32:        SOMMA:=SOMMA+PUNTI[I,K];
33:      END;
34:      IF EOLN THEN
35:        READ(CH); (*annulla lo spazio lasciato da <RET>*)
36:      PUNTEGGI[I,0]:=K;
37:      MED:=SOMMA/K;
38:      WRITELN(' MEDIA:',MED:6:2);
39:      IF MED>MASSMED THEN
40:        BEGIN MASSMED:=MED; IMASS:=I; END;
41:      I:=I+1
42:    UNTIL EOF OR (I > 10);
43:    WRITELN; WRITELN('MIGLIOR GIOCATORE:');
44:    WRITE(NOMI[IMASS]);
45:    FOR J:=1 TO PUNTEGGI[IMASS,0] DO
46:        WRITE(PUNTEGGI[IMASS,J]:5);
47:    WRITELN(' MEDIA:', MASSMED:6:2);
48:  END.
```

Il ciclo principale di questo programma va da linea 12 a linea 42. Alla lettura di ogni linea di dati, i caratteri del nome, fino a 15, sono memorizzati in uno dei vettori

impaccati di caratteri del vettore NOMI. La lista di punteggi, fino a 9, viene poi letta e salvata nel vettore PUNTEGGI.

Questo programma illustra l'uso di vettori a due dimensioni. Fino ad ora abbiamo lavorato con vettori ad una dimensione in cui un solo intero derivante da una <espressione> è usato come indice. Un vettore a due dimensioni può essere pensato come un vettore di vettori a una dimensione, come suggerito dalle dichiarazioni di NOMI e PUNTEGGI nelle linee 2 e 4.

Le linee 31 e 32 mostrano come vengono fatti dei riferimenti a singoli voci nel vettore PUNTEGGI. Il secondo indice "K" fa riferimento a voci singole nel vettore ausiliario PUNTEGGI[I] nel solito modo. In figura 8-3 è rappresentata una sintassi semplificata di questi riferimenti. L'ordine in cui compaiono gli indici in un variabile con due indici, come I e J in questo caso, è il medesimo con cui compaiono nel vettore costituente, come nelle linee da 2 a 5.

Nel programma dell'esempio, lo scopo principale dei vettori è di fornire spazio per una memorizzazione temporanea dei dati di ciascun giocatore. Solo dopo che sono stati letti tutti i dati di tutti i giocatori si può determinare quale giocatore ha il più alto punteggio medio. Solo allora possiamo scegliere le appropriate righe di NOME e PUNTEGGI da usarsi nella stampa dei dati del giocatore che ha fatto più punti.

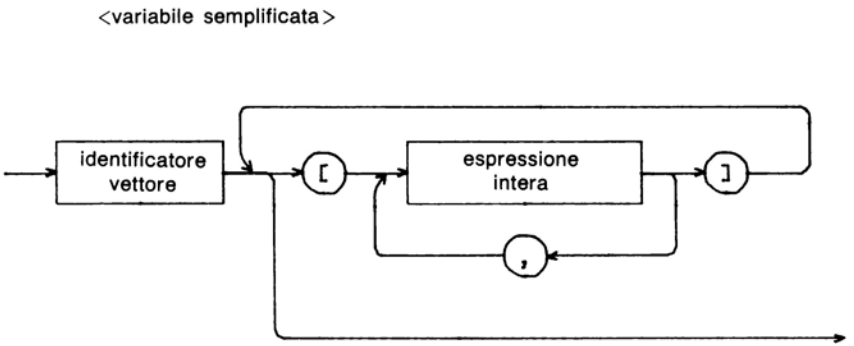


Figura 8-3

Nelle righe 38 e 47, appare la notazione di campo ":6:2", dove 6 determina la lunghezza desiderata del campo che deve essere visualizzato con l'istruzione WRITE. La sintassi è mostrata in figura 8-4. La notazione ":2" determina il numero di cifre che seguono il punto decimale. Se il numero reale richiede, includendo il punto e le due cifre decimali, più di 6 caratteri verrà visualizzato su un più alto numero di carat-

teri. Se il numero può essere visualizzato in meno di 6 caratteri, verranno inseriti degli spazi bianchi a sinistra per rendere la dimensione del campo visualizzato conforme alla prima delle due specifiche.

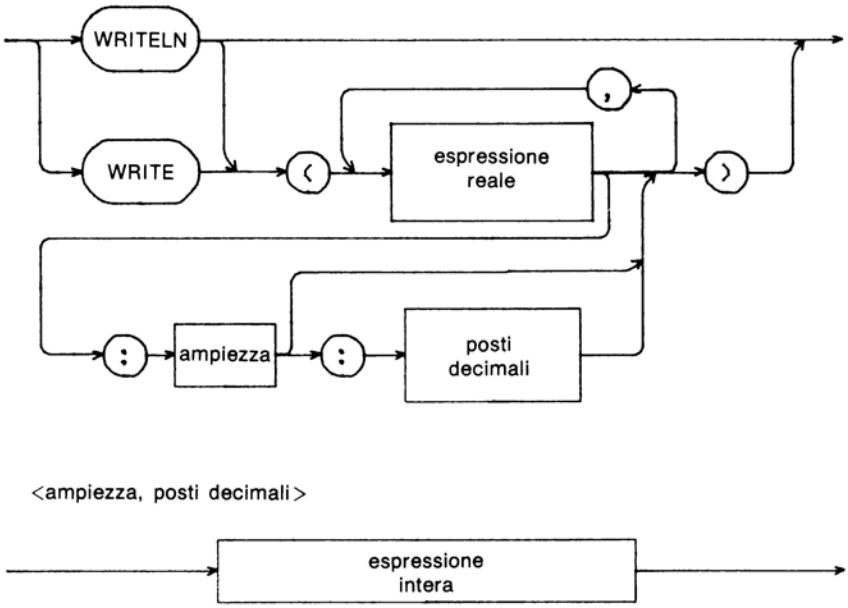


Figura 8-4

8. Somme di riga e colonna — Controlli incrociati

Quando si elaborano dati immessi nell'elaboratore da persone, è solitamente consigliabile dare dei mezzi per verificarne la correttezza, e se possibile correggerne gli errori. Capita spesso la situazione in cui parecchie voci associate con un unico individuo sono immesse nell'elaboratore, in tempi diversi, come *"transazioni"* separate. Un risultato dell'elaborazione è che i vari dati associati a quella persona sono infine messi insieme per la memorizzazione. Durante l'elaborazione si fanno generalmente un certo numero di cambiamenti e questi cambiamenti possono portare a degli errori.

Supponiamo che i dati consistano nei pagamenti fatti dagli studenti ai diversi uffici di una università, per esempio l'ufficio alloggio, la clinica (siamo in una università Americana!), la mensa, la libreria, la biblioteca (per libri in ritardo) e così via. Quando tutti i records sono messi assieme è possibile calcolare il totale pagato da ogni singolo studente. È anche possibile calcolare la somma di tutti i pagamenti avvenuti nei di-

versi uffici dell'università. Naturalmente la somma dei pagamenti fatti dai singoli studenti dovrebbe essere uguale alla somma dei pagamenti ricevuti dai diversi uffici. Il verificare che le somme danno lo stesso totale aiuta a non lasciare passare inosservato un possibile errore di elaborazione.

Se le somme sono diverse, la differenza in sè non dice *dove* è avvenuto l'errore ma serve per segnalare che qualche cosa non va e che è necessaria una correzione. Il problema QUADRATURA illustra questa situazione.

Ogni linea immessa elaborata da questo programma è costituita da tre voci. La prima, un codice alfabetico, dice quale dei cinque uffici universitari deve ricevere il pagamento. Segue quindi l'ammontare del pagamento e, come ultima voce, si ha il codice di identificazione dello studente. Nell'illustrazione si sono usati dei numeri molto piccoli per non rendere confusa la rappresentazione. Se avessimo usato il nome dello studente, oppure il numero di matricola completo, sarebbe stato necessario eseguire una ricerca in una tabella contenente queste informazioni per associarvi il numero che lo individua nella tabella dei pagamenti. Esamineremo gli algoritmi di ricerca nel capitolo 13. Per il programma QUADRATURA, dovete assumere che il numero di identificazione sia quello che si sarebbe ottenuto con una ricerca fatta da un altro programma.

```
 1: PROGRAMMA QUADRATURA;
 2: VAR TRANS: ARRAY[1..10]
 3:              OF ARRAY[1..5] OF REAL;
 4:   CH:CHAR;
 5:   COL,RIG,SNUM:INTEGER;
 6:   PAGATO,RIGTOTALE,COLTOTALE:REAL;
 7:   COLSOMME: ARRAY[1..5] OF REAL;
 8:   RIGSOMME: ARRAY[1..10] OF REAL;
 9:
10: PROCEDURE SOMME;
11: BEGIN
12:   FOR RIG:=1 TO 10 DO
13:   BEGIN
14:     FOR COL:=1 TO 5 DO
15:       RIG SOMME[RIG]:=RIGSOMME[RIG]+TRANS[RIG,COL];
16:     RIGTOTALE:=RIGTOTALE+RIGSOMME[RIG];
17:   END;
18:   FOR COL:=1 TO 5 DO
19:     COLTOTALE:=COLTOTALE+COLSOMME[COL];
20: END (*SOMME*);
21:
22: PROCEDURE SALVA(SNUM,COL:INTEGER);
```

```
23: BEGIN
24:   TRANS[SNUM,COL]:=PAGATO;
25:   COLSOMME[COL]:=COLSOMME[COL]+PAGATO;
26: END (*SALVA*);
27:
28: PROCEDURE INDATI;
29: BEGIN
30:   WHILE NOT EOF DO
31:   BEGIN
32:     WRITE('CODICE UFFICIO:');
33:     READ(CH);
34:     WRITE(' AMMONT:');
35:     READ(PAGATO);
36:     WRITE(' IDENTIF:');
37:     READ(SNUM);
38:     WRITELN;
39:     CASE CH OF
40:       'B': SALVA(SNUM,1);
41:       'C': SALVA (SNUM,2);
42:       'H': SALVA(SNUM,3);
43:       'L': SALVA(SNUM,4);
44:       'M': SALVA(SNUM,5)
45:     END (*CASE*);
46:     READ(CH); (*perdi spazio dopo SNUM*)
47:   END (*Lettura*);
48: END (*INDATA*);
49:
50:
51:
52: PROCEDURE INIZ;
53: BEGIN
54:   FOR COL=1 TO  5 DO COLSOMME[COL]:=0;
55:   FOR RIG:=1 TO 10 DO RIGOSMME[RIG]:=0;
56:   FOR RIG:=1 TO 10 DO
57:     FOR COL:=1 TO 5 DO
58:       TRANS[RIG,COL]:=0;
59:   RIGTOTALE:=0; COLTOTALE:=0;
60:   WRITELN('QUADRATURA');
61: END (*INIZ*);
62:
63: PROCEDURE VISUAL;
64:   PROCEDURE METREALE (R:REAL);
65:   VAR I:INTEGER;
```

```
66:    BEGIN
67:      IF R>=1.0 THEN WRITE(R:7:2) (*in dollari!*) ELSE
68:        BEGIN
69:          I:=ROUND(R*100); WRITE(' ':4);
70:          IF I=0 THEN WRITE(' 0') ELSE
71:          BEGIN
72:            WRITE('.');
73:            IF I<10 THEN WRITE('0',I)
74:                    ELSE WRITE(I);
75:          END;
76:        END (*R<1.0*);
77:    END (*METREALE*);
78:  BEGIN (*VISUAL*)
79:    WRITELN ('LIBRI':7, 'MENSA':7,'ALLOG':7,'BIBL':7
80:                        'MED':7',TOTALE':7);
81:    FOR RIG=1 TO 10 DO
82:    BEGIN
83:      FOR COL:=1 TO 5 DO
84:        METREALE(TRANS[RIG,COL]);
85:      METREALE(RIGSOMME[RIG]); WRITELN;
86:    END;
87:    WRITELN;
88:    FOR COL:=1 TO 5 DO METREALE(COLSOMME[COL]);
89:    METREALE(COLTOTALE); WRITELN;
90:    IF COLTOTALE< >RIGTOTALE THEN
91:      BEGIN
92:        WRITELN('*** ERRORE: RIGTOTALE=');
93:        METREALE( RIGTOTALE);
94:      END;
95:  END (*VISUAL*);
96:
97:  BEGIN (*PROGRAMMA PRINCIPALE*)
98:    INIZ;
99:    INDATI;
100:   SOMME;
101:   VISUAL;
102: END.
```

Riportiamo un esempio delle linee visualizzate durante la fase di immissione del programma QUADRATURA.

```
CODICE UFFICIO: B     AMMONT: 7.95 IDENTIF:2
CODICE UFFICIO: H     AMMONT: 150.00 IDENTIF:1
```

```
CODICE UFFICIO: H     AMMONT: 150.00 IDENTIF:3
CODICE UFFICIO: L     AMMONT: 4.00 IDENTIF:2
CODICE UFFICIO: C     AMMONT: 2.25 IDENTIF:4
CODICE UFFICIO: M     AMMONT: 55.60 IDENTIF:9
CODICE UFFICIO: M     AMMONT: 10.00 IDENTIF:7
CODICE UFFICIO: B     AMMONT: 15.80 IDENTIF:6
CODICE UFFICIO: B     AMMONT: 14.25 IDENTIF:8
CODICE UFFICIO: C     AMMONT: 75.00 IDENTIF:10
CODICE UFFICIO: H     AMMONT: 300.000 IDENTIF:5
CODICE UFFICIO: L     AMMONT: 0.50 IDENTIF:6
CODICE UFFICIO: M     AMMONT: 40.00 IDENTIF:1<ETX>
```

A fine immissione il programma dovrebbe visualizzare la seguente tabella:

LIBRI	MENSA	ALLOG	BIBL	MED	TOTALE
0	0	150.00	0	40.00	190.00
7.95	0	0	4.00	0	11.95
0	0	150.00	0	0	150.00
0	2.25	0	0	0	2.25
0	0	300.00	0	0	300.00
15.80	0	0	.50	0	16.30
0	0	0	0	10.00	10.00
14.25	0	0	0	0	14.25
0	0	0	0	55.60	55.60
0	75.00	0	0	0	75.00
38.00	77.25	600.00	4.50	105.60	825.35

Per permettervi di verificare l'addizione abbiamo immesso nel programma un piccolo numero di dati. La linea in fondo alla tabella contiene le somme di tutte e 6 le colonne. Il numero all'incrocio tra ultima riga ed ultima colonna dovrebbe essere sia la somma degli elementi della riga che degli elementi della colonna associata. Questo numero è memorizzato in COLTOTALE ed è stampato in linea 89. Poichè il messaggio di errore di riga 92 non è visualizzato significa che RIGTOTALE è stato correttamente calcolato uguale a COLTOTALE. Le linee orizzontali di numeri in una tabella come questa sono solitamente chiamate *"righe"* mentre i numeri raggruppati in verticale formano delle *"colonne"*. Il metodo di verificare l'uguaglianza della somma dei totali di colonna con i totali di riga è spesso chiamato "controllo incrociato": termine che proviene dalle prime macchine a schede perforate.

La procedura METREALE è usata per evitare di visualizzare numeri inferiori a 1.00 in notazione scientifica; per evitare, ad esempio, di avere 0.50 visualizzato come

5.0E-1. Se la quantità è inferiore ad 1 dollaro viene calcolato l'equivalente numero intero di centesimi che viene poi visualizzato nella forma corretta da METREAL.

9. Tre e più dimensioni

Molto spesso capita di dover eseguire dei calcoli su tabelle con più di 2 dimensioni. In questa sezione discutiamo un'applicazione tipica ma lasciamo a voi il compito di scrivere il programma. I principi coinvolti sono una semplice estensione dei principi appena discussi per i vettori a due dimensioni. Tuttavia, quando il numero di dimensioni aumenta, diviene sempre più difficile rappresentare il problema con schemi o programmi semplici.

Supponete di far parte di una commissione pubblica come: commissione per l'energia, mezzogiorno... L'organizzazione ha bisogno, per determinare le priorità degli interventi considerati più importanti, di un campionamento regolare delle opinioni dei suoi membri. La tabella che segue illustra il tipo di dati raccolti dall'intervista di circa 1000 membri.

	Aumento	*Nessun cambiamento*	*Riduzione*	*Nessuna Opinione*
	1.	*2.*	*3.*	*4.*
1. Ambiente	243	527	149	53
2. Energia	185	617	195	78
3. Inflazione	318	442	83	27
4. Disoccupazione	306	499	117	62
5. Mezzogiorno	97	377	270	141

Ciascun numero nella tabella rappresenta il numero di membri che ha indicato che la commissione dovrebbe dedicare l'attenzione indicata nelle intestazioni delle colonne ai problemi indicati nelle 5 righe. Chiaramente questi dati possono facilmente essere memorizzati in un vettore a 2 dimensioni. Ma la commissione deve eseguire un campionamento sui suoi membri per quanto riguarda queste opinioni una volta ogni 3 mesi se vuol tenere sotto controllo la situazione. Si potrebbe per esempio ritenere che l'opinione sull'importanza del problema energetico sia bruscamente variata dopo che gli strumenti di informazione (giornali, TV,...) hanno catastroficamente annunciato la fine dell'era del petrolio. È perciò importante avere parecchie versioni di questa tabella che rappresentano i dati raccolti in tempi diversi. Un programma scritto per analizzare le tendenze insite nelle variazioni dei dati avrebbe la necessità di avere contemporaneamente in memoria tutte le versioni.

Questo può essere ottenuto usando un vettore di vettori a due dimensioni. Questo vettore è solitamente chiamato, più semplicemente, vettore a tre dimensioni. Ecco come un tale vettore può essere dichiarato in PASCAL:

```
YAR OPINIONE: ARRAY[1..12] OF (*numero campionamenti*)
                  ARRAY[1..5] OF (*5 problemi per campionamento*)
                    ARRAY[1..4] OF INTEGER;(*4 codici di opinione*)
```

Per far riferimento ad una qualsiasi locazione in questo vettore bisogna usare una variabile con tre indici, una per ogni dimensione:

```
OPINIONE[CAMP, PROBLEMA, CODICE]
```

dove CAMP, PROBLEMA e CODICE sono variabili intere.

Proseguendo sulla stessa linea, potreste aver bisogno di determinare qualche cosa per quanto riguarda i gruppi di età in relazione alle preferenze. Potreste quindi aggiungere un'altra dimensione al vettore OPINIONE che porterebbe a qualche cosa del genere:

```
VAR OPINIONE: ARRAY[1..4] OF (*4 gruppi di età*)
                  ARRAY[1..12] OF (*numero campionamenti*)
                    ARRAY[1..5] OF (*5 problemi*)
                      ARRAY[1..4] OF INTEGER; (*4 codici di opinione*)
```

Il riferimento ad un singolo elemento di questo vettore richiede l'uso di 4 indici.

ESERCIZIO 8.1:
Scrivete e correggete un programma PASCAL per risolvere il problema che segue. Tracciate un diagramma di struttura per descrivere l'algoritmo usato.

Gli elaboratori sono spesso usati per eliminare voci doppie da un elenco. Negli esempi si possono includere l'elaborazione di elenchi di firme presentate per una petizione, elenchi di nomi ed indirizzi gestiti da una società che vende elenchi di indirizzi, elenchi di numeri di identificazione (come il numero di codice fiscale) in cui possono essere stati fatti degli errori o ci sono numeri falsi (!), e così via. A scopo illustrativo in questo problema useremo piccoli numeri interi, anche se i dati da verificare dovrebbero solitamente essere costituiti da stringhe alfanumeriche, o numeri contenenti molto più di due cifre.

L'elenco che vi presentiamo è talmente corto che non è necessario considerare degli efficienti algoritmi di ricerca come quelli presentati nei capitoli 13, 14 e 15.

Dato un elenco di numeri si tratta di eliminare i numeri duplicati, visualizzando alla fine solo la prima presenza del numero. Costruite il vostro programma in modo tale che tratti un elenco contenente fino a 100 elementi. Usate il seguente breve elenco per verificare il vostro programma.

```
2 3 2 2 6 4 9 7 4 5 3
```

Il risultato sarà:

```
2 3 6 4 9 7 5
```

Lanciate ora il programma con il seguente elenco:

```
74  74  92  72  29  34  65  34  43  23
91  81  61  43  74  83  83  77  79  83
64  24  22  20  49  65  88  60  43  63
99  84  23  48  27  43  83  74  83  91
```

Il programma dovrebbe visualizzare l'elenco senza i numeri duplicati, e con i numeri rimanenti sempre nello stesso ordine dell'elenco originale. Come verifica dei risultati ottenuti, dovrebbe visualizzare i numeri che ha trovato più di una volta ed il numero di volte che erano ripetuti. Analizzate l'elenco originale con una penna, assicurandovi che i risultati sono corretti. Il programma dovrebbe essere progettato per analizzare numeri che abbiano fino a 5 cifre. Questo sarebbe più simile al tipo di dati che si possono incontrare in un'applicazione realistica di questo problema.

Metodo proposto: memorizzate ogni numero successivamente immesso in un vettore, senza però i numeri duplicati. In un secondo vettore memorizzate il numero delle volte che compare ogni numero. Durante la lettura di ogni numero, scandite, nel primo vettore, i numeri già memorizzati cercando le duplicazioni; se ne trovate una incrementate di 1 il corrispondente contatore nel secondo vettore. Se il numero non esiste, memorizzatelo nel primo vettore e ponete ad 1 il contatore del secondo. Dopo che tutti i numeri sono stati letti, l'elenco risultante può essere visualizzato direttamente partendo dal primo vettore, in quanto i numeri che vi sono contenuti rispecchiano l'ordine

di immissione. L'elenco dei numeri duplicati può essere prodotto scandendo i contatori del secondo vettore e cercando i valori maggiori di uno. Usate delle variabili intere distinte per tener conto dei numeri già immessi, e per puntare alla prossima posizione libera del primo vettore.

ESERCIZIO 8-2:
Modificate il programma SPORTPUNTI2 affinchè visualizzi i nomi dei giocatori, i punteggi e le medie, secondo valori di media decrescente: il giocatore con la media più alta deve cioè essere visualizzato per primo. Consiglio: usate un vettore ausiliario MEDIE come base per memorizzare le medie e controllare l'ordine di visualizzazione. Dopo che sono state lette tutte le linee, scandite MEDIE alla ricerca del valore più alto, visualizzate il numero corrispondente e l'elenco dei punti, ponete quindi la corrispondente locazione di MEDIE a 0 affinchè non venga più usata. Continuate fino a che tutti i valori di MEDIE sono a zero.

ESERCIZIO 8-3:
È spesso richiesto ai programmatori di *riorganizzare* un insieme di dati secondo linee simili a quelle illustrate in questo problema. La parte "prima" della tabella mostra un vettore DATI contenente 101 righe di 5 elementi ciascuna; il vettore associato SELEZIONE contiene elementi Booleani. Il contenuto di DATI deve essere riorganizzato cosicchè contenga solo le righe corrispondenti al valore TRUE in SELEZIONE (con tutte le altre righe di dati sostituite con FALSE), ed in modo che l'ordine di apparizione delle righe selezionate sia *invertito*. Le righe non selezionate, che devono essere riempite con zeri, devono essere poste alla fine del vettore DATI riorganizzato.

Disegnate dapprima un diagramma di struttura che descriva un algoritmo che operi codesta riorganizzazione. Scrivete e correggete quindi il corrispondente programma PASCAL. Nota: potrebbe essere conveniente dichiarare un secondo vettore dello stesso < tipo > di DATI per la memorizzazione mentre è in corso la riorganizzazione.

Come dati di verifica, riempite inizialmente il vettore DATI con numeri casuali generati da una procedura simile alla procedura CASUALI usata nel programma QUATTROLETTERE e CASUALGIRO nella sezione 11 del capitolo 5. Mettete a punto il generatore di numeri casuali per produrre numeri da 1 a 99. Posizionate SELEZIONE a TRUE

per una riga in DATI solo se il primo numero casuale nella riga termina con la cifra "3".

		PRIMA	DOPO
RIGA	SELEZIONE	DATI	DATI
0	F	27 54 32 68 13	93 32 04 53 67
1	T	93 74 08 16 56	83 33 09 54 14
2	T	73 61 01 91 76	
3	F	16 55 27 36 79	etc.etc.
4	T	53 90 52 82 58	
			53 90 52 82 58
			73 61 01 91 76
			93 74 08 16 56
		etc.etc.	0 0 0 0 0
97	T	83 33 09 54 14	etc.etc.
98	F	95 98 44 33 86	
99	T	93 32 04 53 67	0 0 0 0 0

ESERCIZIO 8-4:
Scrivete e correggete un programma PASCAL che scandisca un testo Italiano, con un numero variabile di linee, e conti il numero delle volte in cui compare ogni lettera. Potete ignorare gli spazi ed i caratteri di punteggiatura. Assumete che tutte le lettere siano maiuscole. Terminata la lettura, visualizzate una tabella in cui sono riportate le lettere ed il numero di volte in cui le lettere stesse erano presenti. Suggerimento: è corretto dichiarare un vettore che ha come limite inferiore 'A' e superiore 'Z' e far riferimento ad una locazione ARA[CH] dove CH è una variabile di < tipo > CHAR.

ESERCIZIO 8-5:
Tracciate un diagramma di struttura che pur grossolano sia facilmente leggibile e descriva come risolvereste il problema che segue; scrivete poi e correggete il corrispondente programma PASCAL.

Leggete 6 valori interi da ognuna delle 12 righe immesse (72 valori in tutto). Mettete i dati in un vettore bidimensionale con 12 righe e 6 colonne: ogni riga corrisponde ai dati di una linea. Riordinate i dati in ogni *colonna* in modo che il valore più piccolo appaia nelle prima riga, ed i valori successivamente più grandi nelle righe successive. Visualizzate infine tutti i valori del vettore *riordinato*.

ESERCIZIO 8-6:
Supponete di essere un linguista interessato allo studio di quanto spesso parole diverse sono usate nella lingua italiana. Supponete che il testo italiano che leggete nel vostro programma non contenga più di 200 parole distinte provenienti da un numero variabile di linee. Tracciate un diagramma di struttura approssimato che descriva come risolvereste il problema. Sviluppate quindi e correggete un programma PASCAL che svolga le azioni dell'algoritmo descritto dal diagramma.

Contate il numero di volte che ogni parola distinta compare nel testo. Assumete che nessuna parola superi i 15 caratteri, e che nessuna parola sia troncata per andare a capo. Alla fine del programma visualizzate ogni parola con associato il contatore partendo dalla parola con il conteggio più alto e procedendo in modo decrescente. Potete usare, come dati di verifica, il testo di questo esercizio. Semplificate il problema usando solo lettere maiuscole. Se S1 e S2 sono variabili STRING, è corretto usare:

```
IF S1 = S2 THEN...
```

ESERCIZIO 8-7:
Tutti e tre i programmi analizzati in questo capitolo hanno la proprietà non molto simpatica di essere "inesorabili" per quanto riguarda gli errori di battitura. Li abbiamo presentati in quella forma, principalmente per evitare complicazioni per quanto riguarda il formato di presentazione, che ci avrebbero distratto dai punti principali dell'analisi. Se avete implementato e provato uno qualsiasi di questi programmi, avrete scoperto di non poter usare i tasti <backspace> o <rubout> per cancellare un errore di battitura come era possibile in parecchi dei programmi precedenti..

Un modo semplice per evitare questo problema è di usare variabili STRING per l'immissione. Per immissioni a una variabile STRING, il nostro sistema PASCAL accetta tutti i caratteri fino a che è battuto <RET>; o <backspace> e <rubout> (<DEL>) servono per cancellare un carattere o la linea battuta fino a quel momento. Fino a che non è battuto <RET> il vostro programma non esamina i caratteri introdotti. Questo permette semplici correzioni di errori sullo schermo prima che il programma si scontri con problemi che sarebbero complessi da correggere. Una difficoltà connessa con questo metodo è che dovete accettare ogni nuovo elemento su una nuova riga, se è usato un suggerimento per ogni voce, oppure dovete battere "alla cieca" senza separatori di suggerimento tra ogni elemento.

Come esercizio modificate i programmi SPORTPUNTI2 e QUADRATURA per usare i metodi di immissione appena descritti. Dovrete cambiare il modo in cui i numeri sono convertiti da caratteri alla forma binaria interna usando vostre funzioni per eseguire la conversione. Ricordate che il valore intero di un carattere, per esempio D, può essere ottenuto da:

```
D := ORD(CH) - ORD('0')
```

dove CH è di <tipo> CHAR, e la funzione interna ORD converte un carattere in forma intera.

CAPITOLO 9

STRUTTURE DATI BASE II
INSIEMI (SETS)

1. Obiettivi

Questo capitolo tratta soprattutto di metodi per manipolare informazioni che devono, per essere elaborate, essere raggruppate in categorie. Sono introdotti il < tipo > PASCAL SET ed i numerosi concetti ad esso correlati.

1a. Apprendimento a definire variabili *scalari* per trattare dati non numerici che possono essere raggruppati in unità logiche.

1b. Apprendimento ad usare variabili *sottocampo* per evitare che il programma tenti di elaborare valori che non dovrebbero esserci.

1c. Apprendimento a definire vostri < *tipi* > per le variabili ed ad usare e dichiarare variabili di quei < tipi >.

1d. Apprendimento ad usare i *sets* per semplificare verifiche su complesse combinazioni di dati.

1e. Modifiche a programmi di moderata complessità usando i nuovi concetti introdotti in questo capitolo.

2. Premessa

Mentre il concetto di vettore permette di associare molti dati dello stesso < tipo > sotto un unico nome, è spesso necessario eseguire le stesse operazioni su molti elementi associati soltanto per il loro *valore*. In un'università, per esempio, un insieme di elaborazioni potrebbe essere applicabile soltanto agli iscritti a Scienze, mentre un altro lo potrebbe essere per gli specializzandi in Lettere e così via. Le registrazioni degli studenti conterranno quindi tipicamente un elemento "*codice*" rappresentante la

facoltà dichiarata. Nel corso delle elaborazioni si potrebbe quindi usare una complessa sequenza di istruzioni IF per determinare se uno studente è iscritto a discipline scientifiche. Per esempio:

```
IF (FACOLTÀ=2) (*biologia*)
  OR (FACOLTÀ=4) (*chimica*)
  OR (FACOLTÀ=15) (*fisica*)
  OR (FACOLTÀ=16) (*fisica applicata*)
    THEN TRATTASCIENZE;
```

Poniamo in questo esempio che TRATTASCIENZE sia una procedura scritta per elaborazioni riguardanti gli studenti di discipline scientifiche. I "codici" sono valori numerici assegnati arbitrariamente come rappresentazione di ogni facoltà. Codici di questo tipo sono stati spesso usati nelle elaborazioni di dati commerciali perché risparmiano spazio e permettono una maggiore efficienza rispetto a quella che si avrebbe con l'uso di stringhe più facilmente riconoscibili dall'uomo.

Naturalmente è possibile fare un simile controllo soltanto al momento dell'introduzione dei dati nell'elaboratore e porre poi un diverso codice nella registrazione per indicare se lo studente è iscritto a una facoltà scientifica, artistica, letteraria o qualsivoglia. Può comunque non essere sempre prevedibile in anticipo quali categorie di questo tipo siano importanti. Resta quindi la probabilità di dover scrivere complicate sequenze di istruzione IF per separare i dati in categorie rilevanti.

PASCAL prevede diverse possibilità che semplificano notevolmente il trattamento di elementi correlati, vale a dire:

a) Una <variabile> *"SET"* è simile ad un vettore contenente soltanto elementi Booleani. Ogni elemento di un SET corrisponde ad un valore specifico. È possibile che tutti gli elementi Booleani all'interno di un SET siano contemporaneamente TRUE. Per esempio, in un SET rappresentante tutti gli studenti universitari, gli elementi rappresentanti gli studenti del primo, secondo, terzo e dell'ultimo anno sarebbero tutti TRUE, ma quelli rappresentanti i laureati e gli specializzati potrebbero non essere FALSE. Gli elementi corrispondenti a valori TRUE sono detti *"membri"* del SET.

b) Una <variabile> *"SCALARE"* permette di associare un nome ad ogni possibile codice per elementi che debbano essere logicamente raggruppati. Questo permette di non riportare il valore del codice durante la stesura dei programmi perché verrà risolto dal compilatore PASCAL. Come esempio, l'istruzione IF vista precedentemente potrebbe essere modificata nella seguente:

```
IF (FACOLTÀ=BIOLOGIA) OR (FACOLTÀ=CHIMICA)
  OR (FACOLTÀ=FISICA) OR (FACOLTÀ=FISAPPLIC) THEN
    TRATTASCIENZA
```

Come si vedrà in questo capitolo anche questa complicazione può essere evitata associando una <variabile> SET con i codici definiti in modo da rendere veramente molto semplice quest'istruzione IF.

c) Si può definire una <variabile> *"SOTTOCAMPO"* che includa sotto certi limiti, tutti i valori interi o scalari. Una <variabile> UNIVERSITARIO potrebbe cioè assumere qualsiasi valore da MATRICOLA a LAUREANDO. Una <variabile> ORA potrebbe avere un valore qualunque da 0 fino a 24. Il tentativo di assegnare ad una variabile SOTTOCAMPO un valore esterno all'intervallo dichiarato provocherà una fine abnorme del programma da parte del sistema PASCAL. Poichè il sistema PASCAL controlla che soltanto valori accettabili vengano associati a <variabili> SOTTOCAMPO, si è protetti nei confronti di programmi in esecuzione contenenti errori logici difficili da scoprire.

Per illustrare tali concetti in questo capitolo e nel successivo, esamineremo i tipi di elaborazione che potrebbero essere fatti da un professore interessato alla determinazione dei progressi dei vari studenti della sua classe. Gli aspetti dettagliati di questo esempio saranno studiati man mano che si procede di sezione in sezione.

3. Tipi scalari

Ogni qualvolta i problemi di elaborazione dei dati rendano conveniente suddividere gli elementi in *categorie* distinte, apparirà utile agganciare ad ogni categoria distinta un semplice *codice* numerico. Si porebbe ad esempio utilizzare i seguenti codici numerici per separare gli studenti in categorie associate ad ogni livelllo di classe.

LIVELLO	CODICE
matricola	1
secondanno	2
terzanno	3
laureando	4
laureato	5
specializzato	6

Un tale codice fa risparmiare memoria rispetto al nome esteso del livello poiché il codice può essere memorizzato nello spazio necessario per un singolo carattere (o in qualche caso per un intero).

Codici di questo tipo sono stati a lungo usati nelle elaborazioni dati sia per risparmiare spazio sulle schede perforate, o nella memoria dell'elaboratore, sia per ridurre

il tempo di elaborazione richiesto. Ad esempio è molto più semplice fare un controllo del tipo:

```
IF LIVELLO = 2 THEN <istruzione>
```

che costruire un ciclo che confronti carattere per carattere una <stringa> contenente la parola "SECONDANNO" con un vettore contenente la stessa parola. Sfortunatamente la semplificazione risultante nel programma dall'uso di codici numerici al posto di parole ha anche avuto l'effetto di rendere gli stessi programmi più facilmente sbagliati per errori dovuti a confusioni del programmatore. In un grosso programma, dove parecchi elementi distinti sono suddivisi in categorie con codici numerici, è molto facile che il programmatore dimentichi i tanti diversi significati associati allo stesso valore del codice. C'è poi il problema che si è portati a scrivere programmi che possono usare valori di codici non esistenti.

PASCAL è stato costruito per eliminare questi problemi, caratteristici di tutti i linguaggi di programmazione più diffusi. Si può dichiarare un nuovo <tipo> associato ad una <variabile> che possa assumere solo codici ammessi. Per esempio:

```
VAR L1, L2: (MATRICOLA, SECONDANNO, TERZANNO, LAUREANDO,
             LAUREATO, SPECIALIZZATO);
```

Questo permette di scrivere il controllo visto in precedenza come segue:

```
IF L1= SECONDANNO THEN <istruzione>
```

Ogni identificatore nella parte <tipo> della dichiarazione di L1 e L2 può essere considerato una <costante> di un tipo associato a quella dichiarazione. È cioè corretto usare un'istruzione del tipo:

```
L2 := TERZANNO
```

o:

```
IF L2 >= LAUREATO THEN <istruzione>
```

I codici dell'<identificatore> nella dichiarazione di L1 e L2 hanno valori numerici per le elaborazioni interne del vostro programma. Il primo valore richiamato nella dichiarazione ha codice interno di valore 0, il secondo 1, e così via. Quindi SECONDANNO è maggiore di MATRICOLA e tutti i valori da MATRICOLA fino a LAUREANDO sono minori del valore di LAUREATO.

È anche possibile usare un'istruzione CASE del tipo:

```
CASE L1 OF
  MATRICOLA: P1;
  SECONDANNO: P2;
  TERZANNO, LAUREANDO: P3;
  LAUREATO: P4;
  SPECIALIZZATO: P5;
END (*CASE*);
```

dove si suppone che da P1 a P5 siano tutte procedure precedentemente definite nel programma.

Il < tipo > delle variabili L1 e L2, come stabilito dalla dichiarazione precedente è *SCALARE*. Il compilatore PASCAL protegge dagli assegnamenti di valori impropri ad una < variabile > tipo L1, poiché vi sono soltanto sei possibili valori associati a L1. Allo stesso modo si può considerare una < variabile > Booleana come equivalente a quella che appare in una dichiarazione del tipo

```
VAR BOOL: (TRUE, FALSE)
```

Si noti comunque che non è possibile usare gli identificatori "TRUE" e "FALSE" in questo tipo di dichiarazione. Tali dichiarazioni hanno l'effetto di associare un valore costante ad ogni identificatore nell'elenco fra parentesi, e TRUE e FALSE sono predichiarate dal sistema in associazione al < tipo > BOOLEAN. Al tentativo di assegnare qualcosa di diverso dagli identificatori costanti dichiarati o dal valore di un'espressione associato allo stesso < tipo >, il compilatore genererà un messaggio di errore sintattico.

4. Dichiarazione di Tipi Propri

Un altro modo per dichiarare le variabili L1 e L2 usate nella sezione precedente potrebbe esserè il seguente:

```
TYPE LIVELLO = (MATRICOLA, SECONDANNO, TERZANNO, LAUREAN-
                DO,
                LAUREATO, SPECIALIZZATO);

VAR L1,L2: LIVELLO;
```

L'identificatore riservato *"TYPE"* è usato in modo simile all'identificatore "VAR" in

quanto introduce una sequenza di dichiarazioni nella testata di un <blocco>. Nelle figure 9-1 e 8-2 sono mostrate le regole sintattiche fondamentali. La dichiarazione precedente associa l'<identificatore> LIVELLO al <tipo> SCALAR mostrato sulla destra del segno uguale ('='). D'ora in poi la comparsa dell'identificatore LIVELLO in dichiarazioni di variabile o parametro dovrà soddisfare le richieste della sintassi date per un <tipo>. Si noti che la sintassi di un <blocco> richiede che tutte le dichiarazioni di nuovi <tipi> precedano nel blocco il loro utilizzo nelle dichiarazioni di variabile VAR. *L'ambiente* degli identificatori <tipo> è soggetto alle stesse regole dell'ambiente di identificatori <variabile>.

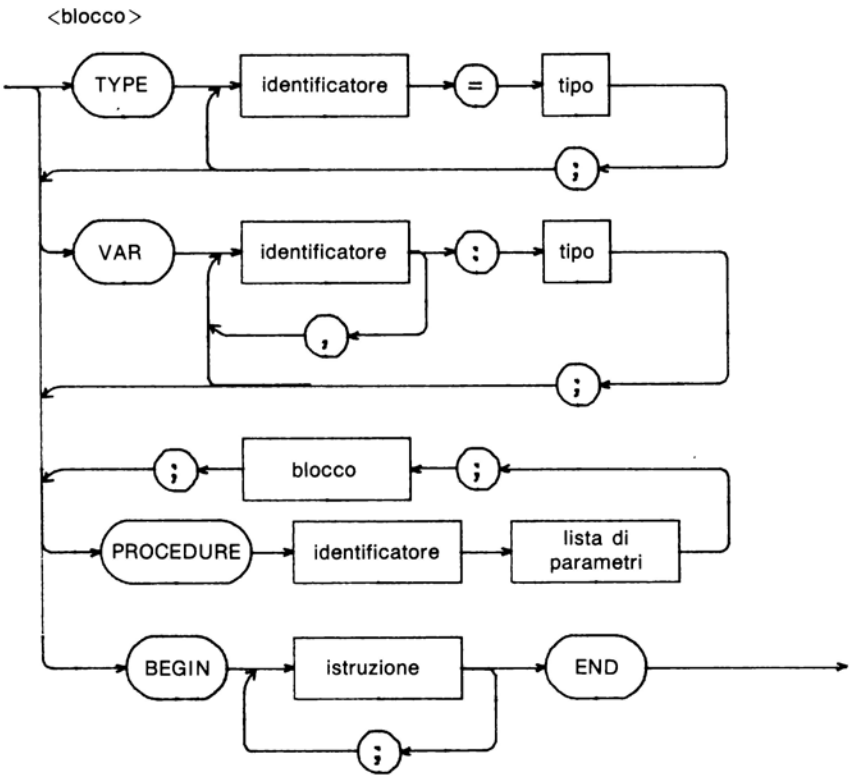


Figura 9-1

Come si vedrà nelle prossime sezioni di questo capitolo e del successivo, si possono dichiarare identificatori associati ad un'ampia varietà di nuovi <tipi>. Uno dei vantaggi nell'utilizzo di <tipi> propri per dichiarare variabili usate per scopi particolari è che il compilatore può aiutare ad evitare errori. Ogniqualvolta venga assegnato un nuovo valore ad una variabile, il compilatore controlla che il <tipo> dell'espres-

sione alla destra dell'operatore di assegnamento (':=') sia *compatibile* con il < tipo > della variabile. Se non sono compatibili viene generato un messaggio di errore sintattico. L'interpretazione più semplice della "compatibilità" nel caso di variabili Scalari è la richiesta di identità di < tipo >.

Il compilatore controlla il < tipo > dell'espressione anche in un confronto logico, come può essere la testata di un'istruzione IF. Ancora una volta entrambe le espressioni dovranno essere compatibili, in caso contrario verrà assegnato un messaggio di errore di sintassi.

Al crescere della complessità dei programmi vi accorgerete di come sia difficile tenere a mente le associazioni delle diverse variabili e costanti ai compiti specifici. Il compilatore vi può aiutare, mediante il controllo del < tipo >, a non confondere variabili destinate a scopi diversi. Per usufruire di questo controllo è naturalmente necessario usare speciali variabili di < tipo > ovunque ne sia il caso.

5. Tipi Sottocampo

Talvolta la logica di un algoritmo può richiedere che una variabile di programma possa assumere soltanto valori limitati all'interno dell'intero campo cui è associata come < tipo >.

Il *"campo di variabilità"* ("range") di una variabile si estende dal minimo al massimo valore che tale variabile può assumere. Un *"sottocampo"* ("subrange") è la parte di campo di una variabile cui appartengono tutti i valori fra un minimo e un massimo specificati.

Supponendo che il < tipo > LIVELLO sia stato dichiarato come nella sezione precedente:

```
TYPE
     etc.  etc.
  NON LAUREATO = MATRICOLA.. LAUREANDO;
     etc.  etc.
VAR
  SL: NONLAUREATO;
```

dichiara che la variabile SL è del < tipo > NONLAUREATO. Quindi l'assegnamento:

```
SL := TERZANNO
```

è corretto, mentre:

```
SL := LAUREATO
```

provocherà l'interruzione abnorme del programma.

A prima vista potrà sembrarvi controproducente il fatto che il sistema PASCAL fornisca un'altra causa per una fine anormale dei programmi. In realtà la ragione per cui un programma viene interrotto in maniera abnorme di fronte al tentativo di assegnare valori illegali ad una variabile di sottocampo è quella di rendere più facile la scoperta di errori logici. Capita abbastanza spesso che un programmatore si accorga, nello scrivere un programma, che una certa variabile può logicamente assumere soltanto alcuni valori. È facile poi dimenticare questa limitazione quando successivamente si assegna un valore alla variabile. La mancanza di un controllo esplicito sul valore può spesso condurre a nascondere errori molto difficili da trovare nella messa a punto del programma. L'uso di una variabile di sottocampo equivale a richiedere l'inserimento automatico di controlli sui valori da parte del compilatore. Quando il programma viene interrotto in maniera abnorme, il messaggio di errore generato dal sistema indicherà l'istruzione di programma che l'ha provocato e si potrà così porvi rimedio con uno sforzo minimo.

Oltre a permettere la dichiarazione di un sottocampo di un <tipo> SCALAR, PASCAL permette di dichiarare sottocampi di <tipi> INTEGER e CHAR. Per esempio:

```
TJPE SUBI = 1..10;
     SUBCH = 'I' .. 'N';
VAR
  IRV: SUBI;
  SV: 0..9;
  I: INTEGER;
  CHRV: SUBCH;
  CH: CHAR;
```

Quindi:

```
I := 11;
IRV := I;
```

provocherà un'interruzione anormale quando il sistema cercherà di assegnare a IRV il valore di I, poiché 11 è esterno al sottocampo SUBI cui IRV è stato associato nelle dichiarazioni.

Osservate i diagrammi sintattici di <tipo> e <tipo semplice> nell'Appendice E facendo particolare attenzione alla sintassi per la dichiarazione di un VETTORE (ARRAY). L'espressione di un <tipo semplice> usata come indice (valore di selezione)

di un vettore può ora essere vista sia come Scalare sia come < tipo > Sottocampo. Inoltre la dichiarazione delle < costanti > che definiscono i limiti inferiori e superiori di un VETTORE può essere vista come definizione dei limiti di un < tipo > SOTTOCAMPO. Nell'esempio:

```
TYPE LIMITI =–5 .. +5;

VAR
  A1 : ARRAY[–5 .. +5];
  A2 : ARRAY[LIMITI];
```

i vettori A1 e A2 hanno la stessa dimensione e gli stessi limiti inferiori e superiori. Si può verificare un'interruzione del programma per *"indice non valido"* quando:

a) si faccia riferimento ad una locazione di vettore non esistente, cioè esterna ai limiti dichiarati per il vettore.

b) si assegni ad una variabille di sottocampo un valore esterno ai limiti associati al suo < tipo >.

6. Insiemi (Sets)

Immaginiamo ora che al nostro ipotetico professore interessi confrontare le votazioni ottenute nel suo corso dagli studenti delle aree Artistiche, Umanistiche, di Scienze Naturali e Scienze Sociali. I flussi disponibili presso gli archivi universitari indicano però soltanto la facoltà, e non l'area di studio. Per fare il confronto richiesto il programma del professore deve quindi prima di tutto associare ogni studente ad un'AREA. Questo potrebbe essere fatto con un'istruzione CASE del tipo:

```
CASE FACOLTÀ OF
  ANTROPOLOGIA: AREA:=SCISOC;
  FISICAPPLIC, BIOLOGIA, CHIMICA: AREA:=SCINAT;
  COMUNICAZIONI: AREA:=SCISOC;
  TEATRO: AREA:=ARTI;
  ECONOMIA: AREA:=SCISOC;
  STORIA, LINGUE, LETTERE: AREA:=UMANIS;
  MATEMATICA: AREA:=SCINAT;
  MUSICA: AREA:=ARTI;
  FILOSOFIA: AREA:=UMANIS;
  FISICA: AREA:=SCINAT;
  SCIPOLITICHE, PSICOLOGIA, SOCIOLOGIA: AREA:=SCISOC;
```

```
    ARTIVIS: AREA:=ARTI;
END (*SELEZIONI*);
```

Le elaborazioni successive dipenderanno dal valore della <variabile> AREA.

È possibile fare la stessa cosa in un modo più semplice, usando gli Insiemi. Definiamo dapprima un <tipo> Scalare che chiameremo FACOLTÀ.

```
TYPE FACOLTÀ = (ANTROPOLOGIA, FISICAPPLIC, BIOLOGIA,
    CHIMICA, COMUNICAZIONI, TEATRO, ECONOMIA, STORIA,
    LINGUE, LETTERE, MATEMATICA, MUSICA, FILOSOFIA,
    FISICA, SCIPOLITICHE, PSICOLOGIA, SOCIOLOGIA, ARTIVIS);

VAR ARTISET, UMANISET, SCISOCSET, SCINATSET:
                    SET OF FACOLTÀ
```

A questo punto abbiamo quattro <variabili> di <tipo> SET OF FACOLTÀ. Ognuna contiene un elemento Booleano corrispondente ad ogni costante dichiarata nel <tipo> Scalare FACOLTÀ. Prima del lancio del programma tutti gli elementi di questi Insiemi sono non definiti. È necessario inizializzare le variabili Insieme prima di utilizzarle, così come è necessario inizializzare ogni altra variabile assegnandole un valore. Per fare questo si usano istruzioni del tipo:

```
ARTISET : = [TEATRO, MUSICA, ARTIVIS]
```

dove le parentesi sul lato destro includono uno speciale genere di <espressione> noto come *"costruttore"* di Insieme. La sua sintassi è inserita nella definizione di <fattore> nei diagrammi dell'appendice E. Questa istruzione di assegnamento assegna un valore equivalente al Booleano TRUE ai tre elementi di ARTISET corrispondenti alle FACOLTÀ incluse nel costruttore. A tutti gli altri elementi di ARTISET viene assegnato un valore equivalente al booleano FALSE. Se abbiamo una variabile dichiarata di <tipo> FACOLTÀ, per esempio:

```
VAR FAC: FACOLTÀ
```

tale <variabile> può assumere un valore uguale ad una qualsiasi delle costanti viste nella dichiarazione di TJPE di FACOLTÀ. Possiamo quindi usare un controllo del tipo:

```
IF FAC IN ARTISET THEN <istruzione>
```

piuttosto che scrivere il controllo equivalente:

```
IF (FAC=TEATRO) OR (FAC=MUSICA) OR (FAC=ARTIVIS)
    THEN <istruzione>
```

L'operatore *"IN"* verifica se la <variabile scalare> a sinistra è un *"membro"* del valore contenuto nella <variabile set> a destra.

Sono anche disponibili operazioni che permettono di combinare due o più <variabili set> dichiarate associate allo stesso <tipo> in una <espressione set>. Se ne dà di seguito qualche esempio:

```
VAR S1,S2,S3,S4: SET OF FACOLTÀ;

S1 := [LETTERE] + ARTISET;

S2 := [COMUNICAZIONI, TEATRO, ECONOMIA];

S3 := S1 * S2

S4 := S2 - S1
```

Il valore assegnato a S1 è il SET

```
[TEATRO,LETTERE,MUSICA,ARTIVIS]
```

L'operatore "+", usato su Insiemi, produce un nuovo SET contenente un elemento TRUE per ogni elemento TRUE alla sua sinistra OR alla sua destra. Il risultato è detto l'*"unione"* dei due Insiemi.

Il valore assegnato a S3 è il SET

```
[TEATRO]
```

poiché l'operatore "*", usato su Insiemi, produce un nuovo SET contenente come elementi TRUE soltanto quelli corrispondenti ad elementi TRUE nell'espressione 'insieme', a sinistra AND in quella a destra. Il risultato è detto l'*"intersezione"* dei due insiemi concatenati dall'operatore "*".

S4 contiene come elementi TRUE tutti quelli che sono TRUE in S2 e FALSE in S1; cioè ad S4 è assegnato il valore

```
[COMUNICAZIONI, ECONOMIA]
```

Il risultato prodotto dall'operatore "–" è detto l'insieme "differenza". Per la sintassi correlata alle dichiarazioni di SET si vedano <tipo> e <tipo semplice> nell'Appendice E. La sintassi di <espressione>, <espressione semplice>, <termine> e <fattore> coinvolgono le operazioni appena descritte.

7. Programma di esempio CIBISETS

Il programma CIBISETS fornisce un semplice esempio dell'uso sia di variabili Scalari, sia di variabili set associate a tali Scalari. Si danno di seguito le segnalazioni che tale programma dovrebbe dare in uscita:

CIBISETS

ELEMENTO	S1	S2	S3	+	*	–
MELA	T	T	F	T	F	T
BANANA	F	F	F	F	F	F
CAROTA	F	F	T	T	F	F
FAGIOLO	F	F	T	T	F	F
UVA	F	F	F	F	F	F
PANINO	F	F	F	F	F	F
PATATA	F	F	T	T	F	F
POMODORO	F	T	T	T	T	F
PERA	T	T	F	T	F	T
ARANCIA	T	T	F	T	F	T

Il programma inizializza dapprima gli insiemi S1, S2 e S3 alle linee 33, 34, 35. Visualizza poi gli elementi memorizzati in ciascun insieme, assieme all'unione, intersezione e differenza di S2 e S3 così come sono passati alla procedura di visualizzazione TORF ("T OR F"). Vi consigliamo di controllare ogni linea di questa tabella per essere sicuri di aver capito come ogni entrata acquisti il valore mostrato.

Si noti l'uso dei valori costanti dichiarati nelle linee 2 e 3 come costanti di selezione per l'istruzione CASE nella procedura ASSEGNANOME. Si noti anche l'uso dei <tipi> dichiarati CIBO e CS nelle dichiarazioni dei parametri di ASSEGNANOME e TORF.

```
1: PROGRAMMA CIBISETS;
2: TYPE CIBO=(MELA, BANANA, CAROTA, FAGIOLO, UVA, PANINO,
3:                PATATA, POMODORO, PERA, ARANCIA);
4:    CS=SET OF CIBO;
```

```
 5: VAR
 6:    S1,S2,S3: FS;
 7:    C: CIBO;
 8:    U,I,D: CHAR;
 9: PROCEDURE ASSEGNANOME(N:CIBO);
10: BEGIN
11:    CASE N OF
12:       MELA: WRITE('MELA ');
13:       BANANA: WRITE('BANANA');
14:       CAROTA: WRITE('CAROTA');
15:       FAGIOLO: WRITE('FAGIOLO ');
16:       UVA: WRITE('UVA ');
17:       PANINO: WRITE('PANINO');
18:       PATATA: WRITE('PATATA');
19:       POMODORO: WRITE('POMODORO');
20:       PERA: WRITE('PERA ');
21:       ARANCIA: WRITE('ARANCIA');
22:    END (*SELEZIONI*);
23: END (*ASSEGNANOME*);
24:
25: PROCEDURE TORF(C:CIBO; S:CS);
26: BEGIN
27:    IF C IN S THEN WRITE(' T ')
28:              ELSE WRITE(' F ');
29: END (*T OR F*);
30:
31: BEGIN (*PROGRAMMA PRINCIPALE*)
32:    WRITELN('CIBISETS'); WRITELN;
33:    S1:=[MELA,PERA,ARANCIA];
34:    S2:=S1 + [POMODORO];
35:    S3:=[CAROTA,FAGIOLO,PATATA,POMODORO);
36:    WRITELN('ELEMENTO ', ' S1 ', ' S2 ', ' S3 ' , ' + ',
37:            ' * ', ' – ');
38:    FOR C:=MELA TO ARANCIA DO
39:       BEGIN
40:          WRITELN; ASSEGNANOME(C); WRITE(' ');
41:          TORF(C,S1); TORF(C,S2); TORF(C,S3);
42:          TORF(C, S2+S3); (*unione*)
43:          TORF(C, S2*S3); (*intersezione*)
44:          TORF(C, S2–S3); (*differenza*)
45:          WRITELN;
46:       END;
47: END.
```

8. Programma tipo SETDEMO

Il programma legge un nome di studente, un'abbreviazione per la facoltà e un punteggio. La visualizzazione associata ad un testo in ingresso con 7 studenti è mostrata su una pagina separata. Al termine della fase di immissione, in cui EOF è segnalato dalla battitura di <EXT> (linea 35 della stampa) il programma visualizza il punteggio medio ottenuto in ogni area di studio.

Si noti che i vettori PUNTEGGI, CONT, MT e NOMEAREA sono tutti indicizzati con sottocampi di variabili scalari. Questo si può vedere sia nelle dichiarazioni (righe 12, 13, e 20 del programma), sia nell'uso di tali vettori come variabili indicizzate (dalla linea 55 fino alla 71 di INIZ).

Nella linea 89 usiamo la funzione interna *"successore"* SUCC per incrementare la <variabile> M che è di <tipo> FACOLTÀ. Per esemplificare tale uso,

```
M := SUCC(BIOLOGIA)
```

assegna a M il valore CHIMICA. Poiché ARTIVIS non ha alcun successore, il riciclo sulle linee 88 e 89 deve interrompersi quando M=ARTIVIS, cioè l'ultima costante dichiarata nelle FACOLTÀ. Altrimenti il programma terminerebbe in modo anomalo al tentativo di assegnare a M un successore di ARTIVIS non esistente. Una funzione analoga *"predecessore"* lavora in senso inverso, per esempio:

```
M := PRED(CHIMICA)
```

assegna a M il valore BIOLOGIA. Poichè ANTROPOLOGIA non ha alcun predecessore, un tentativo di usare PRED(ANTROPOLOGIA) provocherà una fine anomala.

```
 1: PROGRAMMA SETDEMO;
 2: TYPE
 3:    FACOLTÀ=(ANTROPOLOGIA, ARTIVIS, BIOLOGIA,
 4:       CHIMICA, COMUNICAZIONI, ECONOMIA,
 5:       FILOSOFIA, FISICA, FISICAPPLIC, LINGUE,
 6:       LETTERE, MATEMATICA, MUSICA, PSICOLOGIA,
 7:       SCIPOLITICHE, SOCIOLOGIA, STORIA, TEATRO);
 8:    FACSET=SET OF FACOLTÀ;
 9:    AREE=(ART, UMANIS, SCISOC, SCINAT);
10: VAR
11:    ARTISET, UMANISET, SCISOCSET, SCINATSET:FACSET;
12:    PUNTEGGI,CONT: ARRAY[ARTI..SCINAT] OF INTEGER;
13:    TF: ARRAY[ANTROPOLOGIA..TEATRO] OF
14:                   PACKED ARRAY[1..4] OF CHAR;
```

```
15:    ABUF: PACKED ARRAY[1..4] OF CHAR;
16:    P,I,MAXI: INTEGER;
17:    CH:CHAR;
18:    F: FACOLTÀ;
19:    A: AREE;
20:    NOMEAREA: ARRAY[ARTI..SCINAT] OF STRING;
21:    NOME,S: STRING;
22:
23: PROCEDURE CONFER
24: BEGIN
25:    WRITELN;
26:    WRITELN(NOME, ' ':3, G, ' ':3, NOME AREA[A]);
27:    WRITELN;
28: END (*CONFER*);
29:
30: PROCEDURE ALLINEA;
31: VAR G:INTEGER;
32: BEGIN
33:    WRITELN; WRITELN('MEDIE:');
34:    FOR A:=ARTI TO SCINAT DO
35:       BEGIN
36:          WRITE(' ',NOMEAREA[A],' ');
37:          IF CONT[A]>0 THEN
38:             G:=10*PUNTEGGI[A] DIV CONT [A]
39:          ELSE
40:             G:=0;
41:          WRITELN(G DIV 10, '.', G MOD 10);
42:       END;
43: END (*ALLINEA*);
44:
45: PROCEDURE INIZ;
46: BEGIN
47:    ARTISET:=[ARTIVIS,MUSICA,TEATRO];
48:    UMANISET:=LETTERE,LINGUE,FILOSOFIA];
49:    SCISOCSET:=[ANTROPOLOGIA,COMUNICAZIONI,
50:                   ECONOMIA,PSICOLOGIA,SOCIOLOGIA];
51:    SCINATSET:=[BIOLOGIA,CHIMICA,FISICA,
52:                   FISICAPPLIC,MATEMATICA];
53:
54:    (*inizializziamo ora TF con le abbreviazioni*)
54:  TF[ANTROPOLOGIA]:='ANTR';       TF[ARTIVIS]:='AVIS';
56:  TF[BIOLOGIA]:='BIOL';           TF[CHIMICA]:='CHIM';
57:  TF[COMUNICAZIONI]:='COM';       TF[ECONOMIA]:='ECON';
```

```
 58:  TF[FILOSOFIA]:='FIL';                TF[FISICA]:='FIS';
 59:  TF[FISICAPPLIC]:='FAPP';             TF[LINGUE]:='LING';
 60:  TF[LETTERE]:='LETT';                 TF[MATEMATICA]:='MAT';
 61:  TF[MUSICA]:='MUS';                   TF[PSICOLOGIA]:='PSIC';
 62:  TF(SCIPOLITICHE]:='SPOL';            TF[SOCIOLOGIA]:='SOC';
 63:  TF[STORIA]:='STOR';                  TF[TEATRO]:=TEAT';
 64:
 65:    (*inizializziamo i nomi delle aree*)
 66:    NOMEAREA[ARTI]:='ARTI';
 67:    NOMEAREA[UMANIS]:='SC UMANISTICHE';
 68:    NOMEAREA[SCISOC]:='SC SOCIALI';
 69:    NOMEAREA[SCINAT]:='SC NATURALI';
 70:    FOR A:=ART TO SCINAT DO
 71:      BEGIN PUNTEGGI[A]:=0; CONT[A]:=0; END;
 72:  END (*INIZ*);
 73:
 74:  BEGIN (*PROGRAMMA PRINCIPALE*);
 75:    INIZ;
 76:    WHILE NOT EOF DO
 77:    BEGIN
 78:      WRITE('NOME:');
 79:      READ(NOME);
 80:      WRITE('FACOLTÀ:');
 81:      READ(S);
 82:      ABUF:=' ';
 83:      IF LENGTH (S)>4 THEN MAXI:=4
 84:                      ELSE MAXI:=LENGTH(S);
 85:      FOR I:1 TO MAXI DO ABUF[I]:=S[I];
 86:      (*cerca l'abbreviazione corrispondente alla facoltà*)
 87:      M:=ANTROPOLOGIA;
 88:      WHILE (TF[M]< > ABUF) AND (M<TEATRO) DO
 89:        M:=SUCC(M); (*in M il successore di M*)
 90:      IF M IN (SCISOCSET + SCINATSET) THEN
 91:        BEGIN
 92:          IF M IN SCISOCSET THEN A:=SCISOC
 93:                            ELSE A:=SCINAT;
 94:        END ELSE
 95:          IF M IN ARTISET THEN A:=ARTI
 96:                          ELSE A:=UMANIS;
 97:      WRITE('PUNTEGGIO:'); READ (P); READ(CH);
 98:      PUNTEGGI[A]:=PUNTEGGI[A]+P;
 99:      CONT[A]:=CONT[A]+1;
100:      CONFER;
```

```
101:    END (*WHILE*);
102:    ALLINEA;
103: END.
```

```
 1: Visualizzazioni associate al programma SETDEMO
 2:
 3: NOME:Spini, Rosa
 4: FACOLTÀ:ECON
 5: PUNTEGGIO:85
 6: Spini, Rosa 85 SC SOCIALI
 7:
 8: NOME:Debelli, Maria
 9: FACOLTÁ:PSIC
10: PUNTEGGIO:80
11: Debelli,Maria 80 SC SOCIALI
12:
13: NOME:Luzzi,Giulia
14: FACOLTÁ:FIS
15: PUNTEGGIO:88
16: Luzzi,Giulia 88 SC NATURALI
17:
18: NOME:Rossi,Paolo
19: FACOLTÀ:AVIS
20: PUNTEGGIO:92
21: Rossi,Paolo 92 ARTI
22:
23: NOME:Bella,Annalisa
24: FACOLTÁ:LETT
25: PUNTEGGIO:75
26: Bella,Annalisa 75 SC UMANISTICHE
27:
28: NOME:Turi,Pietro
29: FACOLTÁ:SPOL
30: PUNTEGGIO:95
31: Turi,Pietro 95 SC UMANISTICHE
32:
33: NOME:Bardotti,Brigida
34: FACOLTÁ:CHIM
35: PUNTEGGIO:77<ETX>
36: Bardotti,Brigida 77 SC NATURALI
37:
38: MEDIE:
39: ARTI 92.0
```

```
40: SC UMANISTICHE 85.0
41: SC SOCIALI 82.5
42: SC NATURALI 82.5
```

L'effetto dell'istruzione IF nidificata da linea 90 fino a linea 96 è simile a quello dell'istruzione CASE mostrata all'inizio della Sezione 6. Determiniamo dapprima se M appartiene ad uno dei due insiemi scientifici, in caso affermativo si determina poi qual è l'insieme interessato. In caso contrario si suppone che M appartenga all'insieme artistico o umanistico e il controllo di riga 95 determina a quale dei due. Questo metodo di ripartire l'istruzione IF è leggermente più efficiente come tempo di elaborazione di quanto lo sarebbe stato il seguente.

```
IF M IN ARTISET THEN A:=ARTI
  ELSE IF M IN UMANISET THEN A:=UMANIS
    ELSE IF M IN SCINATSET THEN A:=SCINAT
      ELSE A:=SCISOC;
```

Si noti l'uso della <variabile >A di <tipo > scalare AREE per controllare le istruzioni FOR nelle linee 34 e 70.

9. Uso di insiemi con caratteri

In molte applicazioni gli insiemi forniscono un mezzo particolarmente utile per lavorare con stringhe di caratteri. Per esempio, prendiamo nuovamente in considerazione il programma di prova DEVOCALIZZA del capitolo 7 Sezione 10. Il complicato controllo nelle righe 14 e 15 di quel programma, cioè:

```
IF (CH< >'A') AND (CH< >'E') AND (CH< >'I')
   AND (CH< >'0') AND (CH< >'U') THEN
```

può essere sostituito da:

```
IF NOT (CH IN ['A','E','I','O','U']) THEN
```

Allo stesso modo potremmo avere

```
VAR VOCALI,CONSONANTI,LETTERE: SET OF CHAR;
```

e

```
VOCALI :=['A','E','I','O','U'];
CONSONANTI:=['a' .. 'Z'] - VOCALI;
```

in cui l'insieme costruttore nella seconda istruzione di assegnamento inserisce tutte le lettere, dalla 'A' alla 'Z' in un unico insieme, le vocali vengono poi separate per lasciare soltanto le consonanti. Potremmo anche costruire un insieme di tutte le lettere, includendo sia le maiuscole che le minuscole

```
LETTERE :=['A'..'Z', 'a'..'z']
```

Si può anche controllare se due insiemi sono uguali o diversi:

```
IF S1 = S2 THEN..
IF S2 <> S3 THEN..
```

Infine si può anche controllare se un insieme è *incluso* in un altro. Questo controllo può essere reso più chiaro se si fa riferimento al programma di prova CHARSETS. Nella riga 21, la frase

```
IF VOCALI <= LETTERE
```

controlla se l'insieme delle vocali è incluso nell'insieme delle lettere. La linea 3 della stampa mostra che il risultato di tale controllo è TRUE. Comunque, alla riga 23, il controllo era fatto al contrario, cioè

```
IF VOCALI >= LETTERE
```

e verifica se LETTERE è incluso in VOCALI. Come mostrato nella riga 4 di stampa, il risultato di tale controllo è stato FALSE. I due controlli per l'inclusione di insiemi sono quindi simmetrici, avremmo cioè potuto usare

```
IF LETTERE >= VOCALI
```

e il risultato del confronto sarebbe stato TRUE.

```
 1: PROGRAMMA CHARSETS;
 2: TYPE SOFCH = SET OF CHAR;
 3: VAR LETTERE, CIFRE, VOCALI, CONSONANTI
 4:                 ALFA, SPECIALE:SOFCH;
 5:
 6: PROCEDURE MOSTRA(S:SOFCH);
 7: VAR CH:CHAR;
 8: BEGIN
 9:   FOR CH:='0' TO'_' DO
10:     IF CH IN S THEN WRITE(CH) ELSE WRITE(' ');
11:     WRITELN:
```

```
12: END (*MOSTRA*);
13:
14: BEGIN
15:   LETTERE:=['A'...'Z'];
16:   VOCALI:=['A','E','I','O','U'];
17:   CONSONANTI:=LETTERE - VOCALI;
18:   CIFRE:=['0'...'9'];
19:   ALFA:=LETTERE + CIFRE;
20:   SPECIALE:=[' '..'_'] - LETTERE - CIFRE;
21:   IF VOCALI <= LETTERE THEN
22:     WRITELN('VOCALI <= LETTERE');
23:   IF NOT(VOCALI >= LETTERE) THEN
24:     WRITELN('NOT (VOCALI >= LETTERE');
25:   WRITELN;
26:   MOSTRA(LETTERE);
27:   MOSTRA(VOCALI);
28:   MOSTRA(CONSONANTI);
29:   MOSTRA(CIFRE);
30:   MOSTRA(ALFA);
31:   MOSTRA(SPECIALE);
32: END.
```

```
 1: Visualizzazione associata al programma CHARSETS
 2:
 3: VOCALI <= LETTERE
 4: NOT (VOCALI >= LETTERE)
 5:
 6:                          ABCDEFGHIJKLMNOPQRSTUVWXYZ
 7:                          A    E    I       O       U
 8:                            BCD  FGH JKLMN  PQRST  VWXYZ
 9: 0123456789
10: 0123456789               ABCDEFGHIJKLMNOPQRSTUVWXYZ
11:              :;<=>?                                  [/]^_
```

Le altre linee stampate dal programma CHARSETS mostrano i valori dei vari insiemi inizializzati tra linea 15 e linea 20 del programma. A causa delle limitazioni di larghezza nella pubblicazione di questo libro, sono mostrati soltanto alcuni dei caratteri speciali. Potrete stampare tutti i caratteri speciali, i primi 64 dei 95 visualizzabili nell'insieme ASCII (American Standard Code for Information Interchange) cambiando '0' con '' alla riga 9 del programma. I rimanenti 31 caratteri possono essere stampati separatamente usando

```
FOR CH:=CHR(96) TO CHR(126) DO...
```

In questo caso noi usiamo la funzione CHR di trasferimento interi-caratteri a causa dei problemi legati all'insieme di caratteri di stampa di questo libro.

ESERCIZIO 9-1:
Dovendo usare il programma SETDEMO come descritto in questo capitolo, vi accorgereste che non è ben protetto contro certi errori possibili nella fase di battitura dei dati in ingresso. Il programma presentato potrebbe terminare in modo anomalo o fornire risultati non corretti in ognuno dei seguenti casi:

a) abbreviazione di FACOLTÀ non corretta o non contenuta nel vettore MT (tavola delle facoltà)

b) PUNTEGGIO immesso in modo non corretto e quindi esterno all'intervallo di accettabilità 0..100.

c) caratteri in ingresso per il NOME diversi dai caratteri alfabetici, più ',' e '.'.

Modificate il programma in modo che ciascuno di questi errori possa essere "recuperato". Generalmente, al riconoscimento di un errore, il programma dovrebbe visualizzare un messaggio d'errore, far suonare la segnalazione acustica dell'elaboratore, e quindi richiedere la ribattitura dello stesso dato. Dovreste inoltre modificare il programma in modo che nella lettura del punteggio accetti un qualsiasi carattere diverso da ['0'..'9', ' '] (incluso lo <spazio> generato da <RET>). Controllate e mettete a punto il vostro programma modificato usando dati in ingresso atti a controllarlo in tutte le situazioni d'errore sopra citate.

ESERCIZIO 9.2:
Il programma QUADRATURA di Capitolo 8 Sezione 8 non controlla la correttezza del codice di servizio necessario a identificare la colonna in cui dovrebbe essere introdotto l'ammontare di transazioni. Si modifichi il programma affinché visualizzi un messaggio d'errore a fronte di un codice non corretto introdotto da tastiera, e richieda di ricominciare la battitura da tastiera. Controllate e mettete a punto il programma modificato verificando che i vostri cambiamenti sono corretti.

ESERCIZIO 9.3:
Supponete di essere assunti dall'ufficio per le risorse energetiche del vostro stato per scrivere programmi che controllino tutti i contratti e i contratti pianificati per lo sviluppo dell'energia nel vostro stato. È ne-

cessario tenere registrazioni per le seguenti risorse e tecniche per generare energia utilizzabile:

olii, gas naturali, carbone, potenza idroelettrica, potenza geotermica, potenza solare, vento, legno, rifiuti urbani, fissione nucleare, fusione nucleare, raccolte affioramenti speciali, altri.

Il lavoro è complicato perché le prime tre sono usate anche come risorse per le industrie plastiche e petrolchimiche, le risorse nucleari sono controverse e ritenute pericolose da alcune persone, le risorse idroelettriche, solari e il vento sono fortemente dipendenti dal tempo e così via.

Scrivete e mettete a punto un programma che accetti come dati, un nome di 20 caratteri indicante la fonte di un contratto d'energia, un'abbreviazione per il tipo di energia, il numero di BTU (in bilioni) promessi per lo sviluppo ogni anno, e il prezzo (in miliardi di lire) per un numero non definito di risorse. Il programma dovrebbe tenere un totale dei numeri di BTU da tutte le risorse in ognuno dei raggruppamenti:

carburante fossile (olii, gas naturale, carbone)
nucleare (fissione e fusione)
risorse inesauribili (acqua, sole, vento, legno, rifiuti urbani, affioramenti)
carburante esauribile (olii, gas naturale, carbone, geotermale, fissione).

Dopo aver accettato una sequenza di dati in ingresso contenenti le informazioni specificate, il programma dovrebbe visualizzare il totale di BTU e costo in lire per ciascuna delle precedenti categorie. Dovrebbe inoltre raggruppare i contratti in ogni categoria elencando separatamente per nome ogni fornitore. Per controllare i dati, createvi dei vostri dati di prova assicurandovi che rappresentino tutte le diverse risorse energetiche e che siano usati almeno 5 diversi nomi di fornitori. Confrontate le uscite del programma con i risultati da voi calcolati a mano, per assicurarvi che i risultati siano corretti.

Nota: Questo esercizio e il successivo fanno riferimento a elaborazioni dati che normalmente richiederebbero l'utilizzo di supporti di registrazione permanenti, come dischi magnetici, per memorizzare flussi di dati che devono essere elaborati dopo essere stati raccolti. Sfortunatamente i dettagli di utilizzo di tali supporti vanno oltre gli scopi di questo libro. Per questa ragione, sarà pratico controllare i programmi svilup-

pati soltanto con pochi dati campione che rappresentino i dati che potrebbero venire acquisiti in una situazione reale.

ESERCIZIO 9.4:
La maggior parte delle comunità urbane (negli U.S.A.!) hanno ormai uffici di pianificazione regionali simili al San Diego Comprehensive Planning Organization (CPO). Una delle principali funzioni del CPO è quella di archiviare i diversi modi in cui ogni sezione della comunità occupa e utilizza i suoi territori. Tali informazioni sono necessarie ogni volta venga richiesta l'approvazione di un nuovo edificio o di uno sviluppo industriale, nella pianificazione di cambiamenti nelle scuole o nei trasporti, nei tentativi di migliorare le organizzazioni di polizia e vigili del fuoco, nell'affrontare l'inquinamento dell'aria e dell'acqua, nei preparativi per affrontare calamità naturali come terremoti e allagamenti, nella pianificazione di nuovi parchi e aree di ripopolazione e così via... Uno dei più grossi compiti che il CPO di San Diego ha dovuto affrontare negli ultimi anni è stato quello di trovare un posto per un nuovo aereoporto internazionale. L'attuale aereoporto è vicino al centro della città e sta diventando sovraffollato.

Per portare a termine tale missione il CPO mantiene su un elaboratore dei record che "riproducono" la regione suddivisa in piccole sezioni.
I confini di tali sezioni sono definiti da confini naturali come alvei, rilievi rocciosi, la costa dell'Oceano Pacifico e da confini creati dall'uomo come suddivisioni di case, i maggiori centri di vendita, parchi industriali, aree agricole e così via. La registrazione riguardante ogni sezione include una caratterizzazione di tale sezione (effettivamente) espressa come un Insieme. I membri dell'insieme classificano la sezione secondo categorie del tipo:

densità di popolazione (rurale, suburbana, densa)
industrie (nessuna, agricola, leggera, pesante)
inquinamento dell'aria (leggero, moderato, notevole, eccessivo)
percentuale di delitti (bassa, moderata, notevole, eccessiva)

Si supponga che le sezioni della regione siano identificate sulle carte semplicemente tramite interi arbitrari nell'intervallo 1.. <numero di sezioni>. Si scriva e si metta a punto un programma che archivi fino a 20 registrazioni di sezioni della regione (un numero basso per scopi di controllo). Le registrazioni devono essere tenute come un vettore

di insiemi i cui membri includano, come minimo, tutte le possibili categorie sopra citate.
Nella fase di immissione l'utente dovrebbe essere invitato ad indicare quale sezione è applicabile ad ogni nuovo dato, e ad indicare poi quali categorie della registrazione selezionata dovrebbero essere cambiate. Prima di terminare il programma dovrebbe visualizzare una sintesi dei dati accumulati per tutte le sezioni. Per simulare il tipo di analisi per il quale sono spesso usati questi dati il programma dovrebbe identificare quelle sezioni dove esiste:

a) densità popolazione suburbana AND eccessivo inquinamento atmosferico

b) industria leggera OR alta densità di popolazione entrambe combinate con alta criminalità.

Costruite i vostri propri dati di verifica, inserendo le caratteristiche rappresentative in ciascuna delle 20 sezioni usate per controllare il programma. Preparate i vostri dati per verificare sia la condizione TRUE che FALSE per le categorie combinate a) e b).

CAPITOLO 10

STRUTTURE DATI BASE III
RECORDS

1. Obiettivi

Questo capitolo introduce il concetto di gestione di vari elementi correlati, ma di < tipo > diverso, come di un'unica entità detta "*record*". Tale concetto è fondamentale nell'elaborazione di dati amministrativi, nella tecnica dello sviluppo della programmazione, e in altri campi elaborativi connessi.

1a. Apprendimento a dichiarare < variabili > record e < tipi > record.

1b. Gestione di elementi all'interno di records individuali.

1c. Ove consigliabile, uso di variabili record come unità complete, senza preoccuparsi degli elementi che le costituiscono.

1d. Uso di vettori di variabili records e records come elementi di altri records.

1e. Modifiche di un programma di media complessità introducendo l'utilizzo di records.

2. Premessa

È frequente l'occasione di gestire molti dati strettamente correlati di < tipo > diverso: ad esempio i dati di ingresso del programma QUADRATURA del capitolo 8 erano divisi in gruppi di tre elementi, ove ogni gruppo era visto come un "messaggio" riguardante una singola transazione. Nell'esempio SPORTPUNTI2 ogni gruppo consisteva di un nome di persona seguito da 1 fino a 10 valori interi rappresentanti i punteggi ottenuti dalla stessa persona. Memorizzammo i nomi in un vettore NOMI e i punteggi in un altro vettore PUNTEGGI. Dopo aver memorizzato tali vettori, il solo modo di associare un nome ai punteggi corrispondenti fu quello di sfruttare lo stesso

valore di indice di riga per le righe di entrambi i vettori. In altre parole il nome in NOMI[RIGA] corrispondeva ai punteggi in PUNTEGGI[RIGA]. Se la dimensione dei vettori è grande e vi sono più di due elementi in corrispondenza, è facile commettere errori nello scrivere programmi con relazioni di questo tipo.

Per evitare queste confusioni PASCAL vi permette di dichiarare una <variabile> composta detta "*record*" costituita da due o più elementi il cui <tipo> può essere mischiato. Potete pensare ad una <variabile> RECORD come agli elementi di una scheda perforata o di una scheda di un catalogo librario. Generalmente ogni scheda contiene molti valori specifici di certi elementi correlati. Ad esempio una scheda potrebbe contenere il valore di uno Scalare indicante il livello di classe di uno studente, oltre al nome dello studente sotto forma di stringa, e una classe sotto forma di costante intera. Quando tenete in mano una scheda, è chiaro che i dati perforati in tale scheda sono mutuamente associati. Una <variabile> RECORD fornisce un modo per memorizzare assieme valori di elementi nella memoria dell'elaboratore in modo da mantenere le correlazioni fra gli elementi contenuti nel record.

In alcuni casi troverete preferibile fare riferimento ad elementi specifici all'interno di un record, in altri sarà conveniente considerare l'intero record come un'unica entità. PASCAL fornisce la possibilità di fare riferimenti con entrambi i metodi; anche COBOL e PL/1 forniscono tali possibilità. Gli altri linguaggi comuni da noi menzionati non forniscono la possibilità di gestire records. I records sono ora considerati essenziali per strutturare elementi nell'elaborazione di dati amministrativi e nella progettazione di grossi "programmi di sistema" come i compilatori per linguaggi di programmazione. I records potrebbero anche essere più comunemente usati in applicazioni numeriche di scienze e matematica se fossero più facilmente disponibili possibilità adatte.

3. Programma TIPO CLASSDATI

Dopo quanto detto prima, il modo più semplice per introdurre l'uso dei Records è quello di iniziare direttamente con il programma di prova CLASSDATI. Le regole sintattiche fondamentali sono mostrate in Figura 10-1 e 10-2. La versione iniziale di tale programma è costruita per illustrare l'uso di una semplice <variabile> RECORD chiamata STUDENTE e dichiarata nelle linee da 5 a 10 del programma. Si sarebbe potuto scrivere il programma senza usare una <variabile> RECORD e senza incontrare grosse difficoltà, ma scriverlo con la struttura record ci permetterà di vedere le caratteristiche del record in stadi semplici.

La sintassi mostrata in Figura 10-1 per la dichiarazione di <tipo record> è semplificata rispetto alla sintassi completa di PASCAL mostrata nell'Appendice E. La sintas-

si più completa permette una clausola CASE che consente la dichiarazione di *"varianti"* per lo stesso record. Questo è un concetto utile per applicazioni che richiedano dispositivi di memorizzazione esterna, come dischi magnetici. L'uso del concetto di variante nel record va però oltre lo scopo di questo libro.

```
 1: PROGRAMMA CLASSDATI;
 2: VAR I,FCONT,MCONT: INTEGER;
 3:   FSOMMA,MSOMMA:REAL;
 4:   CH:CHAR;
 5:   STUDENTE;
 6:     RECORD
 7:       NOME: STRINGA[20];
 8:       PUNTI: REAL;
 9:       SESSO: (FEMMINA,MASCHIO)
10:     END (*STUDENTE*);
11:
12: PROCEDURE ACCUMULA;
13: BEGIN
14:   IF STUDENTE,SESSO=FEMMINA THEN
15:   BEGIN
16:     FSOMMA:=FSOMMA+STUDENTE.PUNTI;
17:     FCONT:=FCONT+1;
18:   END ELSE
19:   BEGIN
20:     MSOMMA:=MSOMMA+STUDENTE.PUNTI;
21:     MCONT:=MCONT+1;
22:   END;
23:   WRITELN;
24:   WRITE(STUDENTE.NOME,'':20-LENGTH(STUDENTE.NOME),
25:         '',STUDENTE.PUNTI:3:1,'');
26:   IF STUDENTE,SESSO=FEMMINA THEN WRITELN('F')
27:                             ELSE WRITELN('M');
28:   WRITELN;
29: END (*ACCUMULA*);
30:
31: PROCEDURE SCRIVIRISULTATI;
32:
33:  PROCEDURE DETTAGLIO(S:STRING; R:REAL; I:INTEGER);
34:   VAR DIVISORE:INTEGER;
35:   BEGIN
36:     IF I>0 THEN DIVISORE:=I ELSE DIVISORE:=1;
37:     WRITELN(S, ' ':15-LENGTH(S),
38:             R/DIVISORE:3:1,' ':15,I);
```

```
39:    END (*DETTAGLIO*);
40:
41: BEGIN
42:    WRITELN(' ':10,'MEDIA PUNTI',' ':5,
43:                    'CONT STUDENTI');
44:    WRITELN;
45:    DETTAGLIO('FEMMINE, FSOMMA, FCONT);
46:    WRITELN;
47:    DETTAGLIO('MASCHI', MSOMMA, MCONT);
48: END (*SCRIVI RISULTATI*);
49:
50: BEGIN (*PROGRAMMA PRINCIPALE*)
51:    FSOMMA:=0; MSOMMA:=0;
52:    FCONT:=0; MCONT:=0;
53:    WHILE NOT EOF DO
54:    BEGIN
55:       WRITE('NOME:');
56:       READ(STUDENTE,NOME);
57:       IF NOT EOF THEN
58:       BEGIN
59:          WRITE('PUNTI:');
60:          READ(STUDENTE.PUNTI);
61:          READ(CH); (*scarta <spazio> o <RET>*)
62:          WRITE('M/F:');
63:          READ(CH);
64:          IF CH IN ['F','f'] THEN
65:             STUDENTE.SESSO:=FEMMINA
66:          ELSE
67:             STUDENTE.SESSO:=MASCHIO;
68:          ACCUMULA;
69:       END;
70:    END (*WHILE NOT EOF* );
71:    WRITELN;
72:    SCRIVIRISULTATI;
73: END.
```

```
1: Visualizzazioni associate al programma
2:
3: NOME:Vicini,Cristina
4:    PUNTI:3.8
5:    M/F:f
6: Vicini,Cristina                    3.8   F
7:
```

```
 8: NOME:Trogu,Giovanni
 9:    PUNTI:2.4
10:    M/F:m
11: Trogu,Giovanni                2.4   M
12:
13: NOME:Torchiana,Noemi
14:    PUNTI:2.9
15:    M/F:f
16: Torchiana,Noemi               2.9   F
17:
18: NOME:Marino,Marina
19:    PUNTI:3.3
20:    M/F:f
21: Marino,Marina                 3.3   F
22:
23: NOME:Falossi,Giulio
24:    PUNTI:2.7
25:    M/F:m
26: Falossi,Giulio                2.7   M
27:
28: NOME:Origgi,Mauro
29:    PUNTI:3.5
30:    M/F:m
31: Origgi,Mauro                  3.5   M
32:
33: NOME:Dell'Amore,Maurizio
34:    PUNTI:3.2
35:    M/F:m
36: Dell'Amore,Maurizio           3.2   M
37:
38: NOME:<ETX>
39:
40:                    MEDIA PUNTI          CONT STUDENTI
41:
42: FEMMINE:              3.3                    3
43:
44: MASCHI:               3.0                    4
```

La <variabile> RECORD STUDENTE è dichiarata come composta da tre *"campi"* detti NOME, PUNTI, e SESSO. I tre elementi dell'informazione saranno memorizzati in locazioni adiacenti nella memoria dell'elaboratore. Se fate un diagramma di quest'area di memoria apparirà come una mappa contenente diversi "campi", da cui la

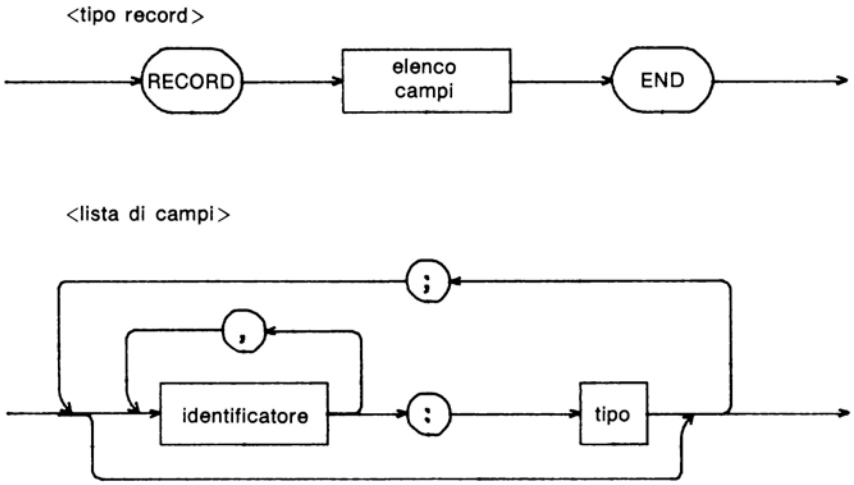


Figura 10-1

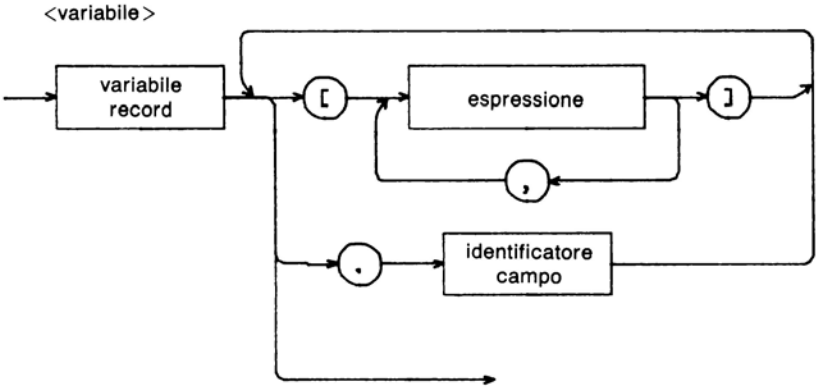


Figura 10-2

terminologia. Ogni campo è formato da una <variabile> e una dichiarazione di tipo sulla linea corrispondente della dichiarazione record. Il campo NOME è una stringa come quelle che avete visto per tutto il libro, tranne che in questo caso la dichiarazione restringe la lunghezza della STRING a 20 caratteri. La variabile SESSO è uno Scalare avente due possibili valori, FEMMINA e MASCHIO.

Il modo più diretto di fare riferimento a una variabile, campo di una <variabile>

RECORD è quello di unire il nome del record al nome del campo separandoli con un punto, ad esempio:

STUDENTE.RISULTATO

usato nelle linee 16, 20, 25 e 60 del programma. Poichè è possibile riutilizzare lo stesso nome campo all'interno di variabili records diverse, il compilatore richiede l'indicazione del nome record associato al nome della <variabile> campo.

La Figura 10-2 mostra la sintassi di <variabile> allargata a riferimenti a campi all'interno di un record. Si noti che nell'esempio visto prima NOME è un vettore (<variabile> STRING), ed è quindi possibile fare riferimento all'n-simo carattere di NOME nel modo seguente:

STUDENTE.NOME[N]

La sintassi vista in Figura 10-1 mostra che un campo può essere associato a qualunque <tipo>, ora anche al <tipo record>, ed è quindi chiaro che è possibile:

a) dichiarare un record contenente come campo un altro record.

b) dichiarare un vettore i cui elementi siano records.

c) dichiarare un record contenente un campo costituito da un vettore di records.

e così via. Le correlazioni fra i dati in grossi programmi possono talvolta diventare molto complesse. La sintassi di dichiarazione record è abbastanza generale e permette una larga varietà di relazioni fra i dati all'interno di strutture record. Da come si vede nella dichiarazione del record STUDENTE, le relazioni sono strutturate *ad albero*, cioè *gerarchiche*, proprio come le relazioni nel diagramma di struttura di un programma o nell'equivalente tabella di struttura.

Il programma CLASSDATI accetta tre elementi, richiedendo ciascuno, come si vede nella parte alta della pagina che mostra la visualizzazione associata a questo programma. Dopo aver registrato i tre elementi nei campi associati del record STUDENTE, la procedura ACCUMULA conferma l'immissione con una sola linea contenente tutti e tre gli elementi. Ancora una volta abbiamo evitato di imbrattare il programma con passi logici per renderlo meno suscettibile di interruzioni anomale per errori nella battitura del valore di PUNTI o M/F. Alla fine dell'immissione, segnalata dall'<EXT> nella linea 38 della visualizzazione, la procedura SCRIVIRISULTATI stampa una tabella sommaria che mostra i punteggi medi per i due sessi, e il numero di studenti di ogni sesso.

4. L'istruzione WITH

Lo scrivere programmi contenenti <variabili> RECORD può essere semplificato notevolmente con l'uso dell'istruzione PASCAL WITH. La sintassi relativa è mostrata in Figura 10-3 in una forma leggermente più esplicita di quella presentata nell'Appendice E.

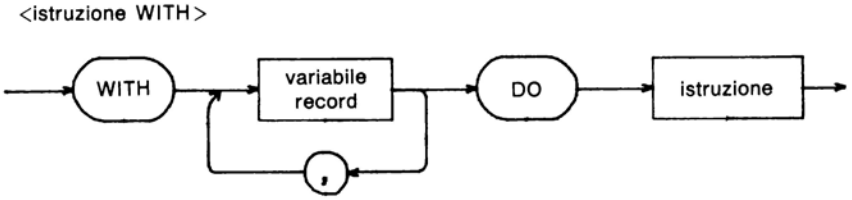


Figura 10-3

Mostriamo di seguito come può diventare la procedura ACCUMULA se si fa uso dell'istruzione WITH.

```
PROCEDURE ACCUMULA;
BEGIN
  WITH STUDENTE DO
    BEGIN
      IF SESSO=FEMMINA THEN
      BEGIN
        FSOMMA:=FSOMMA+PUNTI;
        FCONT:=FCONT+1;
      END ELSE
      BEGIN
        MSOMMA:=MSOMMA+PUNTI;
        MCONT:=MCONT+1;
      END;
      WRITELN;
      WRITE(NOME,' ':20-LENGTH(NOME)
                  ' ',PUNTI:3:1,' ');
      IFSESSO=FEMMINATHEN WRITELN('F')
                          ELSE WRITELN('M');
    END (*WITH*);
  WRITELN;
END (*ACCUMULA*);
```

Si confronti questa versione della procedura con la versione originaria inserita nel programma CLASSDATI. Si noti che nei riferimenti ai tre campi del record STUDENTE — NOME, PUNTI e SESSO — non è più necessario il riferimento esplicito a STUDENTE inserito come prefisso. Al suo posto, la frase "WITH STUDENTE DO" serve come prefisso a tutte le istruzioni successive, che in questo caso è un'istruzione composta (BEGIN ... END). L'effetto è logicamente equivalente, all'interno dell'istruzione controllata dalla clausola WITH, al porre "STUDENTE." all'inizio di ogni identificatore campo associato a STUDENTE. Non vengono toccati i riferimenti a variabili non associate a STUDENTE.

5. Programma di prova STURECORD

Scopo di questo programma è quello di illustrare l'uso di dichiarazioni TYPE connesse al < tipo > RECORD. Il programma, come stampato, esegue l'azione molto semplice di leggere, ripetutamente, nomi di persone nell'ordine dal cognome al nome, stampando poi il nome completo nell'ordine normale. Il < tipo > dichiarato INDIRIZZO non è usato nel programma presentato, ma è stato introdotto in previsione degli esercizi di questo capitolo.

Notate che potete usare un < tipo > precedentemente definito come parte della definizione di un altro < tipo >. Ad esempio il nuovo < tipo > ALFA è usato nel definire sia il < tipo > NOME che il < tipo > INDIRIZZO. Sia NOME che indirizzo sono usati nel definire STUREC. In questo modo potreste continuare ad aggiungere nuovi tipi addizionali a quelli vecchi. Tuttavia *non* sono permesse dichiarazioni *recursive* di < tipo >.

Le dichiarazioni nidificate di record mostrate in questo programma tipo hanno il vantaggio di semplificare la scrittura di strutture dati complesse. Permettono anche al compilatore di effettuare dei controlli più efficaci sul < tipo > come descritto nel capitolo 9. Lo svantaggio è che i riferimenti a campi interni a strutture record nidificate diviene più complesso.

Per esempio:

```
TABSTU[I].NOMESTU.NOME[3]
```

è un riferimento al terzo carattere nel campo NOME del campo NOMESTU all'interno del primo elemento del vettore TABSTU. Chiaramente vi stufereste abbastanza nel dover usare questa lunga sequenza di identificatori ogni volta che dovete usare un carattere di NOME. La via migliore per uscirne è l'uso dell'istruzione WITH, come è esemplificato nelle linee 49, 50 e 51 del programma tipo. (Non lasciatevi confondere dal fatto che NOME è sia un < tipo > che una < variabile >).

```
 1: PROGRAMMA STURECORD;
 2: CONST RIGHEVETT=10;
 3:
 4: TYPE ALFA=STRING[20];
 5:    NOME=
 6:       RECORD
 7:          COGNOME; ALFA;
 8:          NOME: ALFA;
 9:          INIZINT:CHAR
10:       END;
11:    INDIRIZZO=
12:       RECORD
13:          VIA: ALFA;
14:          CITTÀ: ALFA;
15:          STATO: PACKED ARRAY[1..2] OF CHAR;
16:          CODPOST: INTEGER
17:       END (*INDIRIZZO*);
18:    STUREC=
19:       RECORD
20:          NOMESTU: NOME;
21:          INDSTU: INDIRIZZO;
22:          PUNTI: INTEGER;
23:          SESSO: (FEMMINA, MASCHIO)
24:       END;
25:
26: VAR TABSTU: ARRAY[1..RIGHEVETT] OF STUREC;
27:    I: INTEGER;
28:
29:    PROCEDURE ACQUISDATI;
30: VAR RECNOME: NOME;
31: BEGIN
32:    WITH RECNOME DO
33:    BEGIN
34:       WRITE('COGNOME:');
35:       READ(COGNOME);
36:       WRITE('NOME:');
37:       READ(NOME);
38:       WRITE('INIZIALE SEC. NOME:');
39:       READLN(INIZINT);
40:    END (*WITH*);
41:    TABSTU[I].NOMESTU:=RECNOME;
42: END (*ACQUISDATI*);
43:
```

```
44: BEGIN (*PROGRAMMA PRINCIPALE*)
45:   I:=1
46:   WHILE NOT EOF AND (I<RIGHEVETT) DO
47:   BEGIN
48:     ACQUISDATI;
49:     WITH TABSTU[I].NOMESTU DO
50:       WRITELN(NOME,' ',INIZINT,'. ',
51:                     COGNOME);
52:     I:=I+1;
53:     WRITELN;
54:   END;
55: END.
```

```
 1: Visualizzazioni associate al programma STURECORD
 2:
 3: COGNOME:Bianchi
 4: NOME:Mario
 5: INIZIALE SEC. NOME:L
 6: Mario L. Bianchi
 7:
 8: COGNOME:Villa
 9: NOME:Amedeo
10: INIZIALE SEC. NOME:F
11: Amedeo F. Villa
12:
13: COGNOME:Bevilacqua
14: NOME:Renzo
15: INIZIALE SEC. NOME:C
16: Renzo C. Bevilacqua
17:
18: COGNOME:<ETX>
```

A questo punto avrete anche capito che, se voi doveste fare riferimento ad ogni campo dei due records, sarebbe molto complicato copiare i valori memorizzati in un record in un altro dello stesso <tipo>. PASCAL offre un modo semplice per eseguire questo compito concettualmente semplice; metodo che è mostrato in linea 41 del programma tipo. In questo caso sia TABSTU[I]. NOMESTU e RECNOME sono stati dichiarati dello stesso <tipo> NOME. Essi sono perciò *compatibili*, ed il compilatore fa si che il contenuto dell'intero record RECNOME sia assegnato al primo elemento di

TABSTU. Lo stesso genere di operazione di assegnamento è possibile tra due qualsiasi variabili dello stesso <tipo>. Data, per esempio, la dichiarazione:

```
VAR A,B: ARRAY [1..10] OF
           ARRAY [0..99) OF
             RECORD
               R:  REAL;
               I:  INTEGER;
               S:  STRING[10]
             END;
```

è corretto usare sia l'assegnamento semplice:

```
A := B
```

che

```
B[6] := A[3]
```

Certe volte sono possibili anche assegnamenti tra variabili strutturate che non sono dello stesso <tipo> ma sono nondimeno *compatibili*. I dettagli su cosa sia la compatibilità sono più complessi di quanto tolleri una descrizione in questo testo. Nel caso di necessità potete sperimentare o lasciar decidere al compilatore se le vostre variabili strutturate sono compatibili.

Considerate la dichiarazione della linea 2 del programma:

```
CONST RIGHEVETT = 10;
```

Questa è una dichiarazione di *"costante"*; potete trovare la sintassi della dichiarazione nell'appendice E, sotto <blocco>. Una dichiarazione di costante associa un identificatore con un valore costante alla destra del segno uguale ('='). Perciò il compilatore, ogniqualvolta nel testo del programma compare l'identificatore, lo sostituirà con quel valore. Poiché lo stesso valore costante appare in parecchi punti del programma, in questo caso con riferimento all'ampiezza dell'intervallo dell'indice del vettore, è possibile cambiarlo in tutti i punti semplicemente cambiando la dichiarazione *"CONST"* e ricompilando.

Quando si lavora con programmi grossi, o con programmi nei quali le stesse costanti dimensionali appaiono molte volte, si può evitare molta perdita di tempo nella fase di messa a punto usando identificatori di costanti dichiarati piuttosto che costanti esplicite all'interno. Se c'è da fare un cambiamento, lo si può fare molto velocemente cambiando la dichiarazione CONST. Potete anche, abbastanza facilmente,

per mezzo del comando F(ind) dell'editor trovare tutti i punti del programma in cui compare la costante. Si noti che le regole sintattiche richiedono che la dichiarazione CONST compaia prima della dichiarazione TYPE all'interno di un <blocco>. È lecito dichiarare un identificatore di <costante> che abbia un valore che è una costante intera, reale o stringa. Più avanti sono dati altri esempi nei programmi tipo.

ESERCIZIO 10.1:
Una delle occupazioni principali di molta gente che lavora a tempo pieno in programmazione è quella di modificare programmi che sono già stati scritti ed usati per un certo tempo. Questo esercizio richiede che modifichiate il programma STURECORD nel seguente modo:

a) Aggiungete la possibilità di leggere l'indirizzo degli studenti, includendo dei valori per tutti e quattro i campi nella dichiarazione di INDIRIZZO.

b) Aggiungete la possibilità di "catturare" valori per PUNTI e SESSO.

c) Aggiungete la possibilità di cambiare un qualsiasi campo in un record posto precedentemente in TABSTU. Deve quindi essere permesso all'utente di indicare, tramite il numero dell'elemento (indice), quale elemento deve essere cambiato.

d) Aggiungete una procedura per sostituire le linee 50 e 51 della versione stampata del programma con una visualizzazione armoniosa di tutti i campi in un record di <tipo> RECSTU. Usate questa procedura per fare l'eco della conferma dei nuovi valori del record, come nelle righe 6, 11 e 16 della più semplice visualizzazione ottenuta con STURECORD. La procedura dovrebbe anche essere usata per verificare lo stato corrente di un record prima che sia modificato come nel punto (c).

I record di questo tipo sono solitamente memorizzati su disco per essere poi elaborati ogni volta che lo si desidera. Sfortunatamente le complicazioni associate con il salvataggio di flussi su disco sono un po' oltre gli scopi di questo libro.

ESERCIZIO 10.2:
Come parecchi (perversi) responsabili, vogliamo ora modificare il programma STURECORD in modo diverso. Cambiatelo in modo che

la procedura ACQUISDATI memorizzi il codice postale e la sigla della provincia dell'indirizzo degli studenti nello stesso record in TABSTU come il nome. Cambiate la dichiarazione cosicchè codici postali con valore maggiore di 32767 possano essere memorizzati senza che il programma abortisca su un microelaboratore a 16 bits.

ESERCIZIO 10.3:
Usate STURECORD come base di partenza per un programma che sarà usato per calcolare il voto finale di uno studente per un corso con 8 interrogazioni, un esame intermedio ed uno finale. Per semplicità tutti e dieci i voti dovrebbero avere il medesimo peso. Sarà quindi semplicemente necessario sommare tutti i voti e dividere poi per 10. Eliminate tutti i riferimenti a informazioni su indirizzo e sesso. Scrivete il programma in modo che possa memorizzare tutti i voti in un vettore TABSTU opportunamente modificato; dovrebbe anche permettere modifiche per immissioni errate. Calcolate il voto medio finale e visualizzatelo con il nome dello studente a fine programma.

CAPITOLO 11

L'ISTRUZIONE GOTO

1. Obiettivi

L'istruzione GOTO fa sì che il controllo di programma passi da una locazione ad un'altra etichettata e dichiarata esplicitamente nell'istruzione stessa. Generalmente tale istruzione dovrebbe essere usata soltanto come un modo per alterare il flusso di un programma in circostanze particolari, e quindi solo in un <blocco> in cui le regolari strutture di controllo PASCAL sarebbero difficili da usare.

1a. Apprendimento dell'uso del GOTO, includendo dichiarazioni di etichette e ponendo etichette fra le istruzioni eseguibili.

1b. Apprendimento dell'uso della procedura interna EXIT che consente l'esecuzione di un GOTO fuori da un <blocco> con ritorno al <blocco> chiamante.

1c. Apprendimento dell'uso di construtti GOTO per simulare le principali istruzioni di controllo di PASCAL, come preparazione al lavoro con linguaggi di programmazione che mancano di quei construtti.

2. Premesse

Nella normale seguenza di elaborazione, un elaboratore digitale esegue un'istruzione e quindi prende in considerazione l'istruzione successiva memorizzata in memoria. In tutti gli elaboratori è inoltre possibile *saltare* da una locazione ad un'altra nella sequenza delle istruzioni in linguaggio macchina. Ciascuno dei principali linguaggi di programmazione ha un'istruzione *"GO TO"* (talvolta chiamata semplicemente "GO" o, come in PASCAL, "GOTO") per mezzo della quale si forza esplicitamente il compilatore a generare un'istruzione di salto.

Dopo che molte migliaia di persone ebbero scritto programmi nei primi linguaggi di programmazione (particolarmente FORTRAN e COBOL), gli studiosi di informatica i-

niziarono a capire che l'uso indisciplinato di istruzioni GOTO conduceva a errori di programma difficilmente scopribili. PASCAL è stato costruito con l'idea di evitare le fonti di errori di programma più comuni con l'uso dei primi linguaggi. L'uso dell'istruzione GOTO è meno enfatizzato in PASCAL, ma non del tutto eliminato. Si possono avere situazioni in cui un programma scritto senza l'uso di alcun GOTO si rivela difficile e inefficiente paragonato ad un programma simile contenente soltanto una o due istruzioni GOTO. Quindi PASCAL ha un'istruzione GOTO, sebbene in una forma che ne scoraggia un uso più che occasionale.

Mentre i principi generali della programmazione strutturata sono ormai largamente accettati nella tecnica di elaborazione dati, certi argomenti sono tuttora controversi. Il grado in cui sia possibile usare l'istruzione GOTO è uno degli argomenti più controversi in questo campo. La versione di PASCAL implementata nel nostro sistema software permette di saltare da una locazione ad un'altra all'interno di uno stesso <blocco> con l'uso dell'istruzione GOTO. In questa versione di PASCAL non è possibile usare il GOTO per saltare da un <blocco> ad un altro.

In grossi programmi, contenenti molte procedure, è particolarmente scomodo non avere la possibilità di uscire da una procedura di fronte a condizioni eccezionali. Questa necessità viene sentita frequentemente nella stesura di programmi esecutivi o grossi programmi interattivi. Per affrontare questi problemi abbiamo implementato la procedura interna EXIT che è una forma molto specifica e limitata dell'istruzione GOTO. Come avremo modo di vedere con maggiore dettaglio nelle sezioni successive, la procedura EXIT consente di interrompere l'esecuzione di qualsiasi procedura al momento attiva; ciò ha l'effetto di interrompere tutte le procedure *chiamate* da quella procedura, compresa quella contenente l'istruzione EXIT.

Le istruzioni di controllo PASCAL (REPEAT, WHILE, FOR, IF e CASE) possono essere viste come uso controllato dell'operazione di salto dell'hardware dell'elaboratore; in effetti sono usi "sicuri" del GOTO. Avendo capito l'uso delle istruzioni di controllo di PASCAL è possibile ottenere effetti simili in qualsiasi linguaggio di programmazione. Poiché questi linguaggi non hanno generalmente le stesse istruzioni di controllo di PASCAL, sarà necessario in tali linguaggi l'uso del GOTO per simulare PASCAL. Molte persone ritengono più efficiente scrivere i loro algoritmi usando istruzioni di programma in PASCAL o linguaggi similari; dopodiché traducono PASCAL nel linguaggio più conveniente sull'elaboratore usato per fare il lavoro. Sebbene questo schema possa sembrare difficile e contorto, si è dimostrato molto efficiente nell'aiutare queste persone ad evitare errori nei programmi e quindi a ridurre il tempo di messa a punto.

3. Tecnica del GOTO

Per usare un'istruzione GOTO, tre diverse voci devono essere incluse nel program-

ma; dovrà inoltre essere aggiunto uno pseudo-commento come messaggio di controllo al compilatore per permettere l'uso dell'istruzione GOTO. Le frasi interessate sono le seguenti:

a) Dichiarazione di un'etichetta

Un' *"etichetta"*, se presente, deve essere la prima cosa ad essere dichiarata in un <blocco>; si veda la sintassi del <blocco> nell'Appendice E. In PASCAL un'etichetta è un <intero non segnato> formato al massimo da 4 cifre decimali. Nella parte dichiarazione ETICHETTE di un <blocco> possono essere dichiarate più di un'etichetta, ma questo dovrebbe essere raramente necessario.

b) Istruzioni etichettate

Dopo aver definito come etichetta un <intero non segnato>, questo potrà essere usato per evidenziare una qualsiasi <istruzione> in un <blocco>. La stessa etichetta non potrà essere usata in questo modo per più di una volta nell'ambito dello stesso <blocco>; altrimenti il significato di un qualsiasi GOTO facente riferimento a quell'etichetta risulterebbe ambiguo. La sintassi per quest'uso di un'etichetta è data in testa alla sintassi dell'<istruzione> nell'Appendice E; l'etichetta è là menzionata come <intero non segnato>. Un'<istruzione> sarà etichettata in questo modo se si vorrà usare in qualche punto del <blocco> un'istruzione GOTO per saltare all'inizio dell'istruzione etichettata. Sebbene la sintassi non lo richieda, si consiglia di scrivere tutte le etichette al margine sinistro di un programma, in modo che siano visibili alla prima occhiata.

c) L'istruzione GOTO stessa

La sintassi dell'istruzione GOTO è data nel diagramma sintattico dell'<istruzione> nell'Appendice E. Per esempio:

```
GOTO 5
```

significa che il controllo di programma dovrebbe saltare all'inizio dell'<istruzione> con etichetta "5:". Si noti che nonostante la somiglianza superficiale fra un'etichetta usata in questo modo e i selettori usati in un'istruzione CASE, i due concetti sono completamente diversi.

d) Direttiva di compilazione G+

Per difetto, il nostro compilatore PASCAL non permette l'uso di istruzioni

GOTO. Questo controllo può essere eliminato dall'inserimento a inizio programma della riga di commento:

(*$G+*)

Il simbolo dollaro come primo carattere dopo "(*" avvisa il compilatore che il contenuto del commento non è parte del programma PASCAL, ma è una direttiva al compilatore stesso. Copiate tale "pseudo-commento" esattamente come mostrato qui per abilitare l'uso delle istruzioni GOTO. La maggior parte dei compilatori ha la possibilità di ricevere direttive su come devono lavorare. La convenzione di "commento dollaro" usata nel nostro compilatore PASCAL è simile alle convenzioni usate in altri compilatori, ma non esiste alcuna norma su tali convenzioni.

4. Programma tipo GOTODEMO

Le uscite di questo programma dovrebbero essere le stesse del programma BOOLDEMO del Capitolo 3 Sezione 9.

Questo programma è mostrato sotto forma di diagramma di struttura in Figura 11-1. Abbiamo aggiunto due nuovi blocchi al diagramma di struttura per mostrare cosa succede con l'uso delle istruzioni GOTO. La scatola ottagonale (come un segnale di STOP stradale), indica il GOTO (e l'EXIT). Il cerchio piccolo sarà usato per l'etichetta.

L'algoritmo di base seguito da questo programma è lo stesso usato per il programma BOOLDEMO del Capitolo 3. Abbiamo modificato quel programma, non in modo da renderlo migliore, ma soltanto per mostrare l'uso dell'istruzione GOTO. L'esempio potrà sembrare un po' forzato, questo forse vi farà capire la ragione per cui l'istruzione GOTO è poco usata in PASCAL.

Nell'uso dell'istruzione GOTO dovete far riferimento al diagramma di struttura del vostro algoritmo. Un uso "sicuro" del GOTO è quello che ha l'effetto di terminare l'elaborazione di qualche nodo del diagramma di struttura posto lungo il cammino diretto dal nodo origine al nodo in cui è posto il GOTO. Nel programma GOTODEMO, il GOTO del nodo 2b1a è al termine del nodo 2b e di tutti quelli che si dipartono da 2b (se ve ne fossero altri). Poiché l'elaborazione continua allo stesso livello di 2b (il nodo successivo sulla destra), l'etichetta (1:) è posta effettivamente fra i nodi 2b e 2c.

Il GOTO nel nodo 2a1 (linea 12) ha l'effetto di saltare fuori dal nodo 2, cioè dall'intera istruzione REPEAT iniziante alla linea 11. In questo caso non c'è alcuna <istru-

zione > successiva all'istruzione REPEAT; l'etichetta deve quindi essere mostrata nel diagramma di struttura come nodo "fantasma" nella struttura.

```
1: (*$G+*)
2: PROGRAMMA GOTODEMO;
3: LABEL 1,2;
4: VAR NOME,S:STRING;
5:    LN,LS,LK:INTEGER;
6: BEGIN
7:    S:='ALICE,BARBARA,CARLO,LUISA,FRANCO ';
8:    LS:=LENGTH(S);
9:    KS:=1;
```

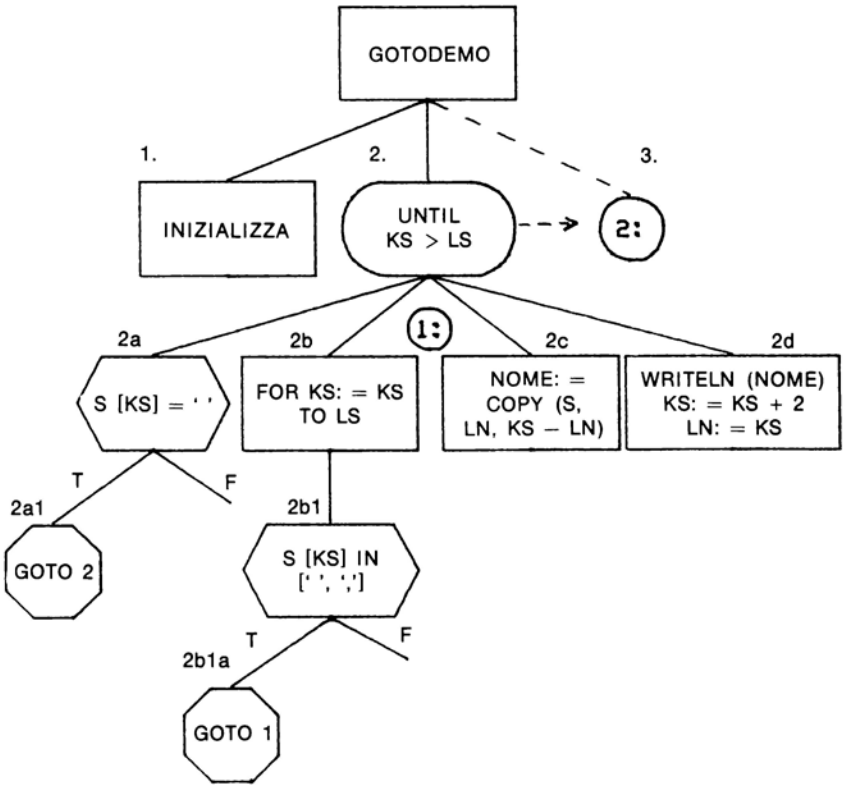


Figura 11-1

```
10:    LN:=1;
11:    REPEAT
12:       IF S[KS]=' ' THEN GOTO 2;
13:       FOR KS:=KS TO LS DO (*nodo 2b*)
14:          IF S[KS] IN [',',' '] THEN GOTO 1;
15: 1:    NOME:=COPY(S,LN,KS-LN); (*nodo 2c*)
16:       WRITELN(NOME);
17:       KS:=KS+1;
18:       LN:=KS;
19:    UNTIL KS >LS;
20: 2:
21: END.
```

5. EXIT

La nostra versione di procedura interna EXIT estende l'istruzione GOTO di PASCAL in modo molto limitato e controllato. Il solo parametro della procedura EXIT è l'<identificatore> della procedura che sarà attivata. Per spiegare cosa intendiamo con "*attivare*" una procedura dobbiamo nuovamente far riferimento alla rappresentazione di un algoritmo mediante un diagramma di struttura. Allo scopo di rendere utile un diagramma come mezzo di rappresentazione di un algoritmo abbiamo scelto di separare la parte di diagramma rappresentante una procedura o una funzione da quella rappresentante il programma principale. Comunque è facilmente immaginabile come sarebbe il diagramma di struttura se ogni chiamata di procedura (o funzione) fosse sostituita dall'intero diagramma di struttura di tale procedura; naturalmente questo renderebbe il diagramma ingombrante e non facile da usare. L'EXIT è equivalente ad un GOTO che interrompe l'elaborazione di qualsiasi nodo di chiamata di una procedura lungo il cammino dal nodo origine al punto in cui si trova il GOTO. Poiché la procedura é conosciuta tramite il suo <identificatore>, non c'è alcun bisogno di usare un'etichetta per informare il compilatore su dove il programma dovrebbe saltare.

Per spiegare meglio questo concetto, guardiamo il programma tipo EXITDEMO. Di seguito viene dato ciò che dovrebbe visualizzare:

```
ENTRO IN P: Prima Linea
LASCIO P
LASCIO Q
LASCIO R
```

```
ENTRO IN P:# Salto
LASCIO R

ENTRO IN P: Seconda Linea <ETX>
LASCIO P
LASCIO Q
LASCIO R
```

```
 1: PROGRAMMA EXITDEMO;
 2: VAR S:STRING;
 3:    CN: INTEGER;
 4:
 5: PROCEDURE Q; FORWARD;
 6:
 7: PROCEDURE P;
 8: BEGIN
 9:    WRITE('ENTRO IN P:');
10:    READ(S);
11:    IF S[1]='#' THEN EXIT(Q);
12:    WRITELN('LASCIO P');
13: END (*P*);
14:
15: PROCEDURE Q;
16: BEGIN
17:    P;
18:    WRITELN('LASCIO Q');
19: END (*Q*);
20:
21: PROCEDURE R;
22: BEGIN
23:    IF CN <=10 THEN Q;
24:    WRITELN('LASCIO R');
25: END (*R*);
26:
27: BEGIN (*PROGRAMMA PRINCIPALE*)
28:    CN:=0;
29:    WHILE NOT EOF DO
30:    BEGIN
31:       CN:=CN+1;
32:       R;
33:       WRITELN;
34:    END;
35: END.
```

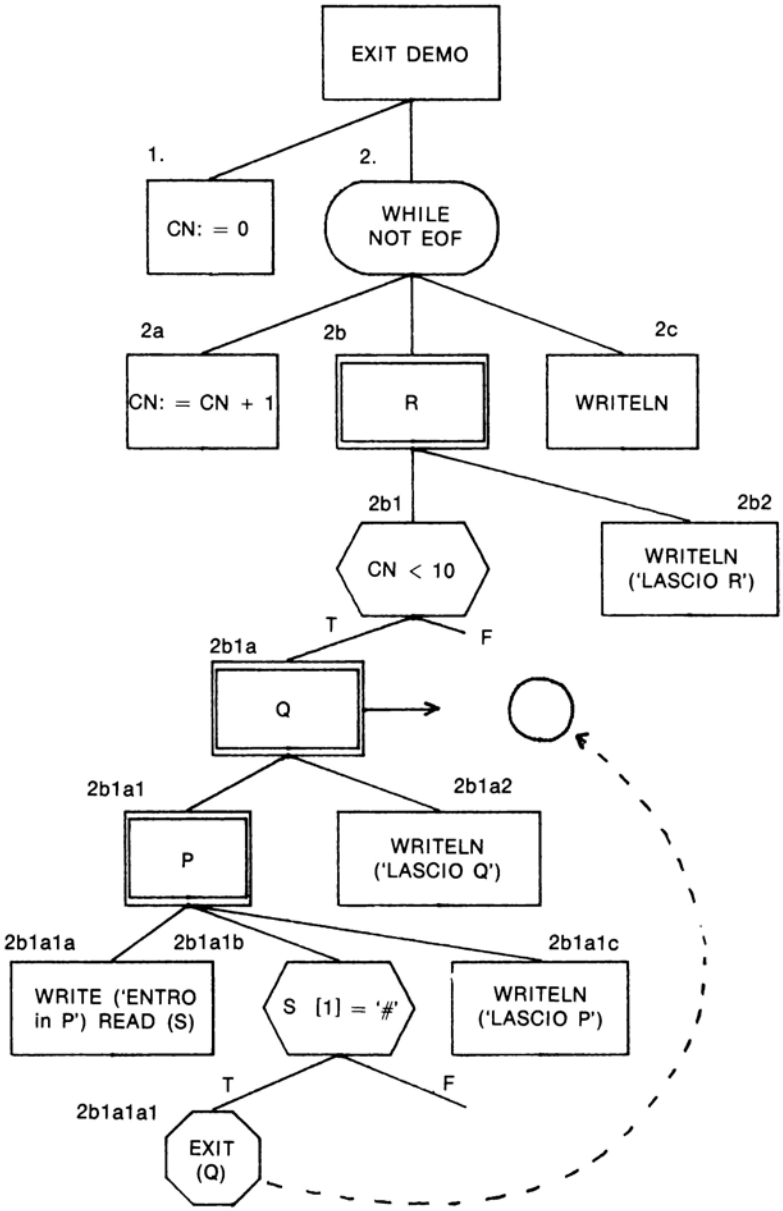

EXIT DEMO
1.
2.
CN: = 0
WHILE
NOT EOF
2a
2b
2c
CN: = CN + 1
R
WRITELN
2b1
2b2
CN < 10
WRITELN
('LASCIO R')
T
F
2b1a
Q
2b1a1
2b1a2
P
WRITELN
('LASCIO Q')
2b1a1a
2b1a1b
2b1a1c
WRITE ('ENTRO
in P') READ (S)
S [1] = '#'
WRITELN
('LASCIO P')
T
F
2b1a1a1
EXIT
(Q)

Figura 11-2

Questo programma vorrebbe suggerire il genere di azione intrapresa da un programma che elabora dati immessi, quando viene trovato un valore non valido. Il diagramma di struttura è dato in forma modificata in Figura 11-2. Questo diagramma mostra le procedure come se fossero state scritte nel programma principale nell'ordine in cui sono richiamate. La EXIT(Q) alla riga 11 provoca la interruzione della procedura P seguita dall'interruzione della procedura Q. L'elaborazione continua normalmente sulla linea seguente la chiamata di Q, cioè alla linea 24. Perciò la sola linea visualizzata successiva all'ingresso "# Salto" è "LASCIO R" alla fine della procedura R. Nei due casi in cui non viene richiamata EXIT, l'elaborazione prosegue normalmente fino al termine delle procedure P e Q.

Si noti che il risultato dell'istruzione EXIT(Q) è lo stesso che si sarebbe avuto con un'istruzione GOTO nel nodo 2b1a1b1 in grado di saltare ad un'etichetta posta immediatamente a destra del nodo 2b1a; ciò viene mostrato in figura 11-2 con la linea tratteggiata.

Si noti anche che è stato necessario dichiarare la procedura Q FORWARD perchè la EXIT potesse operare in modo corretto all'interno della procedura P. Se ciò non fosse stato fatto il riferimento a Q come parametro della EXIT avrebbe provocato una segnalazione di errore di sintassi per identificatore non definito. Q può essere un parametro della EXIT, nonostante la limitazione generale della nostra versione di PASCAL per cui non si possono trattare nomi di procedure come parametri, perchè la EXIT, essendo una procedura interna non è soggetta a tale limitazione.

6. L'uso di GOTO per simulare le istruzioni di controllo di PASCAL

Se, alla fine di questo libro, vi ritrovaste ad usare uno dei linguaggi di programmazione più vecchi, che non hanno le istruzioni di controllo di PASCAL, potreste ottenere qualcosa di simile con l'uso delle istruzioni GOTO. Di seguito vengono dati esempi per ognuna delle quattro principali istruzioni di controllo di PASCAL:

```
IF BOOL
  THEN S1
  ELSE S2;
```

diventa:

```
      IF NOT BOOL THEN GOTO 1;
        S1;
      GOTO 2;
1:    S2;
2:    ecc.
```

o:

```
        IF BOOL THEN GOTO 1;
          S2;
        GOTO 2;
1:      S1;
2:      ecc.
```

dove S1 e S2 rappresentano una qualsiasi istruzione o *gruppo* di istruzioni che si vuole usare. Si noti l'uso dell'indentazione per mantenere l'effetto visivo delle relazioni nella struttura del programma equivalente al nostro uso dei margini nei programmi PASCAL convenzionali. L'indentazione può essere usata per dare l'effetto visivo che abbiamo usato nelle istruzioni composte, poiché la maggior parte dei linguaggi che richiedono l'uso di GOTO per simulare le istruzioni di controllo, non hanno l'equivalente dell'istruzione composta di PASCAL.

Proseguiamo con gli esempi:

```
WHILE I <= 10 DO
BEGIN
  PROCA;
  I := I+1;
END;
```

diventa:

```
1:  IF I > 10 THEN GOTO 2;
      PROCA;
      I := I+1;
    GOTO 1;
```

e allo stesso modo:

```
REPEAT
  PROCB;
  I := I+1;
UNTIL I > 10;
```

diventa:

```
1:  (*un commento potrebbe mostrare la struttura*)
      PROCB;
      I := I+1;
      IF I <= 10 THEN GOTO 1;
      ecc.
```

Molte persone che usano questo metodo per scrivere programmi strutturati in FORTRAN trovano utile aggiungere commenti che mostrano esplicitamente la struttura di controllo di PASCAL che viene simulata. Infine:

```
FOR CV := SV TO LV DO
BEGIN
  PROCA;
  X := Y*Z;
END;
```

diventa:

```
        CV := SV
1:      IF CV > LV THEN GOTO 2;
          PROCA;
          X := Y*Z;
          CV := CV+1;
        GOTO 1;
2:        ecc.
```

Se una qualsiasi di queste conversioni non vi apparisse ovvia, vi consigliamo di rivedere i diagrammi di flusso dei quattro construtti IF, WHILE, REPEAT e FOR mostrati nel Capitolo 3. Negli ultimi tre esempi si presuppone che PROCA e PROCB siano procedure dichiarate da qualche altra parte del programma.

ESERCIZIO 11.1:
Per la necessità di usare molte istruzioni IF nidificate, il programma VERIFICADATA del Capitolo 7 Sezione 11 può apparire difficile da leggere e quindi soggetto ad errori di programma. Si riscriva VERIFICADATA, e si controlli il funzionamento del programma risultante, usando istruzioni GOTO per terminare l'elaborazione di una data errata, permettendo al programma di ritornare nuovamente nel ciclo principale. Il programma dovrebbe comunque, data la stessa data in ingresso, operare esattamente nello stesso modo di VERIFICADATA. Per questa conversione è necessario soltanto l'uso di GOTO in avanti, cioè di salti verso etichette poste più *avanti* nel programma. Quando è possibile, se proprio dovete usare dei GOTO, dovreste perlomeno sempre tentare di farli puntare verso etichette più avanti nel testo del progamma. Questo è equivalente alla regola del diagramma di struttura descritta precedentemente in questo capitolo.

ESERCIZIO 11.2:
Sebbene EXIT sia usato correttamente nel programma esemplificativo EXITDUE, esso non è in realtà necessario. Implementate il pro-

gramma sul vostro elaboratore e controllatelo con le seguenti linee di dati immesse:

```
PROGRAM TEST; VAR X,Y: INTEGER; BEGIN X:=1;
Y:=2; WRITELN(X+Y); END.
```

```
 1: PROGRAMMA EXITDUE;
 2: VAR CH: CHAR;
 3:   LN: INTEGER;
 4:   S: STRING;
 5:
 6: PROCEDURE SCAN;
 7: VAR C: CHAR;
 8:   I:INTEGER;
 9: BEGIN
10:   I:=1;
11:   WHILE I <=LENGTH(S) DO
12:     BEGIN
13:       C:=S[I];
14:       I:=I+1;
15:       IF C IN [';','.'] THEN
16:         BEGIN
17:           WRITELN(C);
18:           IF (C='.') OR (I >LENGTH(S)) THEN
19:             EXIT(SCAN);
20:           LN:=LN+1;
21:           WRITE(LN:3,': ');
22:         END ELSE
23:           WRITE(C);
24:     END (*WHILE*);
25: END (*SCAN*);
26:
27: BEGIN (*PROGRAMMA PRINCIPALE*)
28:   LN:=1;
29:   WHILE NOT EOF DO
30:     BEGIN
31:       WRITELN('LINEA IMMESSA:');
32:       READ(S);
33:       WRITE(LN:3,': ');
34:       SCAN;
35:       LN:=LN+1;
36:       WRITELN;
37:     END;
38: END.
```

facendo attenzione alle uscite generate dal programma. Dopo aver fatto questo si modifichi il programma eliminando EXIT e mantenendo inalterata la logica di programma. Si noti che questo si può ottenere cambiando due righe di programma e aggiungendone non più di quattro.

ESERCIZIO 11.3:
Alcuni linguaggi di programmazione non hanno l'equivalente dell'istruzione CASE, nè l'equivalente dell'istruzione PASCAL a due vie IF ... THEN ... ELSE. Riscrivete il programma CASEDEMO in modo da ottenere lo stesso risultato usando le istruzioni GOTO e l'IF a una via. Controllate il programma sia nella forma originale sia in quella modificata usando come dati di ingresso le seguenti linee:

+	100	200
*	32	64
–	10	20
/	50	150

e assicuratevi che diano gli stessi risultati.

ESERCIZIO 11.4:
Si riscriva il programma DISTURBI del Capitolo 7 Sezione 9 eliminando le istruzioni REPEAT, FOR e CASE. Il programma modificato non dovrà inoltre contenere l'istruzione WHILE. Si controlli il programma modificato assicurandosi che produca gli stessi risultati visti nel caso di DISTURBI.

ESERCIZIO 11.5:
Si riscriva il programma DEVOCALIZZA del Capitolo 7 Sezione 10 eliminando l'istruzione WHILE. Il programma modificato non dovrà inoltre contenere le istruzioni REPEAT e FOR. Si controlli il programma modificato assicurandosi che produca gli stessi risultati visti nel caso di DEVOCALIZZA.

```
1: PROGRAM CASEDEMO;
2: VAR CH:CHAR;
3:    X,Y:INTEGER;
4:    R:REAL;
5: BEGIN
6:    WRITELN('CASEDEMO');
7: WHILE NOT GOF DO
8:    BEGIN
```

```
 9:      WRITE('+ -* or / :');
10:      READ(CH);
11:      WRITE(' X:);
12:      READ(X);
13:      WRITE(' Y:');
14:      READ(X);
15:      CASE CH OF
16:        '+': R:=X+Y
17:        '-': R:=X-Y
18:        '*': R:=X*Y
19:        '/': R:=X/Y
20:      END (*CASE*);
21:      READ(CH); (*elimina lo spazio dopo READ(Y)*)
22:      WRITELN;
23:      WRITELN(X, ' ',CH,' ',Y,' ='R);
24:      WRITELN;
25:    END (*WHILE*);
26: END.
```

CAPITOLO 12

EMISSIONI COMPOSTE (FORMATTED)

1. Obiettivi

Acquisire un'esperienza nella soluzione di problemi mediante lo studio e la comprensione dei programmi esemplificativi dati in questo capitolo, e mediante il lavoro sugli esercizi.

1a. Si impari ad usare la logica di programma per porre caratteri dove li si vuole sullo schermo di un video o su una pagina di stampa.

1b. Se lavorate usando uno schermo con mappatura a bit, rivedete gli esempi dati in questo capitolo per tracciarli con una più alta risoluzione

2. Premesse

Fino ad ora in ogni capitolo abbiamo presentato strumenti di base della programmazione o concetti sull'applicazione della struttura di programma alla soluzione di problemi. In questo capitolo e in tutti i successivi presenteremo esempi di programmi costruiti per risolvere problemi semplici, ma tipici. Se avete dovuto faticare per giungere fino a questo punto, avrete ora una visione ragionevolmente completa su come si possano risolvere i problemi usando gli elaboratori, ma vi potrebbe risultare difficile usare gli elaboratori per risolvere problemi aventi complessità più che elementare. Se il raggiungere questo punto ha presentato una difficoltà relativamente piccola dovreste apprendere velocemente i prossimi capitoli. Se anche questi fossero facili, vi consiglio di approfondire qualche aspetto degli elaboratori e/o dei linguaggi di programmazione.

In questo capitolo daremo diversi esempi della composizione delle emissioni di un programma. Per la maggior parte dei programmatori la composizione delle emissioni è un'attività appariscente ma necessaria. I metodi usati per presentare immagini su un dispositivo grafico di visualizzazione, basati sul principio "Mappatura a bit" (V. Fi-

gura 0-1), sono molto simili a quelli usati per visualizzare caratteri su un video o su una pagina di stampa per molti altri scopi. Ci concentreremo maggiormente in questo capitolo sui caratteri di composizione perché le vostre attività future di programmazione implicheranno più facilmente caratteri di stampa e di visualizzazione piuttosto che grafici.

Parecchi dei più noti linguaggi di programmazione usano una tecnica di compilazione delle emissioni nota come *"formato"*. I formati sono tradizionalmente una delle maggiori fonti di confusione per gli studenti che iniziano corsi di programmazione. Un formato è fondamentalmente un programma per convertire dati memorizzati all'interno dell'elaboratore in una stringa di caratteri in uscita verso un dispositivo tipo una stampante o CRT (o per trattare l'operazione inversa in lettura). Il linguaggio usato per scrivere formati equivale ad un mezzo per chiamare speciali procedure interne che operano la conversione di dati. Sfortunatamente la sintassi usata nei maggiori linguaggi di programmazione per i formati è completamente diversa da quella usata per le altre parti dei programmi. In effetti lo studente è costretto ad imparare due linguaggi insieme per avere un controllo ragionevole di come l'informazione emessa appaia su un video o una pagina di stampa.

Le tecniche di composizione delle emissioni di PASCAL sono molto semplici e la maggior parte di esse vi sono già familiari. Uno svantaggio è che PASCAL non possiede l'ampia varietà di formati di uscita disponibili in altri linguaggi. Per esempio PASCAL non ha tecniche costruite per la stampa di numeri in sistemi di numerazione Ottale (base 8) o Esadecimale (base 16) sebbene siano frequentemente necessari nel lavoro di elaborazione; nè la nostra versione di PASCAL vi dà un controllo completo sul formato di stampa di un numero REAL. Nonostante la semplicità, le idee espresse nei passi di programma per WRITE su un dispositivo di stampa o di visualizzazione possono essere facilmente tradotte nei formati speciali richiesti da altri linguaggi.

Si è già visto come sia possibile visualizzare parecchi dati indipendenti su una stessa linea in uscita semplicemente chiamando la WRITE con parametri adeguati un certo numero di volte. La principale nuova idea presentata in questo capitolo riguarda l'uso di un vettore contenente una cella per ogni carattere di uno schermo di visualizzazione o di una pagina di stampa. Questo permette il posizionamento dei dati per l'uscita ogniqualvolta il flusso del programma lo renda conveniente; esso elimina il problema creato dal fatto che tipicamente le stampanti accettano dati soltanto su basi strettamente ordinate linea per linea da sinistra a destra, dall'alto verso il basso.

3. Programma tipo FORMATDEMO

Questo programma estende la dimostrazione data nel Capitolo 8 sull'uso di specifi-

care le dimensioni di campo nelle istruzion WRITE come mezzo di controllo del numero di colonne in uscita. In generale si dovrà tener conto del numero di colonne di informazione visualizzate su ogni riga se linee visualizzate da istruzioni diverse devono essere mischiate in modo non confuso. Per ottenere questo si può controllare il numero di spazi bianchi a sinistra di un operando, e si può controllare il numero di colonne occupate dall'informazione stessa se ha senso.

Le linee dalla 11 alla 14 di questo programma mostrano questo modo di visualizzare numeri reali. Se non è specificata alcuna lunghezza di campo, il formato per difetto usa tante colonne quante sono necessarie per visualizzare il numero con la massima precisione. Nella riga 3 della lista visualizzata, una colonna a sinistra del '14', è stata lasciata vuota, questo perché quella colonna dovrebbe essere occupata dal segno meno ('—') in caso di numero negativo. Se viene specificata la lunghezza di un solo campo, come nelle linee 12 e 13 del programma, si possono verificare due casi. Se l'ampiezza stabilita per il campo è minore di quella necessaria a rappresentare il numero con la massima precisione, verrà usato il numero più grande di colonne necessario allo scopo, verrà cioè prevaricata l'ampiezza specificata. Se invece vengono specificate più colonne di quelle necessarie per la massima precisione, le rimanenti colonne verranno lasciate vuote alla sinistra del numero visualizzato (linea 5 della lista visualizzata).

Si può specificare di visualizzare un numero minore di cifre a destra del punto decimale: questo è mostrato alla linea 14 del programma. La linea 6 della lista visualizzata mostra come l'aver specificato 3 quale lunghezza del secondo campo abbia limitato il numero di cifre decimali visualizzato.

```
 1: PROGRAMMA FORMATDEMO;
 2: VAR R:REAL;
 3:       I:INTEGER;
 4:       CH:CHAR;
 5:       A:PACKED ARRAY[1..10] OF CHAR;
 6: BEGIN
 7:   R:=100/7;
 8:   I:=16384;
 9:   CH:='*';
10:   A:='FORMATDEMO';
11:   WRITELN('(REALE PER DIFETTO)',R);
12:   WRITELN('(R:6)',R:6);
13:   WRITELN('(R:16)',R:16);
14:   WRITELN('(R:10:3)',R:10:3);
15:   WRITELN;
16:   (*ora gli interi*)
17:   WRITELN('(INTERO PER DIFETTO)',I);
```

```
18:    WRITELN('(I:3)',I:3);
19:    WRITELN('(I:5)',1:5);
20:    WRITELN('(I:7)',I:7);
21:    WRITELN;
22:    (*carattere singolo*)
23:    WRITELN('(CAR PER DIFETTO)',CH);
24:    WRITELN('(CH:5)',CH:5);
25:    WRITELN;
26:    (*vettore impaccato di caratteri*)
27:    WRITELN('(VETTORE CHAR PER DIFETTO)',A);
28:    WRITELN('(A:5)',A:5);
29:    WRITELN('(A:15)',A:15);
30: END.
```

```
 1: Visualizzazioni associate al programma FORMATDEMO
 2:
 3: (REALE PER DIFETTO) 14.28571
 4: (R:6)   14.28571
 5: (R:16)              14.28571
 6: (R:10:3)        14.286
 7:
 8: (INTERO PER DIFETTO)16384
 9: (I:3)16384
10: (I:5)16384
11: (I:7)   16384
12:
13: (CAR PER DIFETTO)*
14: (CH:5)         *
15:
16: (VETTORE CHAR PER DIFETTO)FORMATDEMO
17: (A:5)FORMA
18: (A:15)         FORMATDEMO
```

Per gli operandi interi, le regole sono simili a quelle per i reali. Nessuna colonna è riservata per il segno meno, a meno che sia necessaria. Il numero di colonne usate sarà o il minimo sufficiente alla visualizzazione del valore intero o un numero maggiore di colonne. Una dichiarazione di lunghezza campo troppo piccola per visualizzare tutte le cifre dell'intero verrà ignorata; una dichiarazione di lunghezza campo più grande del necessario porterà alla visualizzazione di spazi vuoti alla sinistra del numero stesso.

Nel caso di operandi carattere questi verranno visualizzati per difetto su una sola

colonna. Se il campo specificato è maggiore di 1, gli spazi vuoti saranno inseriti a sinistra del carattere stesso.

Nel caso di stringhe, cioè di vettori impaccati di operandi CHAR, verranno visualizzati per difetto il numero di caratteri dichiarati nel vettore. Questo va distinto dal valore preso per difetto nella visualizzazione del valore contenuto in una variabile STRING. Se una variabile STRING S è visualizzata senza alcuna dichiarazione di ampiezza di campo, l'assunzione per difetto sarà di visualizzare un numero di caratteri pari a LENGTH(S). Come mostrato per la lista visualizzata da FORMATDEMO, una dichiarazione di ampiezza di campo con un numero di colonne minore del numero di caratteri contenuti nel vettore provocherà la visualizzazione del numero minore di colonne. Un'ampiezza maggiore del contenuto del vettore porterà invece all'inserimento di spazi bianchi a sinistra come in ogni altro <tipo> di operando.

4. Programma tipo CHARPLOT

La visualizzazione in uscita di CHARPLOT è mostrata in Figura 12-1. Questo programma viene presentato principalmente per mostrare l'uso di un vettore PAGINA a due dimensioni come deposito temporaneo di caratteri che saranno eventualmente visualizzati riga per riga. Questo permette al programma di "scrivere" caratteri in qualsiasi posto della pagina e in qualsiasi ordine renda più facile lo scorrere del programma. Senza questa memoria temporanea anche qualcosa di così semplice come quello mostrato sarebbe difficile da visualizzare riga dopo riga.

```
 1: PROGRAMMA CHARPLOT;
 2: CONST
 3:    LARGHEZZA=25;
 4:    ALTEZZA=11;
 5: TYPE
 6:    DX=-LARGHEZZA .. +LARGHEZZA;
 7:    DY=-ALTEZZA .. +ALTEZZA;VAR
 8:    PAGINA:ARRAY[DY] OF
 9:              PACKED ARRAY[DX] OF CHAR;
10:    X:DX;
11:    Y:DY;
12:
13: PROCEDURE RECT(A,L:INTEGER; CH:CHAR);
14: VAR I,XMIN,XMAX:DX;
15:      J,YMIN,YMAX:DY;
16: BEGIN
```

```
17:    XMAX:=L; YMAX:=A;
18:    XMIN:=-L; YMIN:=-A;
19:    (*marca cima e fondo del rettangolo*)
20:    FOR I:=XMIN TO XMAX DO
21:    BEGIN
22:       PAGINA[YMAX,I]:=CH;
23:       PAGINA[YMIN,I]:=CH;
24:    END;
25:    (*marca i lati*)
26:    FOR J:=YMIN TO YMAX DO
27:    BEGIN
28:       PAGINA[J,XMAX]:=CH;
29:       PAGINA[J,XMIN]:=CH;
30:    END;
31: END (*RECT*)
32:
33: PROCEDURE AZZERAPAGINA;
34: VAR I:DX;
35:         J:DY;
36: BEGIN
37:    FOR J:=-ALTEZZA TO ALTEZZA DO
38:       FOR I:=-LARGHEZZA TO LARGHEZZA DO PAGINA[J,I]:=' ';
39: END(*AZZERAPAGINA*);
40:
41: BEGIN (*PROGRAMMA PRINCIPALE*)
42:    AZZERAPAGINA;
43:    RECT(11,10,'*');
44:    RECT(5,25,'#');
45:    RECT(8,15,'&');
46:    FOR Y:=ALTEZZA DOWNTO -ALTEZZA DO
47:    BEGIN
48:       FOR X:=-LARGHEZZA TO LARGHEZZA DO WRITE(PAGINA[Y,X]);
49:       IF Y > -ALTEZZA THEN WRITELN;
50:    END;
51: END.
```

Il programma deposita dapprima spazi vuoti in ogni locazione del vettore PAGINA; a questo punto i caratteri che formano le figure da visualizzare possono essere posti in qualsiasi locazione del vettore. Si noti che il carattere '*' è sostituito da '#' e '&' che sono scritti dopo, mentre '#' è sostituito da '&' dove coincidono. Come ultimo passo il contenuto del vettore PAGINA è poi visualizzato riga dopo riga.

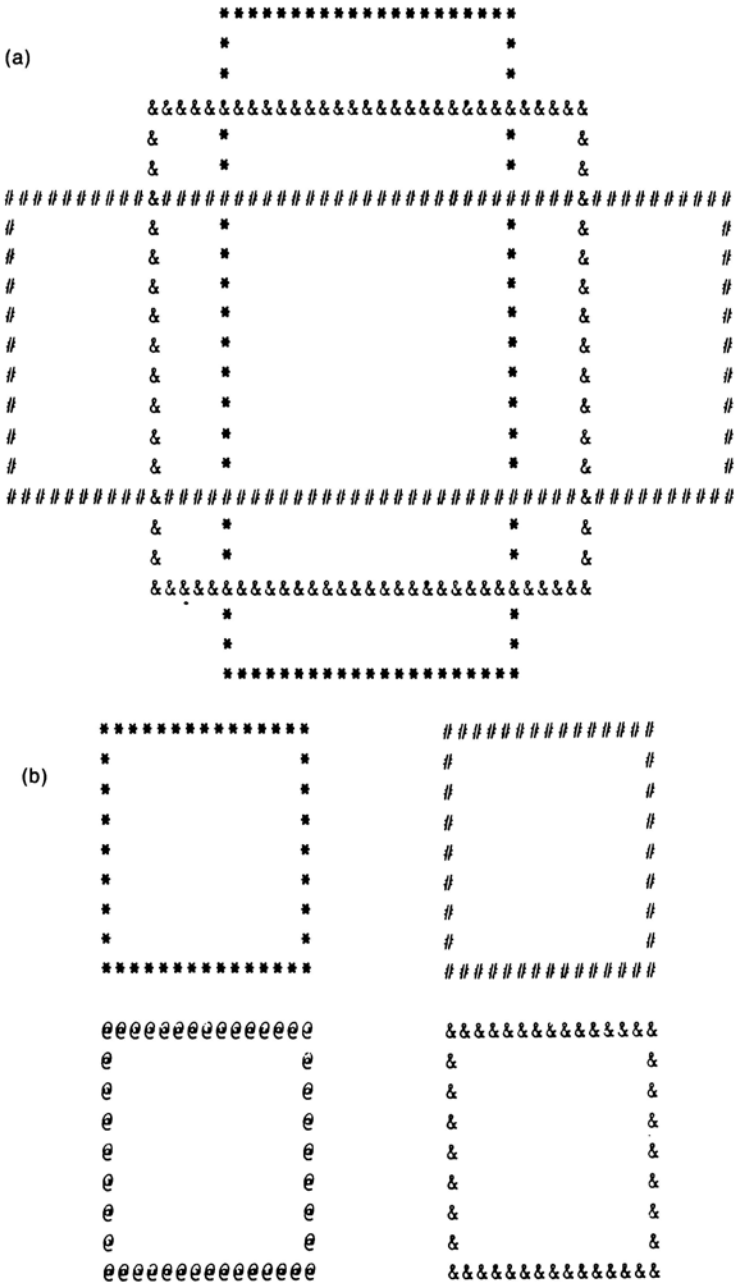


Figura 12-1: Visualizzazione associata con il programma CHARPLOT

Si noti nella linea 46 che la prima riga visualizzata è in cima alla f gura, e che la visualizzazione procede poi verso il basso, cioè lungo PAGINA in ordine inverso; la figura visualizzata sarebbe altrimenti invertita. Ciò non avrebbe alcuna conseguenza per la figura simmetrica visualizzata da questo programma, ma potrebbe essere importante per la visualizzazione di figure che non siano a simmetria verticale.

Si noti che le spaziature fra i caratteri visualizzati sono maggiori verticalmente che orizzontalmente. Se le spaziature fossero le stesse in entrambe le direzioni, la figura disegnata con gli asterischi ('*') sarebbe praticamente quadrata. Se la figura che volete disegnare con uno schermo convenzionale o una stampante avesse scale orizzontali e verticali uguali, dovreste moltiplicare gli spostamenti orizzontali, o dividere quelli verticali, per un fattore equalizzante.

5. Programma TIPO CURVEPLOT

Questo programma illustra l'uso del video alfanumerico dell'elaboratore o di una stampante per tracciare grafi primitivi di funzioni elaborate. I lettori di cultura matematica riconosceranno nella funzione disegnata da questo programma, e mostrata in Figura 12-2, la funzione Sin(x)/x. La conoscenza della funzione interna SIN non è comunque necessaria per capire questo programma.

```
 1: PROGRAMMA CURVEPLOT;
 2: CONST
 3:    LARGHEZZA=25; ALTEZZA=22;
 4: TYPE
 5:    DX= -LARGHEZZA..+LARGHEZZA;
 6:    DY= 0.. ALTEZZA
 7:    VETTOREDATI= ARRAY[DX] OF REAL;
 8: VAR
 9:    PAGINA: ARRAY([DY] OF PACKED ARRAY[DX] OF CHAR;
10:    X:DX;
11:    Y:DY;
12:    F: VETTOREDATI;
13:    RX: REAL;
14:    YZERO: INTEGER;
15:
16: PROCEDURE SCALA (VAR A:VETTOREDATI);
17: (*adatta i dati alla scala verticale del disegno*)
18: VAR FATTORES,DMIN,DMAX: REAL;
19:    I:DX;
```

```
20: BEGIN
21:    DMIN:=1.0E+30;
22:    DMAX:=-1.0E+30;
23:    FOR I:=-LARGHEZZA TO LARGHEZZA DO
24:    BEGIN
25:       IF A[I] < DMIN THEN DMIN:=A[I];
26:       IF A[I] > DMAX THEN DMAX:=A[I];
27:    END;
28:    IF DMAX< >DMIN THEN
29:       FATTORES:=ALTEZZA/(DMAX-DMIN);
30:    FOR I:= -LARGHEZZA TO LARGHEZZA DO
31:       A[I]:=(A[I] -DMIN)*FATTORES;
32:    YZERO:=ROUND(-DMIN*FATTORES);
33: END (*SCALA*);
34:
35: PROCEDURE INIZIA;
36: BEGIN
37:    FOR Y:=0 TO ALTEZZA DO
38:       FOR X:= -LARGHEZZA TO LARGHEZZA DO PAGINA[Y,X]:=' ';
39:    (*asse verticale*) FOR Y:=0 TO ALTEZZA DO PAGINA[Y,0]:='.';
40: END (*INIZIA*);
41:
42: PROCEDURE ASSEX;
43: BEGIN
44:    IF YZERO IN [0..ALTEZZA] THEN
45:       FOR X:= -LARGHEZZA TO LARGHEZZA DO;
46:          PAGINA[YZERO,X]:='.';
47: END ASSEX;
48:
49: PROCEDURE DISEGNA(VAR A:VETTOREDATI; CH:CHAR);
50: VAR J:DX;
51: BEGIN
52:    SCALA(A);
53:    FOR J:= -LARGHEZZA TO LARGHEZZA DO
54:    PAGINA[ROUND(A[J]),J]:=CH;
55: END (*DISEGNA*);
56:
57: BEGIN (*PROGRAMMA PRINCIPALE*)
58:    INIZIA;
59:    FOR X:= -LARGHEZZA TO LARGHEZZA DO
60:    BEGIN
61:       (*RX è X moltiplicato per un'opportuna scala orizzontale*)
62:       RX:=X * 0.25;
```

```
63:       IF RX=0.0 THEN F[X]:=1.0 ELSE
64:          F[X]:=SIN(RX)/RX; (*funzione per disegnare F*)
65:    END;
66:    DISEGNA(F,'*');
67:    ASSEX;
68:    (*stampa i risultati*)
69:    FOR Y:=ALTEZZA DOWNTO 0 DO
70:    BEGIN
71:       FOR X:=-LARGHEZZA TO LARGHEZZA DO WRITE(PAGINA[Y,X]);
72:       IF Y>0 THEN WRITELN;
73:    END;
74: END.
```

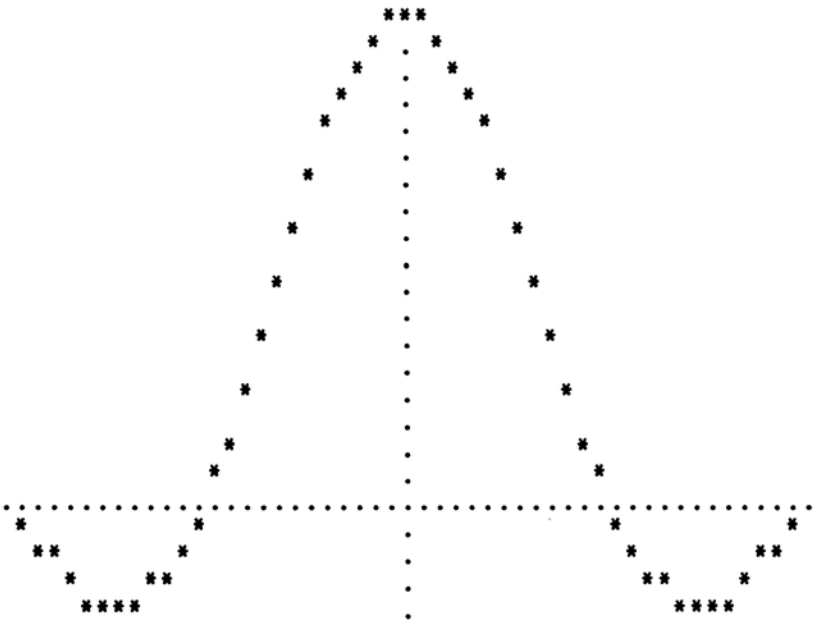


Figura 12-2: Visualizzazione associata con il programma CURVEPLOT

L'illustrazione è stata ottenuta usando la funzione SIN, ma potrebbe sostituire allo stesso modo una qualsiasi funzione vostra nella linea 64. È infatti possibile sovrapporre sulla stessa pagina o sullo stesso video tanti grafici quanti si vuole, però, in modo che non sia troppo difficile leggere il risultato dal grafico visualizzato.

La strategia seguita in questo caso è analoga a quella usata nella sezione precedente per disegnare figure geometriche. Sarà necessario arrotondare il valore reale della funzione, rappresentante l'ordinata, per scegliere una specifica posizione. Per quanto riguarda l'ascissa, si è generalmente liberi di scegliere il valore della variabile indipendente X prima di richiamare la funzione che calcola il corrispondente valore di Y. Come si può vedere nel programma principale, sarà a volte necessario moltiplicare il valore di X per qualche fattore di "scala" (0.25 in questo caso) per coprire tutto il segmento orizzontale desiderato.

Poiché vi sono relativamente poche righe di caratteri su cui lavorare, è consigliabile fare in modo che il grafico riempia verticalmente tutto lo schermo per poter ottenere una curva più smussata possibile. Dal momento che generalmente non si avrà una funzione i cui valori coprono esattamente il numero di righe dello schermo, sarà necessario moltiplicare tutti i valori di Y per un "*fattore di scala*".

Questo programma, nelle linee 63 e 64, calcola dapprima un valore della funzione da disegnare per ogni valore di X scelto, ponendo i valori nel vettore F. L'istruzione IF tiene conto di una caratteristica anomala della funzione Sin(x)/x quando x=0, per cui la funzione assume il valore 1.0. Senza questo controllo IF il programma terminerebbe in modo anomalo al tentativo di dividere Sin(0) per 0. Dopo che tutti i valori della funzione sono stati calcolati, si richiama DISEGNA passando F come parametro. DISEGNA richiama SCALA per sistemare i valori in F (cui si fa riferimento in SCALA e DISEGNA come A) in modo che essi coprano esattamente l'intervallo di righe 0..ALTEZZA. Alla fine di SCALA viene assegnato alla variabile globale YZERO il valore del numero di riga in cui Y=0. Tale numero di riga può appartenere allo schermo o non appartenervi a seconda dell'intervallo di valori della funzione, tale fatto è controllato quando viene chiamata ASSEX per disegnare la riga orizzontale di puntini che rappresenta l'asse X.

Si noti che il "grafo" disegnato da questo programma ha una parte appiattita e dei punti a scala dove un disegno accurato avrebbe mostrato una curva più smussata. La parte appiattita dipende dal fatto che non possiamo disegnare punti fra le locazioni fisse in cui possono apparire i caratteri sullo schermo o sulla pagina di stampa. I dispositivi per disegnare grafi sarebbero un lavoro molto migliore di questo, probabilmente con la spesa di un algoritmo di disegno molto più complesso.

6. Nota sui grafici mappati a bit

Se state usando un microelaboratore fornito di dispositivo video in grado di pilotare linee col metodo illustrato in Figura 0-1, i metodi descritti in questo capitolo possono essere usati per disegnare figure di qualità migliore di quelle qui mostrate. Il sistema

di programmazione UCSD PASCAL prevede un metodo per cui si può avere controllo diretto di ogni posizione puntuale dello schermo: si dà in questa sezione una breve descrizione di tale metodo. Vi sproniamo a cimentarvi in disegni più complessi di quelli che abbiamo potuto mostrare in questo libro.

Descriviamo come *"mappa a bit"* la figura disegnata col metodo illustrato in Figura 0-1 poichè ogni punto dello schermo corrisponde ad un bit della memoria dell'elaboratore. Dovendo usare al posto del vettore PAGINA utilizzato negli esempi di programma di questo capitolo, il vettore SCHERMO per lo stesso scopo, la dichiarazione sarebbe del tipo:

```
VAR SCHERMO: ARRAY[0..ALTEZZA] OF
                    PACKED ARRAY[0..DIMENSIONERIGA] OF BOOLEAN;
```

dove DIMENSIONERIGA è uguale al numero di bits (meno 1) dalla sinistra alla destra dello schermo, e ALTEZZA è inferiore di 1 rispetto al numero di bits necessari a coprire una linea verticale. Ad esempio sul microelaboratore Terak 8510A, il valore di ALTEZZA sarebbe 239 e il valore di DIMENSIONERIGA 319, corrispondente a uno schermo di 240 per 320; si paragoni questo alle 24 righe per 80 caratteri disponibili sulla maggior parte degli schermi alfanumerici. Nei programmi di prova di questo capitolo, scelti specificatamente per un video alfanumerico, useremo una larghezza di riga di soli 51 caratteri per stare nei limiti di pubblicazione di questo libro. Se possedete uno schermo mappato a bit, questo potrà avere ALTEZZA e DIMENSIONERIGA diverse da quelle del Terak, ma tali valori saranno generalmente sostanzialmente maggiori di quelli del vettore PAGINA visto in questo capitolo.

Oltre alle dimensioni il vettore SCHERMO differisce sotto un altro aspetto dai vettori PAGINA utilizzati in questo capitolo: mentre PAGINA memorizzava i caratteri per una successiva visualizzazione sotto forma di caratteri, SCHERMO memorizza semplicemente un valore Booleano indicante se la posizione dello schermo sarà visualizzata come chiara o scura, sarà chiara se TRUE, scura se FALSE.

Gli espedienti esatti necessari a trasmettere i valori dei bits del vettore SCHERMO al video varieranno probabilmente da macchina a macchina a seconda delle caratteristiche tecniche: sull'unità Terak 8510A i meccanismi sono molto semplici. Durante l'inizializzazione del programma dovrete includere la seguente istruzione:

```
UNITWRITE(3, SCHERMO, 63)
```

dove UNITWRITE è una procedura interna utilizzata normalmente soltanto dai "programmatori di base", coloro che lavorano su compilatori, programmi redattori e altri programmi di grossi sistemi. Il primo parametro "3" si riferisce al video; il secondo, SCHERMO, è il nome del vettore a due dimensioni di valori Booleani di cui abbiamo

parlato. Quando viene eseguita questa istruzione UNITWRITE, viene avvisata l'unità grafica di visualizzare i bits memorizzati in memoria nel vettore SCHERMO. Il terzo parametro, qui rappresentato dalla costante intera 63, è un'espressione usata per controllare l'unità video. Il valore 63 si applica specificamente all'unità Terak 8510A, e sarà probabilmente affatto diverso per altre marche e modelli. Dopo l'esecuzione della UNITWRITE, tutti i cambiamenti nei bits del vettore SCHERMO saranno visibili sull'unità video fino al termine del programma.

Per ulteriori informazioni sull'uso dei videi mappati a bit, si faccia riferimento al manuale di utilizzo del dispositivo che state usando, e a un supplemento che descriva come il sistema di programmazione PASCAL comunichi con tale dispositivo.

ESERCIZIO 12.1:
Si modifichi il programma CHARPLOT in modo che visualizzi i quattro piccoli rettangoli mostrati nella parte b della Figura 12-1. Nota: se capite come opera tale programma, questo dovrebbe essere un esercizio molto semplice. Noi abbiamo operato il cambiamento dichiarando due nuovi identificatori (non vi diremo dove, perchè ciò rivelerebbe la soluzione), aggiungendo tali identificatori a quattro istruzioni, e sostituendo tre altre istruzioni.

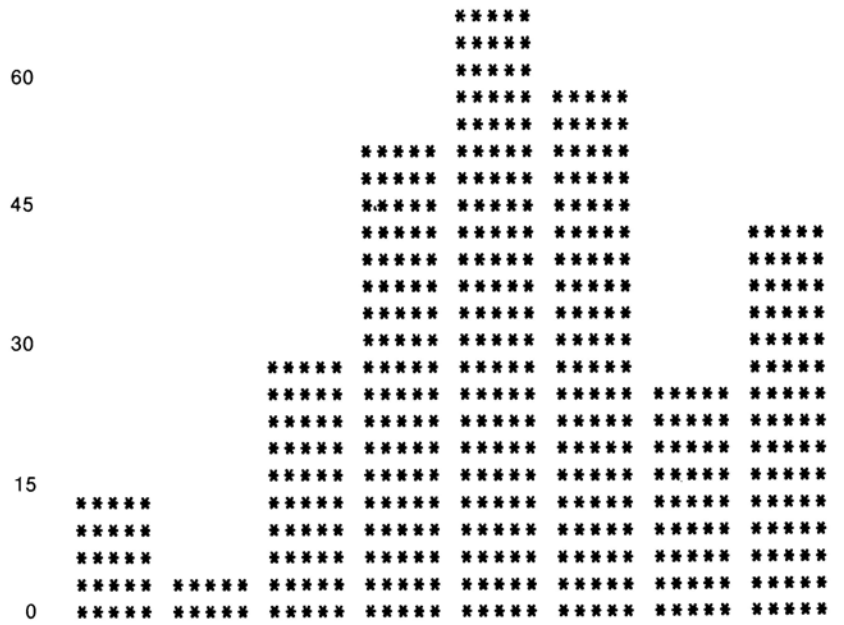


Figura 12-3: Istogramma associato con Esercizio 12-3

Molti studenti affrontano questo problema con un approccio del tipo "vediamo cosa succede a fare così", senza capire il programma: questo è un buon metodo per perdere tempo. Se doveste avere problemi nel capire CHARPLOT, risparmierete tempo eliminando i disegni di due o tre rettangoli, e considerando poi ciò che rimane. In effetti la visualizzazione finale dell'intero contenuto del vettore PAGINA si riduce ad una grande istruzione di messa a punto. Potreste spostare il ciclo di uscita (linee dalla 46 alla 50) in una nuova procedura, e chiamare poi tale procedura in punti strategici di RECT per vedere come vengono disegnati i lati e la cima del rettangolo.

Dopo aver capito come lavora il programma dato, vi potrà essere d'aiuto pensare alle modifiche richieste nel seguente modo: ogni rettangolo è disegnato simmetricamente attorno al suo punto centrale. Tutti e tre i rettangoli disegnati dal programma originario sono centrati in X=0, Y=0; per disegnare i nuovi rettangoli saranno necessari nuovi valori di X e Y.

ESERCIZIO 12.2:
Si riveda il programma CURVEPLOT in modo che disegni *entrambe* le funzioni interne SIN *e* COS, usando caratteri di disegno diversi per le due curve per identificarle visivamente sul video. Si cerchi di cambiare la costante 0.25, come mostrato sulla linea 62 del programma di prova, nei valori 0.20 e 0.33, e si spieghino i cambiamenti risultanti nella figura.

ESERCIZIO 12.3:
Per studi statistici, è spesso utile avere un "*istogramma*" che mostri le dimensioni relative di diversi elementi simili, per esempio il consumo di energia in ognuno dei prossimi anni. Scrivete e mettete a punto un programma PASCAL che disegni l'istogramma mostrato in Figura 12-3 utilizzando come dati la seguente lista di percentuali:

12 5 29 51 66 59 25 43

Prima di pensare a risolvere questo problema, si pensi a come dovrebbe essere costruito l'algoritmo. Pur essendo possibile ottenere la visualizzazione mostrata in Figura 12-3 includendo i numeri sul margine sinistro per indicare la scala e usando un vettore PAGINA simile ai vettori utilizzati nei programmi di prova di questo capitolo, questa non è la via più semplice di risolvere il problema. Pensate a come avreste risolto il problema quando non eravate ancora giunti oltre il Capitolo 3 di questo libro; quindi procedete con la vostra soluzione.

(Morale: la vita reale è raramente simile ad un libro di testo in cui siete condotti per mano al metodo migliore per risolvere un problema. Prima di spendere molto tempo nelle specifiche di un algoritmo, considerate il repertorio di metodi disponibili che potrebbero riguardare il problema).

ESERCIZIO 12.4:
Se avete un'unità video mappata a bit rivedete i programmi CHARPLOT e CURVEPLOT per farli lavorare su mappe a bit. Rivedeteli in modo da poter ripetere i cambiamenti richiesti negli Esercizi 12-1 e 12-2. Cercate di disegnare l'istogramma dell'Esercizio 12-3 con barre verticali "ombreggiate" usando linee oblique uniformemente distanti. Se la vostra unità video è in grado di visualizzare mappe di bit a colori cercate di ombreggiare ogni barra in un colore diverso.

ESERCIZIO 12.5:
Gli elaboratori sono molto usati per fare molti compiti, nel lavoro con mappe, che sono trovati tediosi dall'uomo. Uno di tali compiti è la "ombreggiatura" di aree irregolari in una mappa per evidenziarle come aventi qualche caratteristica comune. Ad esempio l'intera area di uno stato o contea potrebbe essere colorata in una stessa tinta, mentre suddivisioni politiche adiacenti potrebbero essere colorate usando altre tinte. Sul video di un elaboratore o su una stampante che non possa usare i colori, è possibile ottenere un'ombreggiatura equivalente controllando la densità o l'angolo di incontro delle linee tracciate sull'area da ombreggiare. Se la mappa è fatta su una traccia rozza usando un'unità video a carattere o una stampante, l'ombreggiatura può essere ottenuta utilizzando caratteri diversi.

Come primo passo di questo esercizio si usino grafici tartaruga per visualizzare le mappe irregolari limitate dalle linee congiungenti i punti con le seguenti coordinate (X,Y). Dovrete moltiplicare ogni numero per un appropriato fattore di scala, probabilmente nell'intervallo fra 5 e 25, per fare in modo che la mappa riempia una parte ragionevole di schermo. La figura è stata intenzionalmente data simile ad una scatola quadrettata, per permettervi di simulare le linee grafiche tartaruga usando semplici cicli di FOR o WHILE su mappe a bit o video a caratteri. Se disegnate correttamente la figura risulterà completamente chiusa. In altre parole la linea limite sarà come una barriera senza cancelli. Diamo di seguito le coordinate:

(1,2) (4,2) (4,6) (−1,6) (−1,3) (−3,3)
(−3,5) (−6,5) (−6,1) (−2,1) (−2,−3) (3,−3)
(3,−1) (1,−1) e indietro a (1,2)

Scrivete e mettete a punto un programma per tracciare questa mappa come mappa a bit, o un'equivalente traccia a carattere, e colorate poi la figura non lasciando vuota alcuna posizione adiacente al limite. Supponete che tale figura sia solo una suddivisione di una mappa più grande, con molte suddivisioni adiacenti. Quindi la logica di programma non potrà semplicemente partire dal margine sinistro della figura su ognuna delle linee visualizzate proseguendo verso destra finché non trova un confine ed ombreggiare le posizioni fino a trovare un altro confine. Il limite di tale figura è stato fatto in modo molto irregolare per simulare il mondo reale della costruzione di mappe. La procedura di ombreggiatura dovrebbe essere scritta per gestire qualsiasi mappa disegnata circa nello stesso modo e *non* dovrebbe *essere specifica* di questa particolare mappa!

Attenzione: Questo problema può essere risolto con una semplice procedura recursiva una volta che ogni posizione sul confine è stata allocata da un punto interno alla figura chiusa. Dalla posizione corrente sul limite la procedura controlla se c'è una riga non ombreggiata sulla posizione attuale. Se c'è, la procedura memorizza la sua posizione corrente nelle variabili o parametri locali e chiama nuovamente se stessa passando la nuova locazione da controllare. Se non c'è alcuna posizione non ombreggiata sulla successiva riga più alta, allora l'attenzione si sposta lungo il confine seguendo il verso delle lancette dell'orologio. Se il risultato è un movimento *verso il basso* di una riga, allora la procedura ritorna a chi l'ha chiamata. Il chiamante "sa" il numero di riga (posizione Y) ed entrambe le posizioni orizzontali sinistra e destra del confine. Questo permette di ombreggiare la riga prima che la procedura ritorni *a chi l'ha chiamata.*

Quando termina il primo passo della procedura, il problema è logicamente completato soltanto a metà. La procedura dovrà ora essere richiamata in relazione alle righe *sottostanti* la riga attuale anziché quelle precedenti. La mappa sarà completamente ombreggiata quando la procedura, controllando le posizioni, tornerà alla posizione di partenza.

CAPITOLO 13

RICERCA

1. Obiettivi

In questo capitolo e nei prossimi due verranno proposti, come esempi di problema, algoritmi per la ricerca di lunghe liste di dati e per l'ordinamento di dati non ordinati. Abbiamo suddiviso l'argomento in tre capitoli in modo da concentrare l'attenzione sugli aspetti della soluzione del problema piuttosto che sui dettagli delle tecniche di ordinamento e ricerca. Studiando e capendo gli esempi di algoritmo, e lavorando sugli esercizi, dovreste acquisire esperienza nella programmazione e nella soluzione di problemi.

2. Premessa

Le attività di ricerca e ordinamento occupano probabilmente la maggior parte di tempo macchina su medi e grandi elaboratori rispetto a qualsiasi altro gruppo di problemi elaborativi. A causa degli alti costi del tempo macchina una grossa quantità di studi è stata fatta per ricercare algoritmi di ricerca e ordinamento migliori e più rapidi. Ciò ha condotto ad un'ampia gamma di algoritmi da studiare come esempi in questo campo.

Di norma i problemi di ricerca e ordinamento che si presentano riguardano lunghe sequenze di dati registrati come quelle viste nel Capitolo 10. Esempi familiari di dati da memorizzare come sequenze di registrazione potrebbero essere:

a) schede di consultazione nel catalogo di una libreria

b) righe dell'elenco telefonico che identificano gli abbonati e i loro indirizzi

c) l'indice che si trova alla fine di un qualsiasi manuale.

In ognuno di questi casi un record contiene parecchi dati elementari. Per esempio,

un record nell'elenco telefonico conterà i seguenti dati: cognome, nome, iniziali del secondo nome, numero di telefono, numero civico, nome della strada, città.

Quando cercate nell'elenco telefonico il numero di qualcuno che volete chiamare, "*cercate*" nell'elenco il record corrispondente alla persona che vi interessa. Il numero desiderato si troverà vicino al nome. La compagnia telefonica rende relativamente facile trovare il nome cercato ponendo tutti i nomi in ordine alfabetico. Riuscite a immaginare quali problemi si avrebbero nel trovare un nome se i numeri fossero in qualche altro ordine, ad esempio, in ordine per numeri telefonici crescenti?? Poichè la compagnia telefonica non riceve richieste di nuovi allacciamenti dalle persone secondo l'ordine alfabetico dei loro nomi, è necessario che la compagnia ordini l'insieme delle registrazioni per scrivere un elenco telefonico in sequenza alfabetica.

Se andate negli uffici della compagnia telefonica per chiedere una correzione al numero sul vostro record, l'impiegato userà un elaboratore per cercare e richiamare la registrazione che vi riguarda. Si possono quindi vedere nei problemi di tutti i giorni tipiche applicazioni di tecniche di ricerca e ordinamento in cui sono utili soluzioni su di un elaboratore. Gli algoritmi di ordinamento saranno discussi nei Capitoli 14 e 15: in questo capitolo concentreremo la nostra attenzione sugli algoritmi per ricercare dati già ordinati in qualche sequenza logica.

3. Esame dell'approccio alla soluzione di problemi

Studiando gli algoritmi presentati in questo capitolo, cercate di capire il metodo generale usato qui per poterli applicare ai vostri problemi. Riassumiamo i passi che abbiamo discusso:

a) Se l'intero problema è troppo ampio per poterlo seguire tutto in una volta, suddividetelo in sottoalgoritmi.

b) Sviluppate un'immagine mentale, per ogni sottoalgoritmo, di cosa bisogna fare passo dopo passo. Per fare questo può essere spesso utile considerare un piccolo insieme di dati di prova scritti su carta o su un diagramma. Seguite passo passo il processo mentale di conversione di questi dati dal formato iniziale a quello che volete sia il risultato del sottoalgoritmo. Una volta che avrete capito *cosa* deve fare il sottoalgoritmo, passate al passo successivo di decidere *come* lo farà.

c) Disegnate ora un diagramma di struttura che mostri, in modo ordinato ma approssimativo, *come* avete fatto la conversione mentale dei dati al passo (b). Fate attenzione ai passi *ripetuti* due o più volte, a dove dovete decidere

quale azione intraprendere fra due o più (blocchi *decisionali*), e a sequenze di azioni uguali in posti diversi del diagramma (*sottoalgoritmi*). Potreste iniziare con un diagramma sommario per costruirvi un'idea generale di come l'algoritmo dovrebbe girare; potreste poi aggiungere dettagli, e correggere gli aspetti confusi della vostra logica.

d) Dopo aver sviluppato un diagramma di struttura potete ora convertirlo nelle istruzioni del linguaggio di programmazione usato. Questo sarà un mero problema di programmazione se programmate in PASCAL; altrimenti vi potrà essere utile convertire dapprima il diagramma in PASCAL. Dopodichè, con un semplice insieme di regole di conversione, potete convertire PASCAL in un qualsiasi altro linguaggio di programmazione.

e) Ora dovete mettere a punto il programma scritto nel passo (d). Il primo passo nella messa a punto è la correzione degli errori di sintassi segnalati dal compilatore. Il secondo passo, e il più difficile, è la scoperta, "*durante l'esecuzione*" di errori logici nel programma. Questo richiede l'uso di dati di prova per cui si conoscono già i risultati del programma o del sottoprogramma. A questo scopo possono servire i dati di prova usati nel passo (b) per farsi un'idea di cosa bisognava fare. Soltanto quando si sarà arrivati al corretto funzionamento di ogni procedura e funzione con una piccola quantità di dati, si potrà tentare di metterli assieme nel programma; non si dovranno inoltre usare grossi gruppi di dati. Il programma potrà essere usato per "fare" elaborazioni soltanto quando sarà stato fatto un controllo abbastanza accurato con i dati di prova.

4. Ricerca lineare

In questo capitolo e nei successivi ci occuperemo della ricerca, o dell'ordinamento, di liste di *numeri.* Nelle applicazioni effettive capiterà più spesso di cercare, o ordinare sequenze di nomi o altre parole di un testo. Nel nostro caso useremo i numeri semplicemente per rendere più facile la programmazione, e per permetterci di concentrare l'attenzione sulla struttura degli algoritmi considerati.

Per farci un'idea della ricerca lineare, consideriamo una piccola sequenza di dati di prova:

```
86787, 3831, 5138, 1064, 0730, 4308, 4687, 8540, 3667, 9430
  0     1     2     3     4     5     6     7     8     9
```

Se vi sembra una serie di numeri estratti dall'indice telefonico locale, sappiate che

è esattamente quello. L'elenco telefonico è spesso una buona fonte di numeri casuali per il controllo di algoritmi. Sotto ogni dato della sequenza è mostrato il suo indice, cioè il numero d'ordine con cui il dato compare. Chiaramente prevediamo l'uso di quest'indice se i dati devono essere memorizzati in un vettore sull'elaboratore.

In una ricerca "lineare" o "sequenziale", per cercare un dato in una lista, prenderemo in considerazione ogni dato nell'ordine, a partire dall'inizio (o dalla fine se è più conveniente); ci fermeremo al raggiungimento del dato cercato. Potremmo ad esempio voler cercare, nella lista vista sopra, il numero 4308. (Se non vi piace cercare un numero specifico, tentate di ricordare un'occasione in cui avete annotato un numero di telefono da usare in futuro, ma avete dimenticato di segnare il nome della persona che vi ha dato il numero. In seguito avete dovuto cercare chi fosse quella persona). Iniziamo alla locazione "0" con il dato 8687; poichè non è quello cercato, continuiamo con la successiva, trovando 3831. Continuiamo questo procedimento scartando le registrazioni di numeri diversi da quello cercato. Continuando a ripetere tale procedimento speriamo di trovare il dato che stiamo cercando. Nel caso tipico è naturalmente possibile che il dato cercato non sia presente nella lista. Dobbiamo quindi costruire l'algoritmo in modo che abbia la possibilità di fermarsi con un'indicazione di "esito fallito" se si raggiunge la fine della lista senza aver trovato il dato desiderato.

L'immagine mentale di questo procedimento è abbastanza semplice e possiamo procedere direttamente alla stesura del diagramma di struttura che lo descrive. Chiaramente nel diagramma di struttura ci sarà un blocco di ripetizione. Ad ogni ripetizione, avremo bisogno di incrementare di 1 una variabile indice che tenga conto di dove siamo arrivati nella lista. La ripetizione continuerà finché non si troverà una delle due situazioni:

a) si raggiunge il dato cercato
b) si raggiunge la fine della lista

La condizione (a) specifica che la ricerca finirà se si raggiunge il dato cercato. Naturalmente è possibile rendere un po' più semplice l'algoritmo permettendogli di prendere in considerazione *ogni* dato della lista senza preoccuparsi del fatto che il dato sia stato effettivamente trovato. In questo caso si dovrebbe porre TRUE una variabile di "segnalazione" (FLAG) al ritrovamento del dato cercato. Comunque non è chiaro cosa dovreste fare al raggiungimento del dato cercato se state scandendo visivamente una lista. A quel punto dovreste interrompere la ricerca se sapete che il dato cercato è presente una sola volta nella lista; sarebbe una perdita di tempo continuare la scansione della lista dopo aver trovato il dato. Lo stesso ragionamento si può fare per gli algoritmi di ricerca sugli elaboratori dal momento che è costoso sprecare tempo macchina senza necessità (almeno su grossi elaboratori). Pensando ai dati come memorizzati in un vettore sarà istruttivo prevedere l'interruzione della ripetizione sul raggiungimento di uno speciale carattere di "fine" piuttosto che sulla condizio-

ne EOF. Tipicamente tale carattere verrà posto alla fine della lista di dati al momento del caricamento iniziale in memoria da parte del programma principale. Tutto ciò porta all'algoritmo (funzione booleana) mostrato in Figura 13-1. In figura è mostrato come un sottoalgoritmo, con parametro VOCE, che fornisce il valore del dato da cercare.

Il valore della funzione sarà posto FALSE se la ricerca fallisce, altrimenti sarà posto TRUE. Il valore del carattere di fine deve essere un qualche valore che logicamente non può essere trovato nella lista di dati. Nel caso della lista di interi positivi a 4 cifre vista sopra un carattere di fine accettabile potrebbe essere −1 (dal momento che non esistono numeri di telefono negativi), 0 (dal momento che non si può ragionevolmente supporre che il numero 0000 non sia mai usato come numero di telefono), o 10000 o un numero maggiore (poichè questi interi sono a più di 4 cifre).

Studiate la Figura 13-1, osservando i seguenti punti:

a) Avendo deciso di usare il blocco di iterazione

REPEAT azione UNTIL condizione

la locazione del vettore LST[I] menzionata nella condizione è controllata alla fine dell'esecuzione dell'azione ripetuta. Poichè l'azione deve includere un'istruzione che incrementa il valore della variabile indice I ad ogni iterazione, I deve essere inizializzato a −1 prima che inizi il blocco REPEAT. Altrimenti la prima locazione controllata sarebbe LST[1].

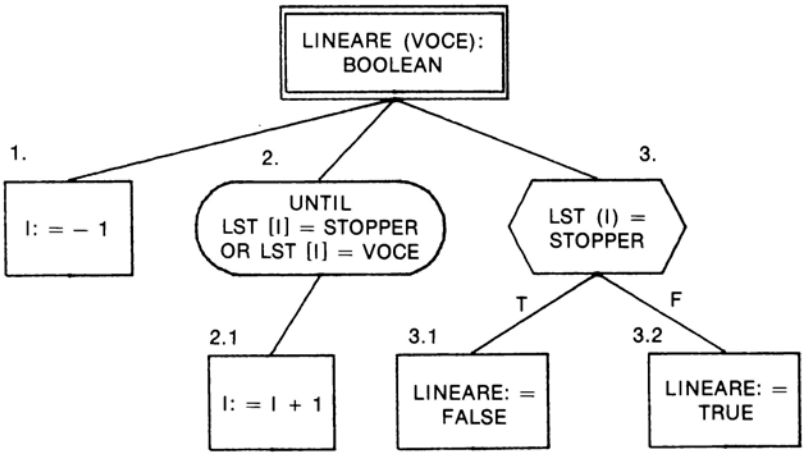


Figura 13-1

b) La scelta nel blocco 3 è necessaria per decidere se il blocco REPEAT è terminato perchè è stato trovato il dato cercato, o se la ricerca è fallita.

La Figura 13-2 mostra un altro modo di esprimere la ricerca lineare. Studiate questa versione, e confrontatela con la figura 13-1, notando i seguenti punti:

a) Il prezzo della condizione più semplice nel blocco REPEAT è l'uso di un blocco EXIT, o di un equivalente GOTO, per riuscire ad uscire dal ciclo se si avvera la seconda condizione.

b) È leggermente più semplice porre LINEARE al valore FALSE come parte dell'inizializzazione invece di fare un controllo su come termina il blocco REPEAT. LINEARE resta FALSE a meno che non sia trovato il dato cercato, nel qual caso viene eseguito il blocco 2.2.1.

c) Se avessimo usato un blocco #3 per porre LINEARE al valore FALSE avremmo dovuto riarrangiare l'EXIT per uscire dal nodo origine del sottoalgoritmo per evitare il blocco #3 se si trova il dato cercato.

Per controllare il vostro programma in entrambe le forme dell'algoritmo per la ricerca lineare, usate la piccola lista di dati vista prima. Usate la funzione per cercare i seguenti numeri:

a) 4308 (nella parte centrale della lista)

b) 8687 (primo valore della lista)

c) 9430 (ultimo valore della lista)

d) 7777 (valore accettabile, ma non nella lista)

e) −123 (valore non accettabile perchè non appartenente all'intervallo di valori accettati)

Fate in modo che il programma principale visualizzi il valore della variabile I al termine della funzione LINEARE. Controllate, in ognuno dei casi visti prima, se il valore di I riportato dal vostro programma concorda con il dato di prova.

Supponiamo ora che il vostro programma giri, ma non dia risultati corretti e non riusciate a trovarne la ragione studiando le istruzioni del vostro programma. Quando compare questo problema è spesso utile "*presentare*" i valori delle variabili usate dalla funzione. Per fare questo aggiungete istruzioni temporanee WRITE nei punti strategici. Potete ora studiare i valori in uscita passo dopo passo e controllare che i

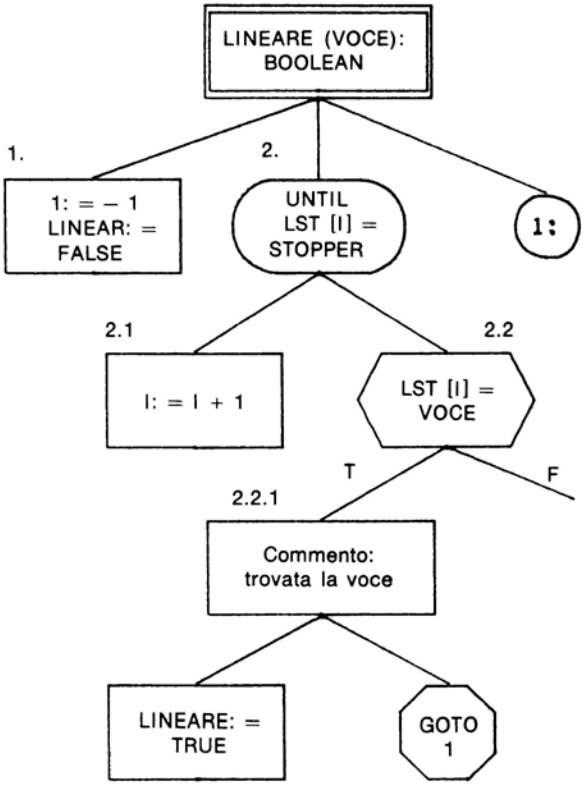


Figura 13-2

risultati dell'elaboratore siano quelli che vi aspettate. Generalmente quest'analisi passo passo dovrebbe aiutarvi a scoprire velocemente un errore.

5. Ricerca Binaria

Sebbene sia semplice, il procedimento di ricerca lineare discusso nella sezione precedente è troppo lungo, a meno che la lista contenga non più di poche dozzine di valori. Voi stessi non fareste mai una ricerca lineare su un elenco telefonico o su un dizionario. Ovviamente occorrerebbe troppo tempo per considerare ogni valore (un nome in un elenco telefonico, una parola in un dizionario) cominciando dall'inizio e continuando finché non si trovi il valore cercato. Affrontereste invece il problema in un altro modo che dall'esperienza si sa, essere più veloce. Per la stessa ragione ge-

neralmente non programmiamo un elaboratore per fare una ricerca lineare su grosse liste. Il metodo usato è invece simile alla logica con cui *voi* cercate un elemento nell'elenco telefonico o in un dizionario. Nella sua forma più semplice, questo metodo è detto di"*ricerca binaria*". Per farvi un'idea della ricerca binaria, pensate a come fareste a trovare la pagina dell'elenco telefonico contenente un nome che state cercando. Supponiamo che il nome sia "Strozzi". Cerchereste dapprima la parte dell'elenco riguardante la "S" aprendolo ad una pagina circa a metà; controllereste poi in base a quella pagina se la lettera "S" è prima o dopo nell'elenco. Se fosse dopo, come probabilmente sarebbe per la "S", restringereste poi la ricerca dividendo la seconda metà del libro in due parti della stessa grandezza. In questo modo trovereste una nuova pagina e controllereste se tale pagina è nella parte relativa alla "S" dell'elenco. Se invece aveste aperto ad una pagina di nomi inizianti per "P", continuereste il procedimento di suddivisione guardando più avanti nell'elenco. Guardereste approssimativamente a metà del quarto di libro ottenuto dalla suddivisione precedente. Se invece foste arrivati alla sezione "T" del libro, dovreste guardare un po' prima.

Continuando questo processo di suddivisione, arriverete ad una parte di libro in cui i nomi iniziano per "S". Ora dovete restringere ulteriormente la ricerca cercando dapprima i nomi che iniziano per "ST", poi per "STR", suddividendo ad ogni passo la prima o la seconda parte della porzione sempre più piccola del libro appena trovata. Quando raggiungete la pagina contenente i nomi che iniziano per "STROZ" sarete probabilmente abbastanza vicini perchè valga la pena di usare una ricerca lineare del nome desiderato all'interno della pagina.

Mentre una ricerca lineare non richiede alcuna ipotesi sull'ordine in cui sono posti gli elementi all'interno della lista, una ricerca binaria richiede che gli elementi siano in ordine crescente o decrescente. Nel caso dell'elenco telefonico, o del dizionario, la sequenza è la familiare sequenza alfabetica. Per farcene un'idea, usiamo invece una sequenza ordinata di numeri a 3 cifre, in cui ogni numero sia maggiore o uguale al suo predecessore nella lista. Di seguito diamo la sequenza che useremo per spiegare l'algoritmo, ancora una volta ogni elemento è accompagnato dagli indici di locazione:

```
105, 172, 221, 279, 324, 324, 331, 392, 426, 439, 541, 615, 684
  0   1   2   3   4   5   6   7   8   9   10  11  12
```

Figura 13-3

Seguiamo ora il processo di ricerca binaria su questa lista, esattamente come l'abbiamo descritto. Supponiamo di voler cercare il valore 392.

```
passo 1:
105, 172, 221, 279, 324, 324, 331, 392, 426, 439, 541, 615, 684
 0    1    2    3    4    5    6    7    8    9   10   11   12
                                   :
passo 2:
                                        392, 426, 439, 541, 615, 684
                                         7    8    9   10   11   12
                                              :
passo 3:
                                        392, 426
                                         7    8
                                         :
```

Figura 13-4

Nel primo passo abbiamo tagliato la lista all'elemento 6, e trovato che il valore 331 lì contenuto era minore del valore cercato. Eliminato 331, già esaminato e rifiutato, cerchiamo nel passo 2 all'interno della metà superiore della lista. Poniamo il "centro" di questa metà all'elemento 9; il centro esatto sarebbe 9.5 ma non abbiamo elementi divisi. Di conseguenza abbiamo preso la decisione arbitraria di considerare l'intero vicino più basso, cioè 9, come nuovo "centro". Il valore in esso trovato è 439, che è maggiore del valore cercato. Eliminando il valore 439, siamo rimasti con gli elementi 7 e 8 come soli superstiti della lista. Prendiamo il "centro" di questa nuova sequenza più piccola, con la stessa regola del passo 2, ottenendo 7.5 che riduciamo a 7. Avendo trovato che il valore alla locazione 7 è 392, il valore cercato, la ricerca termina dopo il passo 3.

Seguiamo ora lo stesso procedimento per un altro elemento, diciamo 279, come in Figura 13-5.

```
passo 1:
105, 172, 221, 279, 324, 324, 331, 392, 426, 439, 541, 615, 684
 0    1    2    3    4    5    6    7    8    9   10   11   12
                                   :
passo 2:
105, 172, 221, 279, 324, 324
 0    1    2    3    4    5
           :
passo 3:
          279, 324, 324
           3    4    5
                :
passo 4:
               279
                3
                :
```

Figura 13-5

Dovremmo ormai esserci fatta un'idea sufficiente su come sviluppare il diagramma di struttura dell'algoritmo per questo tipo di ricerca. Sarà comunque utile, prima di procedere con l'algoritmo, vedere cosa succede cercando un valore non presente in lista, diciamo 329. Questo è mostrato in Figura 13-6.

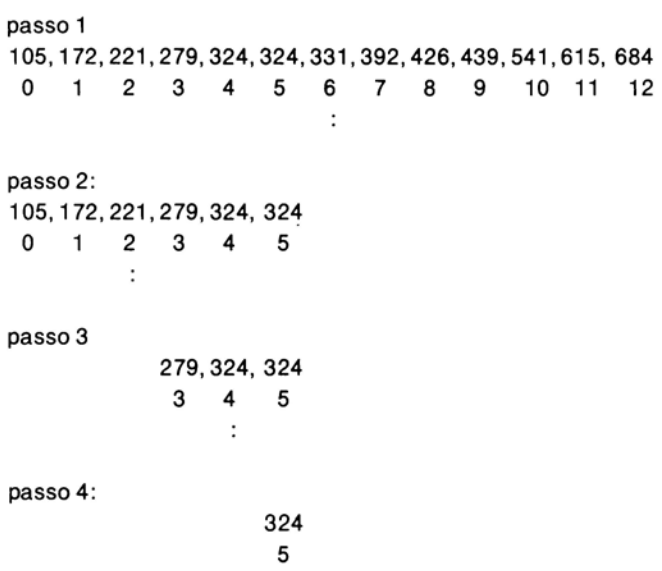


Figura 13-6

6. Algoritmo recursivo di ricerca binaria

Proseguiamo ora con la stesura del sottoalgoritmo per la soluzione del problema della ricerca binaria sull'elaboratore. Poniamo che la lista di dati sia un vettore LST, e di sapere gli indici del primo e dell'ultimo dato nella lista, indici che passeremo al sotto-algoritmo come parametri. Nel nostro esempio tali valori sarebbero stati 0 e 12.

Da uno studio sommario delle figure, dalla 13-4 fino alla 13-6, è ovvia la possibilità di utilizzare un blocco di ripetizione per controllare il numero di passi da fare. È comunque più semplice pensare al sotto-algoritmo sotto forma recursiva. Più dettaglia-

tamente: nel passo 1 abbiamo suddiviso la lista in 3 parti (la "metà" inferiore, il "centro", la "metà" superiore) e, nella maggior parte dei casi si deciderà di esaminare una delle tre parti. Se la nuova parte da esaminare è la metà inferiore o quella superiore, quella parte potrà essere considerata una nuova lista in cui cercare usando lo stesso sotto-algoritmo. Ad esempio, nella Figura 13-6 il risultato di una ricerca binaria sulla lista LST[0], ...,LST[12] è che si dovrà fare una ricerca binaria sulla lista LST[0], ...,LST[5]; ma il risultato di una ricerca su LST[0], ...,LST[5] è la necessità di una nuova ricerca binaria su LST[3], ...,LST[5]. Finalmente il risultato del passo 3 è la necessità di una ricerca binaria su una lista di, al massimo, un elemento. Avendo raggiunto questo punto, non sarà necessario alcun uso ulteriore del sotto-algoritmo. La figura 13-7 mostra la forma recursiva del sotto-algoritmo.

Di seguito si danno alcune annotazioni circa la Figura 13-7:

a) Il termine "chiave"

In figura 13-7 l'identificatore "CHIAVE" è il valore per cui si richiede una ricerca. Il termine "*chiave*" è spesso usato nei problemi di ricerca e ordinamento per indicare l'elemento da cercare o da usare per determinare l'ordine finale della lista. La chiave usata per ordinare un elenco telefonico è formata dalla concatenazione di:

Cognome Nome Iniziale del secondo nome

Se la chiave usata nella preparazione dell'elenco telefonico fosse invece il numero di telefono, l'elenco sarebbe ordinato secondo la sequenza numerica crescente dei numeri di telefono. Allo stesso modo la chiave usata nell'ordinamento di un dizionario è la prova da definire. Tutto degli altri dati entra "procedendo sul sentiero" quando la lista delle definizioni-parola è ordinata per produrre un dizionario.

b) Fine dell'algoritmo

La sequenza di chiamate al sotto-algoritmo RICERCABINARIA si interrompe quando l'ultimo blocco eseguito non è un'altra chiamata a RICERCABINARIA. Questo significa che l'esecuzione del blocco 2.2, o il risultato negativo nell'esecuzione dei blocchi 2.1.1.1 o 2.1.2.1 provocherà la fine dell'algoritmo lasciando la variabile LOC (per locazione) settata a qualche valore. LOC è una variabile globale dichiarata nel <blocco> che richiama RICERCABINARIA la prima volta. Prima di richiamare RICERCABINARIA, LOC è inizializzata a −1. Se viene trovato il valore cercato, LOC è l'indice della locazione del vettore LST contenente quel valore. Se il valore non viene trovato in alcuna locazione della lista, allora LOC resta a −1, valore che non può corrispondere ad alcuna locazione nel vettore.

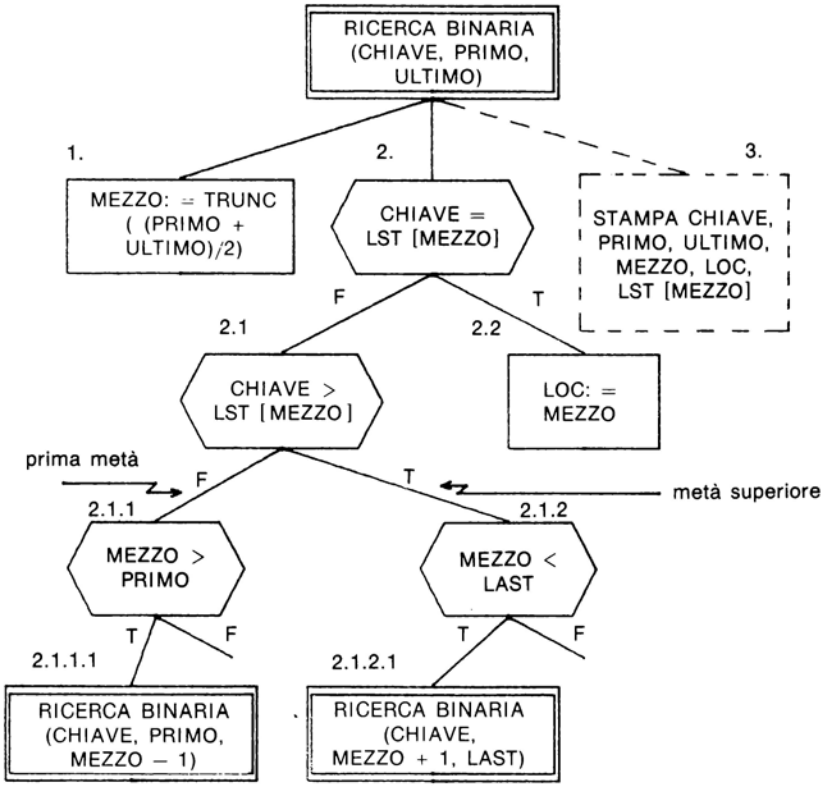


Figura 13-7

ESERCIZIO 13.1:
La tecnica di recursione si è mostrata utile per spiegare come operi la ricerca binaria, ma lo stesso algoritmo può essere scritto senza recursione. Scrivete e mettete a punto un programma che includa una funzione intera RICERCABINARIA il cui valore in uscita sia la locazione della chiave. La funzione dovrebbe essere non-recursiva.

Controllate tale programma con la stessa lista di dati usata in questo capitolo per spiegare gli obiettivi. Dopo la lettura dalla tastiera della lista di dati, controllate il programma con i seguenti valori della chiave:

a) 105 (il più piccolo valore nella lista)
b) 684 (il più grande valore nella lista)
c) 331 (valore della locazione centrale)

d) 426 (valore vicino al centro della lista, ma non il valore centrale)

e) 200 (valore non presente nella lista, ma che, se presente, sarebbe prossimo al centro)

f) 0 (valore minore del più piccolo valore della lista)

g) 999 (valore maggiore del più grande valore della lista)

h) 324 (valore che è presente due volte)

Modificate ora l'elenco di dati per una seconda prova, rimuovendo *uno* dei valori 324 e ripetete ancora le stesse prove (I programmi di ricerca binaria tendono ad avere difficoltà su lunghezze dispari/pari). Assicuratevi che i risultati in entrambi i casi siano corretti. Soltanto quando saranno tutti esatti, il programma potrà girare correttamente su liste più lunghe.

Generate ora una lista di tutti i numeri da 1 a 1000 e provate nuovamente su questa lista la funzione per la ricerca. Provate a cercare in questa lista le chiavi 497, 498, 499, 500, 501, 502 e 503. In questo caso la posizione di ciascuna chiave dovrebbe essere uguale alla chiave, o minore di 1, a seconda di come è stato definito il vettore LST. Per quest'ultima prova aggiungete delle istruzioni che consentano di contare il numero di volte che la procedura confronta la chiave con LST [MEZZO], e di stampare il valore di tale contatore alla fine della funzione. Nonostante la lunghezza della lista, nessuna delle chiavi nel breve elenco sopra scritto dovrebbe richiedere più di 10 confronti. Generalmente la ricerca binaria dovrebbe richiedere meno di

$$Log_2N$$

passi di confronto (arrotondati per eccesso), dove N è il numero di elementi della lista. La ricerca lineare richiederà in media circa N/2 confronti, valore molto maggiore.

CAPITOLO 14

ORDINAMENTO I ALGORITMI SEMPLICI

1. Obiettivi

Presentiamo in questo capitolo due semplici algoritmi di ordinamento e l'argomento strettamente collegato della fusione. Il vostro principale obiettivo dovrebbe essere quello di migliorare la vostra esperienza nella soluzione di problemi scrivendo e correggendo dei programmi che eseguono questi algoritmi.

2. Premessa

La ricerca è una delle principali ragioni per cui è desiderabile ordinare elenchi di dati in una qualche sequenza d'ordine. Un'altra ragione è associata al problema con cui ci si scontra quando si cerca di portare pochi "cambiamenti" in dati già inseriti in un lungo elenco. Se entrambe le liste, la principale e quella dei cambiamenti, sono nella stessa sequenza ordinata (ad esempio in ordine alfabetico o numerico) sarà necessario scorrere una sola volta le due liste per introdurre i cambiamenti nei dati originari della lista principale. Quest'ultima applicazione è chiamata "*fusione*" ("*merging*"), argomento che tratteremo brevemente.

In parte perchè gli algoritmi efficienti di ordinamento portano risparmi sui tempi di elaborazione, e in parte perché gli algoritmi di ordinamento offrono un'interessante sfida intellettuale agli "uomini dell'informatica", ci sono molti algoritmi popolari di ordinamento, regolarmente in uso. La scelta dell'algoritmo da usare può dipendere dal tipo di elaboratore disponibile, dall'ordinamento parziale supposto nei dati da ordinare, dal genere di dati da ordinare, dalla lunghezza della lista e da molti altri fattori, per non parlare delle preferenze estetiche del programmatore.

In questo capitolo analizzeremo brevemente due algoritmi di ordinamento per darvi un'idea della possibile varietà. Si studino attentamente questi algoritmi perché daranno idea di come scrivere algoritmi di crescente complessità.

3. Ordinamento per inserzione

Il semplice algoritmo di "ordinamento per inserzione" è a volte chiamato "ordinamento per affondamento" ("sinking sort") e con piccoli cambiamenti "ordinamento per setacciamento" ("sifting sort").

L'algoritmo è molto simile a quello usato per ordinare una "mano" di carte da gioco per valori crescenti. In tal caso si inizia con la prima carta distribuita e, presa la seconda, la si inserisce prima o dopo la prima carta. Presa la terza, la si può inserire

LOC	LIST ORIGINALE											
0	43											12
1	87										12	23
2	29									12	23	29
3	38								12	23	29	34
4	34							12	23	29*	34	38
5	46						12	23	34	34	38	39
6	12					12*	23	34*	38*	38	39	43*
7	68				23	23	39	39	39	39	46	46
8	23			23*	39	39	46*	46	46	46	55	55
9	75		39	39	55	55	55	55	55	55	68	68
10	55	39	55	55	68*	68	68	68	68	68	75	75
11	39*	55*	75*	75	75	75	75	75	75	75	87*	87
PASSO	1	2	3	4	5	6	7	8	9	10	11	12

Figura 14-1

a seconda del valore in mezzo alle prime due, come prima o come ultima. Presa la quarta, la si confronta, cominciando dalla prima, con ognuna di quelle già nel mazzetto; per ogni nuova carta si ripete questo procedimento fino ad esaurire tutte le carte della mano. A quel punto la "mano" di carte è in sequenza ordinata.

Ancora una volta useremo i numeri per illustrare l'algoritmo, anche se esso lavora fondamentalmente nello stesso modo se i dati sono nomi o altri elementi non numerici. La Figura 14-1 mostra l'ordinamento per inserzione in funzione.

Si osservi che il nuovo elemento introdotto ad ogni passo è messo in evidenza con un asterisco ('*'). Si noti anche che i numeri più grandi (i più pesanti) affondano prima verso il "fondo" della lista. In questo caso abbiamo scelto di rappresentare il fondo come il valore più alto dell'indice di locazione LOC.

Se fatto come mostrato, l'ordinamento per inserzione può essere effettuato nelle locazioni di tabella occupate dalla lista originaria di dati. Occorre una sola variabile, chiamiamola TIENE, in cui porre temporaneamente il valore da inserire. Quindi si inizia dalla cima (in quest'esempio il valore di indice più basso) dell'elenco costruito finora e si confronta il valore posto in TIENE con quello nella lista. Se il valore nell'elenco è più piccolo, allora quel valore viene spostato in alto di una locazione. TIENE viene ora confrontato con il valore successivo nella lista. Il procedimento continua spostando verso l'alto di una locazione tutti i valori più piccoli. Quando si trova nella lista un valore più grande di TIENE, il valore di TIENE viene memorizzato nella locazione immediatamente sopra il valore più grande.

La Figura 14-2 illustra il diagramma di struttura dell'ordinamento per inserzione. Quando il sotto-algoritmo inizia, i dati non ordinati sono nel vettore LST che va da LST[0], ...,LST[N]. Le variabili TIENE, I e J possono essere variabili locali del sotto-algoritmo dal momento che servono solo per la durata del procedimento di ordinamento.

Allo scopo di confrontare questo algoritmo con gli altri algoritmi di ordinamento che verranno studiati, facciamo una stima del numero di elaborazioni ripetute necessarie per sviluppare l'ordinamento per inserzione. Il blocco 1 in fig. 14-2 fa eseguire N ripetizione dove i dati da ordinare sono N+1. All'inizio il passo 1.b fa eseguire 1 sola ripetizione. Alla fine il passo 1.b fa eseguire al massimo N iterazioni. Il passo 1.b è terminato se la locazione per un nuovo valore viene trovata prima della fine della lista col GOTO del blocco 1.b.1.b.2. Questo è un esempio di un uso di GOTO che non è stato discusso nel capitolo 11. In questo caso il GOTO è usato per uscire da un *ciclo* di ripetizione dentro la parte di algoritmo controllata dal blocco 1. In questo modo il procedimento riprenderà dalla scatola 1.a a patto che un'altra ripetizione sia chiamata dalla scatola 1.

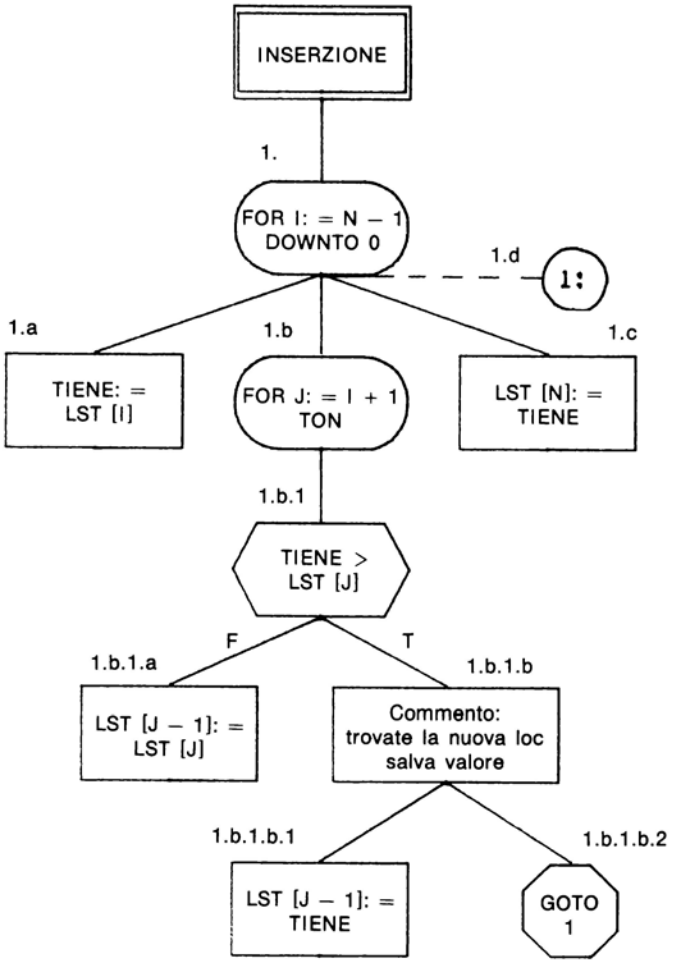


Figura 14-2

Al massimo il numero di ripetizioni richiamate dalla scatola 1.b (che è ripetuta sotto il controllo della scatola 1.) sarà N*(N/2), cioè la metà del quadrato di N. Questo capiterebbe solo se la lista originaria fosse in ordine esattamente inverso rispetto all'ordine desiderato. Se fosse vero, allora ogni nuovo elemento dovrebbe "affondare" in fondo alla lista. Nel caso più tipico, ogni nuovo elemento dovrà affondare a circa metà della lista fino ad allora ordinata. Perciò l'algoritmo richiede circa N*(N/4) ripetizioni della scatola 1,b.1. Se la lista originale fosse già nella corretta sequenza ordinata all'inizio del sotto-algoritmo, saranno comunque necessarie N ripetizione del blocco 1.b.1 per terminare il sotto-algoritmo.

Per completare l'immagine mentale di ciò che succede in questo algoritmo, potreste fare i primi 5 o più passi di figura 14-1, notando cosa avviene nel diagramma di struttura di fig. 14-2 di blocco in blocco. Vedete la necessità del blocco 1.c.??

Per controllare la correttezza di un programma scritto per effettuare l'ordinamento per inserzione, è consigliabile aggiungere temporaneamente istruzioni di visualizzazione ai blocchi 1.b.1.a, 1.b.1.b.1 e 1.c. Ogni istruzione di visualizzazione dovrebbe includere l'identificazione della sua locazione con il numero di blocco. Dovrebbe inoltre fornire i valori di LST[J−1], LST(J] e TIENE. Potreste aggiungere istruzioni READLN per fermare temporaneamente l'elaborazione mentre studiate i risultati degli ultimi cicli; battendo <RET> si permetterà all'elaborazione di continuare.

Attenzione: Il valore di J seguente la fine del construtto FOR ... nel blocco 1.b non è ben definito neanche in PASCAL. A seconda dell'implementazione del linguaggio, il valore della variabile di controllo dopo la fine di un'istruzione FOR, può o non può essere usato; inoltre, sebbene usabile, tale valore può essere uguale al limite dell'istruzione FOR, maggiore o minore di 1 rispetto al limite, o anche qualche altro valore. Quindi non si dovrebbe far affidamento su J dopo la fine di un'istruzione FOR finchè non si assegni a J un nuovo valore.

Se il precedente approccio di controllo non fosse sufficiente per visualizzare l'andamento dell'algoritmo, potrebbe valer la pena di aggiungere altre istruzioni di controllo in modo da poter seguire ad ogni passo i risultati del processo di ordinamento. Per fare questo aggiungete un'istruzione come blocco 1.d che visualizzi l'insieme di elementi della lista compresi fra la locazione I e la N. Potreste stampare la lista di elementi orizzontalmente su una sola riga di uscita, anziché verticalmente come fatto nella Fig. 14-1. Ovviamente potrete fare controlli a questo livello di dettaglio soltanto su liste di dati relativamente corte.

4. Ordinamento a bolla (Bubble sort)

L'algoritmo di ordinamento a bolla appartiene alla famiglia di algoritmi che "*scambiano*" valori vicini per ottenere il riarrangiamento dei dati. La descrizione "a bolla" implica che i valori più leggeri (più piccoli) "si alzino" in cima alla lista. Se i dati sono parzialmente ordinati, l'ordinamento a bolla è relativamente veloce poiché permette di riconoscere quando non sono necessari ulteriori scambi. In altre parole, è abbastanza simile al semplice ordinamento per inserzione descritto nella Sezione 3. Osservate la fig. 14-3 per avere un'idea di come funzioni l'ordinamento a bolla.

Ad ogni passo in questa figura le "bolle" sono evidenziate da un asterisco ('*'). Per esempio, consideriamo il passo 1: nella lista originale, essendo 39 più piccolo di 55, i

due sono scambiati. Quindi 39 è scambiato con 75; al prossimo confronto 39 è più grande di 23 e così non viene operato alcuno scambio. Poichè 23 è più piccolo di 68, 23 diventa la prossima bolla; finalmente 12, essendo più piccolo di ogni altro elemento, risale tutta la lista fino alla cima.

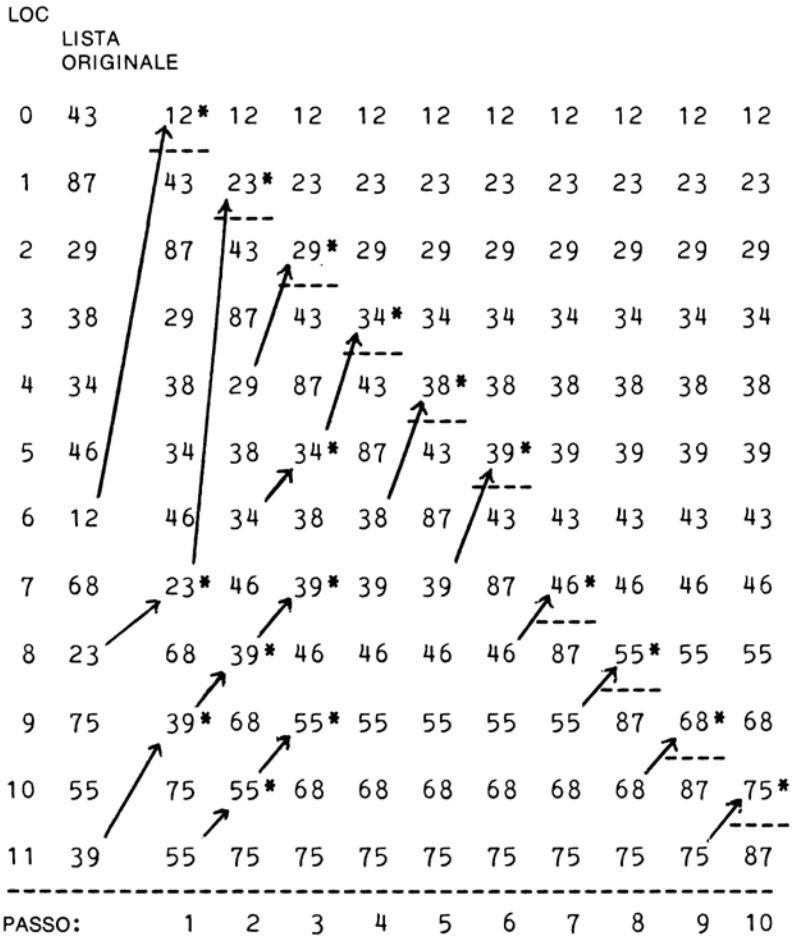


Figura 14-3

Nel preparare il passo 2, sappiamo che il valore più piccolo di tutta la lista è stato spostato nella locazione 0; non ha quindi alcun significato il fare ulteriori confronti con il valore di quella locazione. Perciò, nel passo 2, l'ultimo valore in cima alla lista

che deve essere confrontato è quello alla locazione 1, inizialmente di valore 43. A seguito del procedimento il prossimo valore più piccolo nella lista sale nella locazione 1. Al passo 3 occorrerà soltanto confrontare fino alla locazione 2. Ad ogni passo la linea tratteggiata orizzontale mostra dove è stato fatto l'ultimo confronto il cui risultato sia uno scambio di valori. Nel passo 7 l'ultimo scambio poteva essere fatto fra gli elementi 6 e 7 (43 e 46), ma poichè essi erano già nell'ordine corretto, non è stato necessario alcun scambio. Questo ha permesso di accorciare la lista rimanente di 2 locazioni in un solo passo.

Potete tener conto ad ogni *passo* di dove è avvenuto l'ultimo scambio nella lista con una variabile semplice SCAMBIO. Inizializzate SCAMBIO a N—1 all'inizio di un passo; alla fine del passo ponete una variabile LIMITE al valore di SCAMBIO+1. Nel passo successivo basterà andare dall'N— sino elemento fino a LIMITE.

Potete vedere che l'ordinamento a bolla potrebbe richiedere relativamente poche operazioni se la lista originaria di dati fosse all'inizio in una sequenza pressochè ordinata. Come esempio consideriamo la lista nel passo 10 di figura 14-3, immaginando però che un nuovo valore, il 10, si sia aggiunto alla lista tra i valori 55 e 68. Sarebbe sufficiente soltanto un passo per far salire il valore 10 in cima alla lista; al secondo passo non ci sarebbe alcun scambio. Alla fine di quel passo, la variabile SCAMBIO sarebbe ancora a N—1, LIMITE sarebbe perciò posto a N e la ripetizione terminerebbe.

Potrebbe anche capitare che un valore grande capiti vicino alla cima di una lista quasi ordinata. Come esempio contrario a quello appena discusso, supponiamo che il valore 90 sia inserito proprio sotto il 12 nella lista ordinata del passo 10 (Figura 14-3). In questo caso il numero di passi sulla lista necessari a spostare il 90 nella sua posizione corretta in fondo alla lista sarebbe N—1. Questo suggerisce che l'ordinamento a bolla funzioni meglio per piccoli valori che devono salire in cima alla lista piuttosto che per grandi valori che devono scendere in fondo alla lista.

Un rimedio per questo potrebbe essere fare prima un passo verso l'alto e quindi farne uno verso il basso. Questa procedura potrebbe essere seguita in coppie con entrambi i valori superiori e inferiori di LIMITE che divengono progressivamente più vicini l'uno all'altro. Nei passi di numero dispari gli scambi inizierebbero dal più basso LIMITE e si opererebbe verso l'alto fino alla cima come in figura 14-3. Nei passi di numero pari gli scambi inizieranno dal più alto LIMITE (nel diagramma) e si opererebbe verso il basso fino al fondo. L'algoritmo termina quando il limite inferiore diventa uguale a quello superiore. Questa versione «avanti e indietro" dell'ordinamento a bolla è talvolta detto "ordinamento per miscelazione" ("Cocktail Shaker Sort"). Esso ha il vantaggio di finire in modo relativamente veloce se i dati originali sono in un ordine quasi corretto qualunque sia la posizione dei pochi elementi fuori posto.

5. Fusione (Merging)

Nei problemi di ricerca e di ordinamento ci si trova spesso di fronte alla necessità di combinare due o più liste di dati in sequenza ordinata. Se le liste sono già ordinate prima di scambiarle, il sistema più efficiente e diretto per unirle è *"fondere"* ("to merge") le liste. Si supponga di voler combinare due liste già ordinate in modo "crescente" (il primo elemento è il più piccolo) che chiameremo lista A e lista B fondendole in una nuova lista M. Si confronterà il primo elemento della lista A con il primo della lista B e il più piccolo dei due verrà spostato nella lista M; si provvederà anche a cancellare tale elemento nella lista da cui proviene. A questo punto si farà un altro confronto fra i primi elementi presenti nelle due liste e si aggiungerà il più piccolo nella lista M. Alla fine tutti gli elementi saranno stati spostati da questo procedimento nella lista M e le liste A e B saranno entrambe vuote. La figura 14-4 dà un esempio di questo processo: si noti che non è necessario che le due liste abbiano la stessa lunghezza.

Di seguito vengono date alcune note riguardo questo processo:

a) Occorrono alcune regole per dirimere la situazione quando il primo valore presente nella lista A è uguale al primo presente in B. In questo esempio, quando ci si è trovati in questa situazione si è scelto come primo l'elemento appartenente alla lista A.

b) Prima o poi vi capiterà di esaurire tutti gli elementi in una delle due liste, mentre l'altra conterrà ancora dei dati. Il vostro algoritmo deve essere in grado di riconoscere questa condizione e di continuare ad estrarre valori dall'altra lista finchè anche questa non sia vuota.

c) Questo algoritmo richiede che ci sia spazio di memoria temporanea disponibile sufficiente a permettere la memorizzazione di almeno il doppio del numero totale di elementi da fondere. L'algoritmo si svolge in modo relativamente veloce in quanto ogni elemento viene preso una sola volta.

La fusione è usata generalmente come stratagemma nelle ultime fasi dell'ordinamento, quando la lista da ordinare non può essere memorizzata tutta in una volta nella memoria centrale dell'elaboratore. In tal caso è necessario ricorrere a memorie secondarie come un disco o un nastro magnetico; in questo caso l'ordinamento procede a fasi. Nelle prime fasi vengono portate sequenze di elementi della lista non ordinata nella memoria centrale fino ad esaurire lo spazio. La sequenza ridotta introdotta in questo modo in memoria viene poi ordinata con uno degli algoritmi veloci e "interni" di ordinamento. "Interno" significa in questo caso che l'ordinamento viene fatto all'interno della memoria centrale dell'elaboratore senza ricorrere ad alcuna memoria ausiliaria. Quando una sequenza ridotta è stata ordinata in questo modo, è

poi salvata su disco o nastro, e un'altra sequenza simile è introdotta nella memoria centrale.

```
VIA
  9  15  16  18  25  37  44                  List-A
  3   6   9  17  25  34  45  47  48           List-B

PASSO 1
  3                                          List-M
  9  15  16  18  25  37  44                  List-A
      6   9  17  25  34  45  47  48           List-B

PASSO 2
  3   6                                      List-M
  9  15  16  18  25  37  44                  List-A
          9  17  25  34  45  47  48           List-B

PASSO 3
  3   6   9                                  List-M
     15  16  18  25  37  44                  List-A
          9  17  25  34  45  47  48           List-B

PASSO 4
  3   6   9   9                              List-M
     15  16  18  25  37  44                  List-A
              17  25  34  45  47  48          List-B

PASSO 5
  3   6   9   9  15                          List-M
          16  18  25  37  44                  List-A
              17  25  34  45  47  48          List-B

PASSO 6
  3   6   9   9  15  16                      List-M
              18  25  37  44                  List-A
              17  25  34  45  47  48          List-B

          etc. etc. etc.
```

Figura 14-4

Alla fine tutta la lista iniziale sarà stata separata in sequenza corte e tali sequenze saranno state ordinate internamente e salvate. Nelle successive fasi di ordinamento, le sequenze ridotte saranno fuse a coppie a formare sequenze più lunghe ancora ordinate. La fusione non richiede che tutte le informazioni di ogni sequenza ridotta risiedano in una sola volta nella memoria centrale. Il risultato della fusione è un insieme di sotto-sequenze più lunghe che sono di nuovo salvate su disco o nastro; tali sotto-sequenze sono quindi fuse per produrre altre sotto-sequenze più lunghe. Il processo continua finchè l'intero insieme di dati della lista originaria è stato fuso in una lista finale, che è ora nella corretta sequenza ordinata.

Uno stratagemma simile alla fusione è usato molto comunemente per l'elaborazione di dati amministrativi. In genere un'organizzazione terrà una lista "principale" di informazioni riguardanti le persone su cui si tiene documentazione (ad esempio gli iscritti ad un'università, i clienti di una società di servizio pubblico, gli impiegati di una certa organizzazione,...). Ogni mese si collezionerà un insieme di nuove *"transazioni"* e le elaborazioni consisteranno nell' "aggiornamento" delle registrazioni sul flusso delle persone mediante i dati delle transazioni. La lista principale è tenuta in sequenza ordinata da un mese all'altro; la lista di transazioni verrà ordinata in una delle prime fasi del procedimento. Quando si trovano per uno stesso individuo una registrazione principale ed una registrazione di transazione, la registrazione principale viene corretta, cioè *aggiornata*, tenendo conto delle nuove informazioni di transazione.

Nel caso di registrazioni di grandi magazzini su clienti con carte di credito, una transazione potrebbe essere una nuova vendita o un pagamento da parte del cliente. Dopo aver "fuso" assieme l'informazione contenuta nella vecchia registrazione principale e le informazioni sulla transazione, la registrazione corretta può ora essere salvata su una nuova copia dell'elenco principale. Se non vi sono registrazioni di transazioni in corrispondenza ad una registrazione principale, tale vecchia registrazione viene semplicemente copiata nel nuovo elenco. Se esite qualche registrazione di transazione senza corrispondenza nell'elenco principale allora si è verificato un qualche tipo di errore. Il programma di fusione-aggiornamento dovrebbe quindi fornire la possibilità di visualizzare o stampare un elenco dei casi in cui sono stati trovati risultati particolari. Questo permetterebbe ai responsabili di esaminare questi casi "eccezionali" per correggere dati immessi sbagliati, aggiungere nuove registrazioni principali, e così via.

ESERCIZIO 14.1:

Disegnate un diagramma di struttura che mostri approssimativamente come opera l'ordinamento a bolla. Scrivete e controllate la correttezza di un programma che includa l'ordinamento a bolla come procedura. Dopo aver controllato il programma con i dati usati per illustrare l'algoritmo in figura 14-3, provatelo con una sequenza di dati

più lunga, generata da un numero casuale, come quella usata nei programmi QUATTROLETTERE e CASUALGIRO del Capitolo 5 Sezione 11. Se aveste dei problemi per far funzionare correttamente il programma, aggiungete delle istruzioni di controllo WRITE perchè vi aiutino a trovare i problemi.

ESERCIZIO 14.2:
Scrivete e controllate la correttezza di un programma per fondere due sequenze ordinate di numeri come mostrato in figura 14-4. Iniziate stendendo un diagramma di struttura approssimato che descriva cosa dovrebbe fare l'algoritmo. Controllate il vostro programma con la sequenza di dati usata in Figura 14-4. Dopo aver fatto ciò, controllatelo con due sequenze più lunghe generate da un numero casuale e ordinate con l'ordinamento a bolla. Ogni sequenza dovrebbe essere di circa 100 elementi, ma una dovrebbe essere più lunga dell'altra di almeno 10 elementi. Controllate attentamente la lista ottenuta per essere sicuri che tutti gli elementi siano stati inseriti. In particolare assicuratevi che tutti gli elementi iniziali e finali delle liste abbiano trovato la giusta collocazione nell'elenco finale.

CAPITOLO 15

ORDINAMENTO II
ORDINAMENTO VELOCE

1. Obiettivi

Presentiamo in questo capitolo un importante algoritmo d'ordinamento di media complessità, l'ORDINAMENTO VELOCE ("Quicksort"). Ancora una volta il vostro principale obiettivo dovrebbe essere quello di migliorare la vostra capacità nella soluzione di problemi convertendo la descrizione concettuale di questo algoritmo in un programma funzionante.

2. Premessa

Sebbene i due algoritmi d'ordinamento descritti nel Capitolo 14 abbiano il vantaggio della semplicità, essi sono troppo lenti nel trattare dati ordinati casualmente. L'"ordinamento veloce", descritto in questo capitolo, è uno dei tanti algoritmi che richiedono circa

$$N * \text{Log}_2 N$$

operazioni rispetto alle $\frac{N^2}{2}$

richieste dall'ordinamento per inserzione o da quello a bolla. Ancora una volta vedremo che la politica del "dividi et impera" è pagante come si è visto per la ricerca binaria. Ancora una volta si scopre che la struttura generale degli algoritmi più veloci è una struttura ad albero; di nuovo un accesso recursivo permetterà una migliore esemplificazione dell'algoritmo.

Per farci un'idea del vantaggio in velocità dell'ordinamento Quicksort, consideriamo una sequenza di 1000 elementi da ordinare. Essa è abbastanza piccola rispetto alla lunghezza delle sequenze tipiche che si trovano nei programmi di elaborazioni

amministrative; più grande è la sequenza, maggiore è il vantaggio nell'uso dell'ordinamento Quicksort. Gli algoritmi semplici richiedono circa (N*N/2) operazioni, e cioè circa 500.000; l'ordinamento Quicksort richiede approssimativamente 1000*Log(N) equivalenti a 10.000 operazioni. Gli algoritmi semplici consumano quindi 50 volte il tempo elaborativo del Quicksort per N = 1000! Il Quicksort è stato usato come base per sviluppare parecchi algoritmi avanzati di ordinamento largamente usati.

3. Descrizione del Quicksort

Per farvi un'idea di come operi l'algoritmo Quicksort osservate la figura 15-1 che mostra l'ordinamento della nostra lista di numeri mediante questo algoritmo. L'idea generale dell'algoritmo è quella di suddividere o *ripartire* la lista di dati in due parti, una contenente solo i valori maggiori di un elemento di confronto, l'altra soltanto quelli minori. Ognuna di queste due parti viene poi considerata come una sottosequenza separata da ordinare da sola. L'algoritmo è richiamato recursivamente per ripartire ciascuna di queste due parti separatamente. Questo processo continua fino a quando le Divisioni sono abbastanza piccole da poter essere ordinate con un algoritmo semplice come l'Ordinamento a Bolla.

Facendo riferimento alla Figura 15-1, lo scopo dei passi dall'1 al 5 è quello di ripartire la lista originale in tre parti, una delle quali è costituita dall'elemento di confronto. Per fare questo bisognerò prima stimare quale valore della lista originale sarà circa a metà della lista ordinata quando l'algoritmo finirà. In questo caso abbiamo supposto che il valore 43, originariamente in cima alla lista, finirà circa a metà. Nei passi dall'1 al 5 muoviamo progressivamente i valori più piccoli di 43 nelle locazioni più in alto nella lista (di indice minore) rispetto alla posizione di tale valore. Questo viene fatto nel passo 1 (verso il passo 2), nel passo 3 e nel 5, cioè in tutti i passi in cui la freccia punta verso il basso. Nei passi intermedi, quelli cioè di numero pari, spostiamo i valori maggiori di quello di confronto nelle posizioni inferiori (di indice maggiore) a quella occupata da tale valore. Per evitare confusioni in figura le posizioni vuote sono quelle che non hanno cambiato il loro contenuto rispetto all'ultimo segnato.

Le posizioni evidenziate nella figura dal cancelletto ("#") sono quelle contenenti ad ogni passo il valore di confronto. Le posizioni segnate con un asterisco ('*') sono quelle che ad ogni passo soddisfano le condizioni per uno scambio. Consideriamo per esempio il passo 3 in cui cerchiamo un valore più piccolo di quello di confronto; poichè il passo 2 ha lasciato nella locazione 11 un valore maggiore di quello di confronto, la scansione inizia dalla locazione 10. Il primo valore che si incontra, minore di quello di confronto, è il 23, tale valore è poi scambiato con il valore di confronto per completare il passo 3. Ora la scansione inizia dalla locazione 2 e prosegue verso il

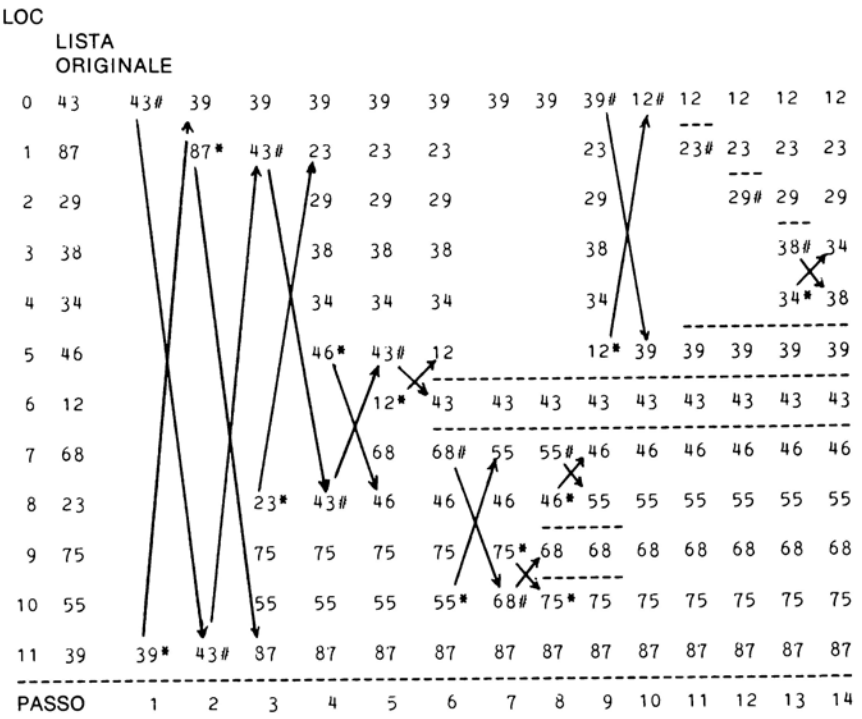


Figura 15-1

basso alla ricerca di un valore maggiore di quello di confronto. Non c'è alcun bisogno di iniziare la scansione alla locazione 0 perchè sappiamo già che le celle 0 e 1 contengono valori minori di quello di confronto. Nel passo 4 il primo valore maggiore di quello di confronto che si incontra è 46 ed esso viene scambiato con il valore di confronto per terminare il passo 4.

Il processo di divisione in liste termina col passo 5 perchè sia la scansione nella parte superiore sia quella nella parte inferiore hanno scambiato le locazioni fino a giungere a quella del valore di confronto. La lista cioè che inizia al passo 6 (completa l'azione del passo 5, come mostrato in figura sopra il contrassegno del passo 6) è ora ripartita in 3 parti, vale a dire:

a) La parte che va dalla locazione 0 alla 5 contenente i valori minori del valore di confronto.

b) La parte contenente il valore stesso di confronto.

c) La parte che va dalla locazione 7 alla 11 contenente i valori maggiori del valore di confronto.

Abbiamo evitato un problema scegliendo per il nostro esempio una lista non contenente valori duplicati. Prima di continuare a leggere potreste pensare a come si può trattare il problema di valori dati doppi se capita di scegliere come valore di confronto un valore che compare più volte nella lista originale.

Dopo aver terminato col passo 5 il primo processo di suddivisione, iniziamo nel passo 6 a ripartire una delle due partizioni create dal primo procedimento. Abbiamo scelto arbitrariamente per il secondo processo di partizione la parte che va dalla locazione 7 alla 11. Questo è fatto nei passi 6 e 7 e produce la lista mostrata vicino al contrassegno del passo 8. Abbiamo usato ancora una volta come valore di confronto quello in cima alla lista di partenza (in questo caso 68). Il risultato è un'altra terna di suddivisioni. La parte dei valori maggiori e quella dei valori minori sono ora abbastanza piccole perchè siano sufficienti due semplici scambi nel passo 8 per terminare il processo per tutti i valori maggiori di 43, valore iniziale di confronto.

Il risultato ottenuto fino ad ora è mostrato accanto al contrassegno del passo 9, dove la parte di lista con i valori minori del valore iniziale di confronto è ancora la stessa della fine del passo 5, cioè quella mostrata sopra il contrassegno del passo 6.

La nostra attenzione si sposta ora al passo 9 sulla parte di lista contenente i valori più piccoli di 43, valore iniziale di confronto. I passi dal 9 fino al 13 mostrano le successive suddivisioni di questa parte della lista. Si confronti questo processo di partizione con quello svoltosi nei passi dal 6 all'8, si vedrà che le cose non sono andate altrettanto bene quanto con la parte contenente i valori più piccoli. La ragione di questo è che il valore di confronto scelto ad ogni passo è stavolta una stima molto scadente del "centro" della lista (ciò che in statistica si chiama "media"). In effetti i passi dal 9 fino al 13 si rivelano una degenerazione dell'algoritmo di ordinamento "per miscelazione" a cui abbiamo brevemente fatto cenno alla fine della discussione dell'ordinamento a bolla nel capitolo 14. A meno di trovare una qualche contromisura, questo problema delle stime cattive può distruggere il vantaggio in velocità degli algoritmi di partizione rispetto all'ordinamento a bolla.

4. Miglioramento di stime scadenti della media

Come regola generale, l'algoritmo di ordinamento veloce non cadrà nella trappola di stime scadenti ripetute purchè la lista originale di dati non sia già quasi ordinata all'inizio. Nei casi in cui si possa essere sicuri che una lunga lista di dati sia in ordine pressochè casuale prima dell'ordinamento, non sarà necessario adottare alcuna

contromisura nell'ordinamento veloce per tener conto di questo problema. Da quando è stato scoperto questo problema di stime cattive, molti autori hanno pubblicato suggerimenti per fare stime migliori della media, o valore mediano. Il metodo più semplice sembra essere il seguente:

> Ad ogni passo in cui si debba iniziare un nuovo processo di suddivisione, identificate tre elementi da usare come stima. Usate il primo e l'ultimo elemento della lista e anche l'elemento allocato fisicamente più vicino al centro della lista originaria. Se i dati sono in numero pari, scegliete quello di indice minore dei due elementi "centrali". Nell'esempio mostrato in Figura 15-1, i tre valori sarebbero:
>
> PRIMO = 43, MEZZO = 46, ULTIMO = 39.
>
> Ordiniamo ora questi tre valori per identificare quello che non è nè il maggiore nè il minore. Questo valore intermedio dovrebbe essere usato come "stima" per la prossima fase di suddivisione. Se necessario, il valore intermedio potrebbe essere scambiato con il primo valore per porre il corretto valore stimato all'inizio della partizione prima dell'inizio dell'elaborazione.

La figura 15-2 mostra come questo raffinamento nell'algoritmo opererebbe a partire dal passo 9. Notate che la stima, fatta nel primo processo di suddivisione sulla lista iniziale completa, era la stessa che sarebbe stata fatta con questo raffinamento.

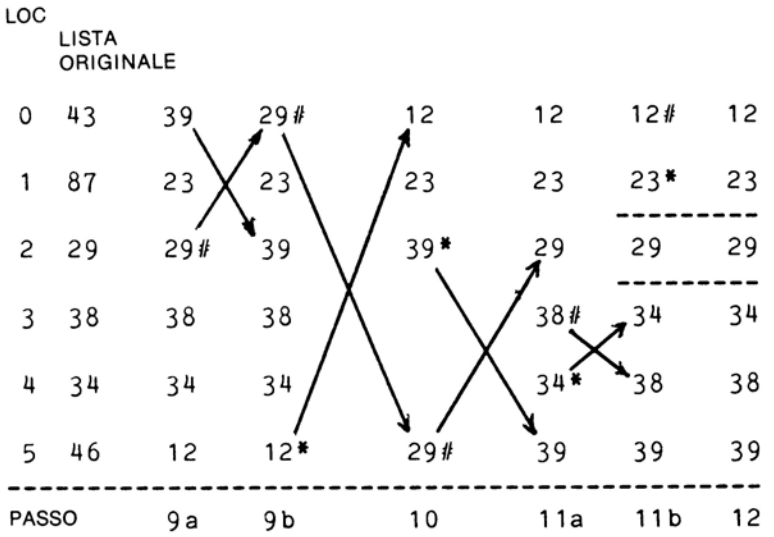


Figura 15-2

Nella figura 15-2 il passo 9 è stato spezzato per mostrare la scelta della stima migliorata del valore mediano. Anche il passo 11 è spezzato per lo stesso motivo. Comunque la partizione considerata nel passo 11 contiene solo tre valori; non c'è quindi alcun motivo per continuare il processo di suddivisione dopo l'ordinamento di tali tre valori come preliminare al prossimo stadio.

Non aspettatevi che l'ordinamento veloce (in entrambe le forme) costi molto meno tempo elaborativo dell'ordinamento a bolla su liste di valori corte come quella qui usata. Se usate una lista di 100 valori ordinati casualmente, l'ordinamento a bolla richiederà circa 5000 confronti di coppie di elementi, mentre l'ordinamento veloce ne richiederà soltanto 700 circa. Il vantaggio dell'ordinamento veloce sull'ordinamento a bolla diventerebbe maggiore con liste più lunghe di questa.

Prima di procedere a considerare il diagramma di struttura di questo algoritmo, eliminiamo il problema creato dall'avere più di una volta nella lista da ordinare il valore stimato. Nell'esempio qui citato, cosa avreste fatto se il valore 43 fosse stato presente più di una volta? Nel primo stadio di suddivisione, dal passo 1 al 5, la migliore soluzione del problema sarebbe stata quella di permettere agli altri <43> di cadere sia nella partizione alta, sia in quella bassa. Fare questo significa semplicemente rimandare la decisione su dove mettere queste ulteriori presenze del valore stimato. Come potete facilmente verificare, il prossimo stadio della partizione avrà l'effetto di muovere questi ulteriori <43> a posizioni immediatamente adiacenti alla posizione finale del valore stimato.

5. Diagramma di struttura recursivo

Iniziamo ora a stendere il diagramma di struttura per l'algoritmo. Chiaramente è abbastanza complicato perchè valga la pena di usare uno o più sotto-algoritmi. Un'ovvia applicazione di un sotto-algoritmo sarà il processo di evoluzione di uno stadio di partizione. Un sotto-algoritmo DIVISIONE(PRIMO,ULTIMO) sarà richiamato una volta per seguire i passi dall'1 al 5 nella figura 15-1, una volta per i passi 6 e 7 e una volta per i passi da 9 a 11 (rivisto). Il sotto-algoritmo dovrebbe essere in grado di far fronte al problema di piccole partizioni contenenti solo due o tre elementi.

In realtà il sotto-algoritmo deve essere scritto in modo da indicare se sono necessarie fasi addizionali di suddivisione o se in una particolare zona della lista è stata raggiunta la "fine linea". Questo suggerisce che una definizione recursiva dell'algoritmo potrebbe essere la più facile da capire per una prima prova. La figura 15-3 illustra una tale definizione recursiva; si studi attentamente questo algoritmo e si verifichi che le azioni intraprese sono le stesse illustrate in figura 15-1. Per una prima prova ignorate l'operazione del blocco 1.a (richiama l'ordinamento a bolla) e della 1.b.2 (richiama il sotto-algoritmo CONGETT). ULTIMO è il valore indice (numero di loca-

zione) maggiore nella partizione da ordinare; PRIMO è il valore indice minore nella stessa partizione. Nell'esempio di Figura 15-1, DIVISIONE inizia al passo 1 con PRIMO=0 e ULTIMO=11.

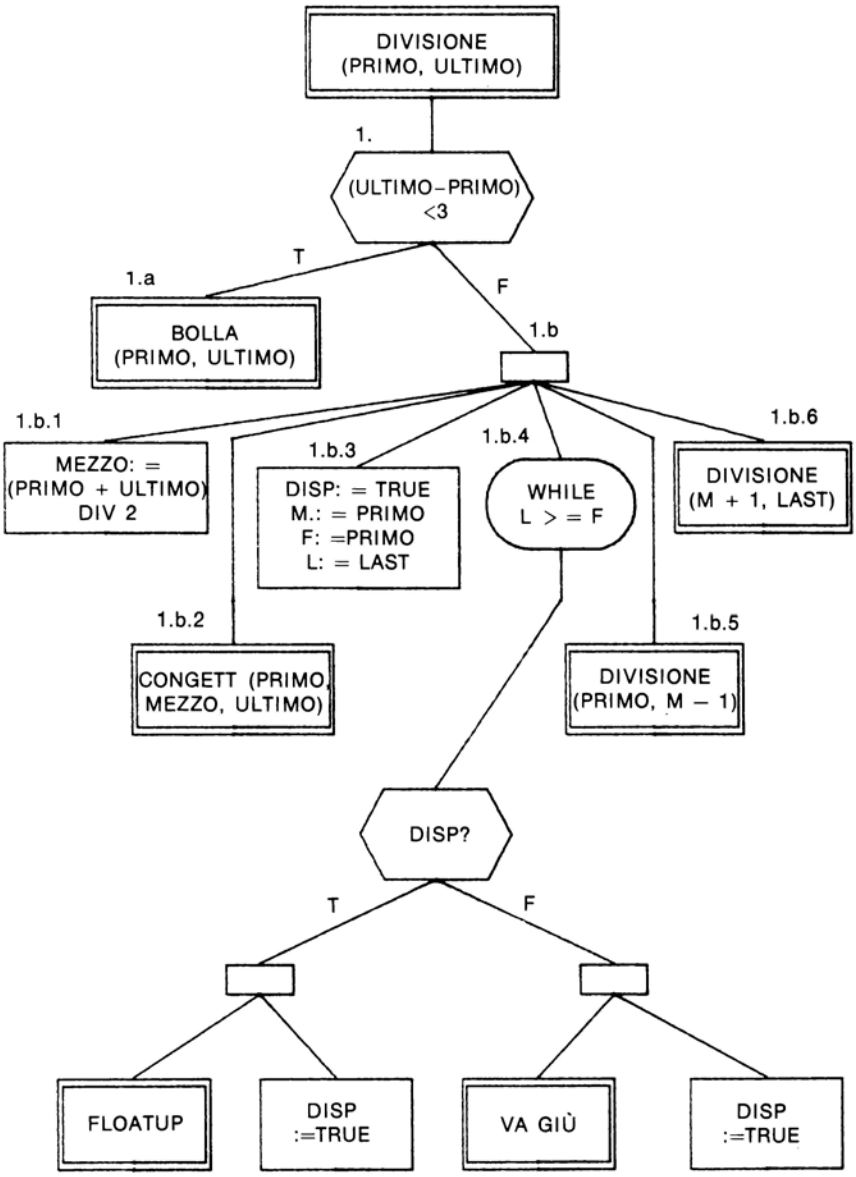


Figura 15-3

Notate i seguenti punti riguardo a questo algoritmo:

a) I sotto-algoritmi FLOATSU e VAGIÚ fanno la maggior parte del lavoro del sotto-algoritmo DIVISIONE. FLOATSU si applica ai passi di numero dispari, 1, 3, 5 ...; VAGIÚ opera nei passi di numero pari. Questi due sotto-algoritmi sono talmente semplici da non poter essere implementati come procedure in un programma.

b) M punta al valore stimato, F al valore più vicino a PRIMO che sarà il prossimo da confrontare col valore stimato. Allo stesso modo L punta al valore più vicino a ULTIMO che sarà il prossimo da confrontare col valore stimato ed eventualmente scambiato.

c) Una singola fase di partizione (come quella che nell'esempio va dai passi 1 al 5) termina quando L non è più maggiore di F. In realtà quando finisce FLOATSU, L dovrebbe essere uguale a M e quando termina VAGIÚ, F dovrebbe essere uguale a M.

d) In generale il risultato di una fase di suddivisione sarà lasciare due nuove partizioni che dovranno essere a loro volta ordinate. Quella contenente i valori minimi sarà ordinata richiamando, nel blocco 1.b.5, DIVISIONE(PRIMO, M−1); l'altra è ordinata richiamando DIVISIONE(M+1, ULTIMO) nel blocco 1.b.6

e) Come sotto-algoritmo BOLLA viene preso quello visto nel Capitolo 14. È richiamato soltanto quando la partizione contiene 3 o meno elementi, come mostrato in figura 15-3. A causa degli ulteriori calcoli necessari per fare stime dei valori, nel caso di liste lunghe l'intero algoritmo di ordinamento funzionerebbe probabilmente molto meglio se il sotto-algoritmo BOLLA fosse richiamato per partizioni contenenti dieci o meno elementi piuttosto dei 3 come mostrato.

f) L'intero procedimento di ordinamento veloce è ottenuto richiamando questa forma recursiva di DIVISIONE dove PRIMO è l'indice minore e ULTIMO è l'indice maggiore della lista da ordinare. Si suppone che il vettore LST contenga i dati iniziali quando DIVISIONE è richiamata la prima volta.

ESERCIZIO 15.1:
Scrivete il programma per l'algoritmo di ordinamento veloce e lo si controlli, provando prima con la sequenza di dati usata in figura 15-1, e quindi con una lista di almeno 100 elementi generata da un generatore di numeri casuali.

Questo programma è abbastanza complesso perchè si rischi di spre-

care molto tempo se non si usa lo sviluppo di programma passo-passo e il procedimento di controllo che abbiamo raccomandato. Lavorando con la sequenza corta di dati usata in questo libro, dovreste fare in modo di dare visualizzazioni di controllo che mostrino ad ogni passo il risultato dell'ordinamento. Controllate ogni passo e assicuratevi che i risultati siano gli stessi dati nelle Figure 15-1 e 15-2.

Per avere tutti i dati di un passo visualizzati in una sola volta, conviene stampare i dati secondo una linea orizzontale piuttosto che in una verticale come in figura. Le vostre stampe di controllo dovrebbero probabilmente includere anche una seconda riga per ogni passo, in cui dare i valori delle variabili rilevanti come F, L, M, DISP e così via. Se possibile dovreste effettuare i controlli con una copia stampata del vostro programma per poter fare riferimento alle righe successive stampate dalle istruzioni di controllo. Come più volte ripetuto, vi suggeriamo di porre delle istruzioni READLN dopo un gruppo di istruzioni di controllo WRITE in modo da permettervi di studiare i risultati di ogni passo prima di continuare col successivo (battendo <RET>).

APPENDICE A

DIFFERENZE FRA PASCAL UCSD E PASCAL STANDARD

Il linguaggio PASCAL usato in questo libro contiene molte delle caratteristiche descritte da K. Jensen e N. Wirth in "Pascal-Manuale e standard del linguaggio" (Jackson 1980).

Anche se non sono stati ancora stabiliti degli standard internazionali, a causa della sua diffusa accettazione, noi faremo riferimento al PASCAL definito da Jensen Wirth come PASCAL "STANDARD". Il Pascal usato in questo libro è stato implementato all'università di S. Diego in California (UCSD) all'interno di un sistema software completo da usarsi su un certo numero di microelaboratori. Questo linguaggio – PASCAL UCSD – differisce dallo standard per alcune omissioni, un numero molto piccolo di cambiamenti, e parecchie estensioni. Questa appendice dà un breve e sintetico sommario di queste differenze; saranno menzionati solo i costruttori usati in questo libro. Presso l'università di S. Diego sono disponibili dei documenti che descrivono il PASCAL UCSD in dettaglio.

1. Istruzione CASE
   Jensen e Wirth stabiliscono che se non ci sono etichette uguali al valore del selettore, allora il risultato dell'istruzione case è indefinito. PASCAL UCSD, in questa situazione, esce normalmente dall'istruzione senza aver fatto niente.

2. Commenti
   Nel PASCAL UCSD un commento è racchiuso tra i simboli "(*" e "*)". Se il simbolo delimitatore iniziale è immediatamente seguito dal segno di dollaro, come in "(*$", allora il resto del commento è trattato con una direttiva al compilatore. L'unica direttiva al compilatore trattata in questo libro è (*$G+*), che informa il compilatore che si desidera usare l'istruzione GOTO. Il compilatore UCSD non gestisce correttamente dei commenti nidificati.

3. EOF(F)
   Per porre TRUE EOF per un flusso-testo F, incluso il flusso standard INPUT, l'utente deve premere il tasto <ETX> o <ENTER> (CONTROL–C per le tastiere

che non hanno un tasto esplicito per questo scopo). Il codice specifico usato per questo scopo può essere cambiato da <ETX>, se voluto.
Se il flusso F è chiuso, EOF ritorna TRUE. Se EOF è TRUE, od il flusso è di <tipo> TEXT, anche EOLN è vero per lo stesso flusso. A seguito di RESET(F), EOF ritorna FALSE se il flusso esiste. Il sistema, automaticamente, esegue un RESET sui flussi INPUT, OUTPUT, e KEYBOARD quando il programma è inizializzato.

4. EOLN(F)
   EOLN(F) è definito solo se F è un flusso di <tipo>TEXT. EOLN diventa TRUE solo a seguito di un READ durante il quale è ricevuto il carattere di fine linea (per difetto <RET>), e prima del prossimo READ.

5. Istruzioni GOTO e EXIT(P)
   PASCAL UCSD permette un GOTO solo dentro lo stesso <blocco> che contiene la dichiarazione dell'etichetta a cui rimanda il GOTO. L'EXIT dà delle possibilità equivalenti al GOTO ma con l'etichetta a cui si rimanda sull'istruzione che segue immediatamente il punto in cui la procedura P è stata chiamata l'ultima volta. Si veda l'appendice C per ulteriori discussioni dell'EXIT. Quando il sistema è inizializzato per gli studenti l'istruzione GOTO è disabilitata; può essere abilitata usando la direttiva (*$G+*).

6. Variabili impaccate
   PASCAL UCSD permette l'impaccamento su vettori di caratteri e sui <tipi> record. Su una parola di 16 bit sono impaccati 2 caratteri. All'interno di un record, l'impaccamento od il disimpaccamento sono fatti automaticamente dentro gruppi di campi che sono di almeno 16 bits. PASCAL UCSD non ha l'equivalente delle procedure interne PACK e UNPACK descritte da Jensen e Wirth.

7. Procedure e funzioni come parametri formali
   PASCAL UCSD non permette l'uso come parametri di identificatori di procedure o funzioni. Unica eccezione è la procedura interna EXIT.

8. Intestazioni dei programmi
   Nei primi rilasci di PASCAL UCSD, l'elenco dei nomi dei flussi standard non può essere incluso come pseudo parametro nella linea PROGRAM. Successivi rilasci includeranno questo meccanismo come opzione da usarsi nel comando X (ecute) del sistema operativo.

9. READ e READLN
   Jensen e Wirth definiscono READ(F, CH) come equivalente alla sequenza:

```
CH := F^;
GET(F);
```

In PASCAL UCSD questa sequenza è *invertita* quando F è uno dei flussi di ingresso standard: INPUT o KEYBOARD. Senza questo cambiamento sarebbe estremamente scomodo scrivere programmi che interagiscano strettamente con l'utente.

10. RESET(F)
    In PASCAL UCSD, RESET(F) porta la finestra del flusso all'inizio ma non carica la variabile finestra F^. L'equivalente UCSD dello standard è perciò:

```
RESET(F);
GET(F);
```

11. REWRITE(F)
    PASCAL UCSD non permette la procedura standard REWRITE. Prestazioni simili sono date da OPENNEW.

12. Sets
    In PASCAL UCSD un set può avere al massimo 255*16 elementi: può cioè avere le dimensioni di 255 parole da 16 bit. Sono permesse tutte le operazioni descritte da Jensen e Wirth.

13. Variabili STRING
    PASCAL UCSD ha il <tipo> STRING già dichiarato. In sostanza una variabile di <tipo> STRING è un PACKED ARRAY[1..80] OF CHAR, con associato un attributo che specifica la lunghezza di <tipo> INTEGER. La lunghezza di 80 assunta per difetto può essere cambiata con una dichiarazione che, dentro parentesi quadre, specifica la massima lunghezza desiderata. Esempio:

```
TITLE: STRING [30];
```

    La massima lunghezza permessa è 255. In questo libro c'è un'ampia descrizione delle variabili STRING.

14. WRITE e WRITELN
    PASCAL UCSD differisce dallo standard in quanto non permette parametri di <tipo> BOOLEAN per le istruzioni WRITE e WRITELN. I parametri di <tipo> STRING danno i risultati descritti nel libro.

15. Grafici tartaruga
    Le possibilità grafiche descritte in questo libro e permesse in PASCAL UCSD non hanno un equivalente nel linguaggio standard.

APPENDICE B

TERMINI DEL GERGO DELL'INFORMATICA

Questo glossario è costituito da un elemento di parole usate in questo libro con un significato diverso dal linguaggio quotidiano; ogni spiegazione è un conciso promemoria del significato del termine. Per un maggior chiarimento rivedete il (<capitolo>. <sezione>) associato ad ogni termine. Per esempio, (II.5) fa riferimento al capitolo 11 sezione 5. Tra parentesi quadre [ ] è rappresentata la traduzione inglese.

Aggiornare (14.5) [Update].
Si dice che si aggiorna un record in un flusso esistente quando il suo contenuto è sostituito con valori più recenti.

Albero (4.10) [Tree].
Una struttura logica ramificata avente uno ed un solo percorso dalla radice ad un qualsiasi nodo foglia.

Algoritmo (0.2) [Algorithm].
Un'affermazione che descrive una sequenza di azioni necessarie per portar a termine uno specifico compito.

Alto livello (5.3) [Highen level].
L'hardware dell'elaboratore può solitamente eseguire solo semplici operazioni primitive. Una operazione di alto livello è una sequenza di queste operazioni primitive progettata per eseguire una operazione più complessa. Un linguaggio di programmazione di alto livello è uno progettato per richiamare operazione di livello più alto in termini facilmente comprensibili.

Argomento (4.9) [Argument].
Un dato da usarsi come parametro effettivo.

Arrotondare (5.7) [Round].
Arrotondare un numero in virgola mobile significa convertirlo nel valore più vicino che può essere espresso con il numero limitato di bits di quel particolare

elaboratore. La funzione interna ROUND converte il valore di una <variabile> reale nel valore intero più vicino.

ASCII (5.2) [ASCII].
Il codice nazionale Americano standard per lo scambio di informazioni. Un'assegnazione arbitraria di valori numerici a 96 caratteri visualizzabili, e 32 caratteri di controllo.

Assegnare (2.7) [Assign], Operatore assegnamento (2.8) [Assignment operator], Istruzione assegnamento (2.7) [Assignment statement].
L'assegnamento di un valore ad una variabile fa si che questo valore sostituisca qualsiasi valore precedentemente memorizzato nella locazione individuata dalla variabile. L'operatore assegnamento (":=") è il simbolo PASCAL che identifica un'istruzione di assegnamento che fornisce alla variabile alla sinistra dell'operatore un nuovo valore.

Asserzione (6.5) [Assertion].
L'affermazione di un fatto la cui falsità o verità sarà verificata in un algoritmo.

Attivare (11.5) [Activate].
Si dice che una procedura o funzione è attivata quando è chiamata ed inizia l'esecuzione. L'attivazione finisce quando l'esecuzione termina normalmente o a causa di un'istruzione EXIT. Una procedura o funzione recursiva può attivare parecchie "copie" di sé stessa simultaneamente.

Baco (1.10) [Bug].
Un errore di logica in un programma che fa si che il programma si blocchi in modo non abnorme o dia risultati non corretti.

Bandiera (7.11) [FLAG].
In questo libro è usato nello stesso senso di carattere di fuga. Solitamente un valore non verosimile o impossibile usato per segnalare al programma di eseguire una ben determinata azione.

Bit (5.2) [Bit].
Una cifra binaria il cui valore può essere 0 o 1.

Mappatura a bit (12.6) [Bit Map].
Termine usato per descrivere un video grafico che crea le immagini fornendo un elevato numero di posizioni in cui un programma può rendere luminoso o meno un punto.

Bit di ordine più alto (5.6) [High order bit].
Il bit nella parola di un elaboratore che ha il peso più alto. È solitamente il bit

più a sinistra nella rappresentazione dei bits di una parola: proprio come la cifra più significativa di un numero.

Campo (5.6), (10.3) [Field].
Un campo è un gruppo contiguo di bits o caratteri all'interno di un record dove l'intero gruppo rappresenta la memorizzazione di una variabile. Un numero a virgola mobile (floating Point) è composto da diversi campi contenenti la mantissa e l'esponente.

Carattere di fuga (7.11) [Escape character].
Un carattere di fuga è un carattere usato per segnalare ad un programma che deve essere eseguita una certa azione. Solitamente un carattere di fuga è inserito in un contesto nel quale lo stesso carattere non potrebbe mai comparire come un dato normale.

Caratteri di controllo (3.5) [Control characters].
Caratteri non stampabili usati per controllare dispositivi di output come videi e stampanti. Caratteri di controllo sono, ad esempio, interlinea, ritorno e cancellazione.

Chiamata (2.3) [Call].
Il processo per cui una parte di un programma blocca temporaneamente la propria esecuzione e manda in esecuzione una procedura.

Chiave (13.6) [Key].
In un'operazione di ricerca, la chiave è l'elemento che permette di individuare dei dati in un elenco o in un flusso. Nell'ordinamento, la chiave è una parte del record che deve essere usato a scopo di confronto per determinare l'ordine dell'elenco risultante.

Ciclo (3.3) [Loop].
Termine che ha avuto origine quando l'esecuzione ripetuta di una serie di passi di programma richiedeva l'uso dell'istruzione GOTO. Questa ha portato a rappresentazioni, su schemi a blocchi, di richiusure indietro su se stessi.

Ciclo (5.4) [Cycle].
È usato in questo libro per indicare una fase operativa della CPU di un elaboratore.

Codice (5.2) [Code].
Un valore numerico o una sequenza di bits usata per identificare in modo non ambiguo un valore non numerico. Il termine "codice" è anche usato per far riferimento all'intera sequenza di istruzioni in linguaggio macchina.

Colonna (8.8) [Column].

Un raggruppamento verticale di elementi in un vettore a tabella. Può anche far riferimento alla posizione di un carattere in una linea o riga di testo. I due significati sono in effetti equivalenti in quanto i testi sono memorizzati come vettori.

Comando (2.1) [Command].

Una direttiva al sistema o programma per chiedergli di eseguire una certa azione o per lanciare uno specifico programma.

Commento (1.4) [Comment].

Una parte di un programma PASCAL che è ignorata dal compilatore e che serve per una maggior comprensione da parte di un lettore umano.

Compilatore (0.9) [Compiler].

Un programma che traduce programmi scritti in PASCAL (o qualche altro linguaggio di programmazione) in una forma che può essere interpretata direttamente dall'hardware di un elaboratore come una sequenza di operazioni elementari - istruzioni.

Comunicazione (4.5) [Communication].

Termine usato in questo libro per indicare il modo in cui l'informazione è passata da una parte di un programma ad un'altra per mezzo di parametri o variabili globali.

Concatenare (2.11) [Concatenate].

Una <stringa> è concatenata ad un'altra quando la prima <stringa> è appesa alla fine della seconda. Per esempio, concatenando "grand" a "uccello" si ottiene "granduccello".

Configurazione (2.11) [Pattern].

Termine usato per descrivere una <stringa> che sarà usata per scandire una seconda <stringa> allo scopo di determinare se la prima <stringa> è contenuta nella seconda.

CONST, Costante (10.5) [Constant].

Parola riservata in PASCAL usata per introdurre una sequenza di dichiarazioni di costanti. Queste sequenze associano uno specifico identificatore con dei valori fissi da usarsi nel programma.

Convalidare (7.11) [Validate].

La convalida è l'azione di verificare i valori di dati immessi per vedere se quei valori sono dentro limiti accettabili e perciò probabilmente non errati.

Copia su carta (0.5) [Hard-copy].
Il termine inglese deriva dal contrasto con copie di registrazioni emesse dall'elaboratore su uno schermo che sono dette "soft".

Costante intera (2.5) [Integer Constant].
Un numero intero positivo o negativo. Esempi: -34, 109, 1, 3, 0, -102.

Costruttore (9.6) [Constructor].
Questo termine è usato a proposito di un elenco di valori costanti racchiusi tra parentesi quadre usati per costruire un nuovo valore da assegnarsi a un insieme.

CPU (5.3) [CPU].
Si veda "Unità centrale di elaborazione".

Cursore (1.5) [Cursor].
Un segno che indica all'utente la sua posizione relativa in un testo visualizzato.

Dati (0.2) [Data].
Informazioni che devono essere fornite ad un programma, il cui compito consiste solitamente nel variare i dati stessi, riassumerli o usarli per prendere decisioni.

Debug (1.10) [Debug].
Il debug è l'operazione di scoperta ed eliminazione degli errori logici (bachi) dal programma.

Delimitare, Delimitatore (1.11) [Delimit, Delimitor].
Delimitare significa segnare l'inizio e la fine di entità come stringhe 5 commenti. Un delimitatore è un simbolo usato come segno per questo scopo.

Diagramma finestra (4.3) [Window diagram].
Un diagramma usato per illustrare l'ambiente dentro il quale gli identificatori dichiarati possono essere usati in un programma PASCAL.

Dichiarare, Dichiarazione (2.3) [Declare, Declaration].
Tutte le variabili definite dall'utente in un programma PASCAL devono essere dichiarate, rispettando le regole sintattiche prescritte, per informare il compilatore sulle caratteristiche delle variabili stesse.

Differenza (9.6) [Difference].
Il termine "differenza" in PASCAL non si applica solo alla ben conosciuta operazione aritmetica ma anche agli insiemi. Data l'istruzione di assegnamento S3

:= S1 - S2, la differenza di S2 da S1 è assegnata a S3: cioè, tutti gli elementi dell'insieme S1 che non sono anche elementi di S2 sono inclusi in S3.

Digitale (5.2) [Digital].
Un elaboratore digitale è un elaboratore che esegue tutte le operazioni lavorando su cifre: solitamente le cifre binarie 1 e 0.

Disco morbido (1.2) [Floppy disk].
Disco magnetico flessibile usato per memorizzare dati su piccoli elaboratori.

Dispositivo periferico (5.3) [Peripheral davice].
Un dispositivo che può essere aggiunto alla struttura base dell'elaboratore. Esempi di dispositivi considerati periferici includono stampanti, unità disco, lettori di schede e terminali interattivi.

DIV (5.7)
Operatore per la divisione intera. Richiede degli operandi interi, e dà come risultato un quoziente intero.

Documento (0.9) [Document].
Per la gente che lavora nel campo dell'informatica un documento è molto spesso una descrizione scritta di un elaboratore o di un programma con le istruzioni per l'uso.

Eco (7.2) [Echo].
Il termine eco implica che ogni carattere battuto su tastiera appaia immediatamente sullo schermo o sulla stampante collegata all'elaboratore.

Editare (1.5) [Edit].
Editare un programma significa variarne il contenuto. Per "Editor" si intende il programma che supporta nell'editare altri programmi.

Effetti collaterali (4.2) [Side effect].
Un cambiamento non previsto del valore di una variabile fatto da una procedura (nel quale lo stesso identificatore avrebbe dovuto essere dichiarato locale) che porta a comportamenti non corretti di altre parti del programma.

Elaboratore (0.2) [Process, Processing].
Un elaboratore elabora (dei dati) tramite un programma eseguendone le istruzioni per piccoli passi. Quando un programma, o una parte di un programma, sta elaborando si dice anche che è in "esecuzione" o che "gira".

Elaboratore analogico (5.2) [Analog computer].
Termine applicato ad una classe di macchine che usa segnali elettrici, leve

meccaniche, liquidi od altri mezzi del genere per simulare il comportamento di altri sistemi.

Elaboratore conversazionale (7.2) [Conversational computer].
Un elaboratore preparato per interagire con l'utente visualizzando o stampando messaggi e ricevendo messaggi da una tastiera.

Elemento (8.2) [Element].
Il termine elemento indica un singolo dato di un insieme o di un vettore.

Elenco parametri (2.8) [Parameter list].
Elenco dei parametri che devono essere usati con una procedura o funzione. È posto nella intestazione della dichiarazione della procedura o funzione come sequenza di identificatori con il <tipo> associato.

Esecuzione (in) (13.3) [Run-Time].
Descrive il periodo di tempo in cui un programma sta effettivamente girando sull'elaboratore. Quando l'esecuzione di un programma termina in modo anomalo, si dice che ha trovato un errore in esecuzione (run-time error).

Eseguire (1.4) [Execute].
Si dice che un programma o una parte di programma è eseguita quando esegue le azioni o passi logici descritti dal programma stesso.

Esercizio (0.2) [Exercise].
In questo libro un esercizio è un problema che deve essere impostato dal lettore affinché la soluzione possa essere trovata su un elaboratore.

Esponente (5.6) [Exponent].
La parte di un numero in virgola mobile che designa la potenza a cui deve essere elevato 2 o 8.

Espressione aritmetica (2.9) [Arithmetic expression].
Gli strumenti di PASCAL che, combinando operandi con valori numerici, permettono di eseguire le operazioni aritmetiche di addizione, sottrazione, moltiplicazione e/o divisione.

Etichetta (11.3) [Label].
Un segnalatore che designa una istruzione in un programma alla quale si può saltare con l'istruzione GOTO.

Fattore di scala (12.5) [Scale factor].
Una costante usata per aggiustare la dimensione di un grafico per fargli riem-

pire lo schermo o la pagina stampata, senza tuttavia perderne una parte fuori dai limiti dello schermo o della pagina.

Filate (5.4) [Hard wired].
Fa riferimento al metodo di connettere parti dell'hardware interno della macchina usando dei fili. La maggior parte degli elaboratori costruiti attualmente usano poche connessioni filate; sfruttano il più possibile delle connessioni logiche basate su memorie a sola lettura che sono più facilmente cambiabili.

Fine flusso (7.2) [End of File (EOF)].
La condizione di fine flusso è raggiunta quando un programma legge da un flusso tutti i dati disponibili; dove il termine "flusso" si applica a qualsiasi dispositivo esterno.

Flusso (1.5), (7.2) [File].
Il termine si applica spesso ad una raccolta di dati memorizzati su disco, nastro, schede o su un qualche altro supporto che può essere tolto dall'elaboratore. Il termine è comunque spesso usato in senso stretto per far riferimento al modo logico con il quale dei dati possono essere scambiati tra CPU ed un dispositivo esterno sotto il controllo di un programma.

Flusso KEYBOARD (7.7) [KEYBOARD file].
Nel sistema software PASCAL UCSD, KEYBOARD è il nome del flusso, senza eco, equivalente allo standard INPUT.

Flusso di lavoro (1.7) [Workfile].
Una copia temporanea di lavoro del programma in corso di variazione con l'editore. Il compilatore prende le istruzioni PASCAL dal flusso di lavoro quando sono usati i comandi R (un) e C (ompile).

Foglia (6.5) [Leaf node].
Un blocco in un diagramma di struttura che non ha rami a livello più basso. Si applica anche a blocchi simili in qualsiasi schema costruito come un albero logico.

Forma Interna (7.2) [Internal Form].
All'interno dell'elaboratore i numeri sono memorizzati con una notazione interna binaria. In emissione, questa forma è solitamente convertita in caratteri. In immissione ha luogo la conversione opposta.

Formato (7.2) [Format].
Nel gergo dell'informatica un formato è una specifica scritta in un linguaggio speciale che descrive la conversione di caratteri in forma interna per l'immis-

sione, e l'opposto in emissione. In PASCAL il formato necessario per transcodificare un numero intero o reale in binario, o viceversa, è inferito dal <tipo> della variabile o dell'espressione che compaiono nelle istruzioni di WRITE o READ.

Forward (4.3)
Una parola riservata uscita nelle dichiarazioni di procedura per permettere che altre procedure possano farvi riferimento prima che il "corpo" principale della procedura compaia nel programma. Spesso usata per rendere possibile che due procedure si chiamino vicendevolmente.

Funzione (2.11) [Function].
Una procedura che ridà un valore quando ha terminato l'elaborazione. Una funzione è chiamata includendo l'identificatore di funzione in un'<espressione>.

Fusione (14.2) [Merge].
Il processo per cui due (o più) liste ordinate di dati sono combinate in un'unica lista ordinata.

Hardware (1.2).
La "macchina elaboratore" che può essere vista e toccata. Per esempio la CPU, un terminale interattivo, un disco, la stampante o la tastiera sono tutte parti dell'hardware.

Identificatore (1.8) [Identifier].
Il nome dato ad un'entità in un programma. Il nome dovrebbe, preferibilmente, aiutare a ricordare lo scopo dell'entità.

Immagine di scheda (7.2) [Card image].
Un record che contiene le stesse informazioni perforate su una scheda.

Immissione/Emissione (5.3) [Input/Output].
Il processo di trasferire dati a/dà la memoria principale da/a qualche dispositivo esterno.

Impaccato (8.7) [Packed].
Un vettore o un record impaccati sono strutture che occupano il minor spazio di memoria possibile pur usando abbastanza bits per memorizzare tutti i possibili valori del <tipo> associato.

Indice (8.4) [Subscript].
Valore usato per accedere ad un elemento di un vettore.

Indefinito [Undefined].

Il valore di una variabile è detto indefinito se non è ancora stato assegnato alcun valore alla variabile stessa; oppure dopo la conclusione di certe operazioni.

Indentazione (3.11) [Indent, Indentation].

Una linea del testo di un programma è indentata inserendo spazi bianchi nonfunzionali a sinistra. Facendo questo con accortezza, si può creare una prima approssimazione ad uno schema che descrive la struttura di un programma in PASCAL.

Indice non valido (9.5) [Invalid Index].

Termine che descrive una fine anomala di un programma che avviene quando si tenta di: a) far riferimento ad una locazione di un vettore che non è stata dichiarata essere entro i limiti del vettore; b) assegnare un valore ad una variabile sottocampo oltre i limiti specificati nella dichiarazione del < tipo >.

Indirizzo (5.4) [Address].

L'indirizzo di una parola nella memoria di un elaboratore può essere pensato come il numero del registro contenente la parola.

Inizializzare (3.7) [Initialize].

Si dice che una variabile è "inizializzata" quando le è assegnato un valore iniziale.

Interno (2.11) [Built-in].

Aggettivo da applicarsi a procedure o funzioni già dichiarate che sono fornite come parte del software di sistema.

Interattivo (7.2) [Interactive].

Un sistema di elaborazione interattivo è uno che può "conversare" con l'utente mandandogli dei messaggi via schermo o stampante, e ricevendo messaggi da tastiera.

Interprete (3.8) [Interpreter].

Un programma che gira su un elaboratore realmente "ospite" nel senso che fa apparire l'elaboratore come se avesse una logica diversa.

Intersezione (9.6) [Intersection].

L'intersezione di due insiemi è un insieme i cui membri includono elementi che sono membri di *entrambi* gli insiemi originali.

Intervallo (9.5) [Range].

Si riferisce solitamente ad un insieme di valori che una variabile può assumere. Si dice che l'intervallo va da un limite inferiore (il valore più piccolo possibile) ad un limite superiore (il valore più alto possibile).

Intestazione - Testata (2.6) [Heading].

Termini usati per far riferimento a quella parte delle dichiarazioni di procedura o funzioni che vengono prima della prima dichiarazione, CONST, TYPE e VAR nello stesso <blocco>.

Inverso [Inverse].

L'inverso di qualcosa è il suo opposto. L'inverso di TRUE è FALSE (cioè NOT TRUE). Se I e J sono interi o reali l'inverso di I >J è L >=J.

Istruzione composta (3.2) [Compound statement].

Un gruppo di istruzioni racchiuse tra le parole riservate BEGIN e END. Un comodo metodo per far si che le istruzioni dentro il gruppo siano controllate assieme.

Istruzione di selezione (4.10) [Case statement].

L'istruzione di selezione permette di scegliere l'esecuzione di una tra più istruzioni. Questo diversamente dall'istruzione IF che permette di operare una scelta su due istruzioni al massimo.

Istogramma (12.6) [Histogram].

Un grafico a "barre" in cui ogni barra (solitamente verticale) rappresenta il valore di un singolo dato.

Istruzione GO TO (11.2) [GO TO statement].

Un'istruzione che si trova nella maggior parte dei linguaggi di programmazione e che fa si che il controllo passi, bruscamente, in un altro punto del programma.

Linea di suggerimento (1.2) [Prompt line].

In certi sistemi come PASCAL UCSD, è solitamente visualizzato un messaggio di suggerimento su una determinata linea dello schermo.

Linguaggio Assemblatore (5.3) [Assembler Language].

Un metodo per programmare un elaboratore simile all'uso del linguaggio macchina; la differenza sta nel fatto che l'utente fa riferimento ad ogni operazione usando un identificatore mnemonico piuttosto che codici numerici. Il linguaggio è tradotto in linguaggio macchina da un programma chiamato "assemblatore".

Linguaggio di programmazione (0.2) [Programming Language].
Un insieme di regole e parole speciali e simboli che sono usati assieme per costruire un programma.

Linguaggio Macchina (5.3) [Machine Language].
I valori dei codici che l'hardware dell'elaboratore interpreta come istruzioni.

Lotti (elaborazione a) (7.3) [Batch].
Locuzione usata per descrivere la maniera in cui dei programmi sono eseguiti su un elaboratore (generalmente di grande capacità) che accetta i suoi input da pacchi di schede perforate, e dà i suoi output all'utente su stampante.

Listato (1.10) [Listing].
Una copia stampata di un programma o l'emissione di un programma.

Mantissa (5.6).
La parte intera di un numero in virgola mobile. Sulla maggior parte degli elaboratori è implicito che la mantissa ha un valore equivalente al valore intero diviso per z elevato alla potenza N (se N è il numero di bits del campo mantissa).

Maxi-elaboratore (0.4) [Maxi-computer].
Un grosso elaboratore capace di servire parecchi utenti contemporaneamente.

Memoria (5.3) [Memory].
Vedete "Memoria principale".

Memoria principale (5.4) [Main Memory].
Un gran numero di registri di deposito raggruppati assieme ed indirizzabili secondo l'ordine numerico nel quale appaiono. La memoria principale è dove sono temporaneamente depositati i valori delle variabili mentre il programma gira.

Micro-elaboratore (0.4) [Micro-computer].
Un piccolo elaboratore che solitamente costa pochissimi milioni e che nella maggior parte dei casi usa come CPU un circuito integrato detto microprocessore.

Mini-elaboratore (0.4) [Mini-computer].
Un elaboratore di piccole-medie dimensioni che molto spesso è suddiviso tra più persone contemporaneamente. Nel 1981 il termine minielaboratore si applica a macchine il cui costo va generalmente da L. 12.000.000 a L. 180.000.000.

Modello (6.8) [Model].

Questo termine è spesso applicato a dei programmi che simulano il comportamento di qualche sistema o processo sia naturale che del mondo degli affari umani.

Modulo (5.3) [Module].

Una parte logicamente separata di un più vasto programma o di un sistema. Un sistema "modulare" è uno strutturato a moduli.

Nidificare (3.10) [Nesting].

Fa riferimento alla situazione in cui un costrutto logico ha un costrutto simile al suo interno. Un'< espressione > racchiusa tra parentesi, ad esempio, può nidificare in un'espressione più grande. Una procedura può nidificare in un'altra. Una istruzione composta può nidificare in un'altra, e così via.

Nodo (6.5) [Node].

Un punto di diramazione ad una foglia in un diagramma ad "albero".

Normalizzazione (5.9) [Normalize].

L'azione di allineare le cifre di un numero in modo da soddisfare le richieste delle operazioni di addizione/sottrazione, o della memorizzazione per avere la massima precisione.

Numero casuale (5.11) [Random Number].

Un numero scelto a caso (come se fosse "estratto da un cappello") da uno specificato insieme di numeri. Ogni volta che si ha un nuovo numero casuale, dovrebbe (in teoria) non avere relazione alcuna con quelli estratti precedentemente.

Operando (5.4) [Operand].

Un elemento di informazione messo in un contesto nel quale può essere manipolato da un elaboratore in una qualche maniera specificata.

Operare (0.2) [Operate].

Eseguire una sequenza di azioni su operandi per arrivare al risultato voluto.

Operatore (2.10) [Operator].

Il termine è usato per simboli PASCAL che richiedono che vengano eseguite certe operazioni. Per esempio il simbolo ":=", chiamato "operatore di assegnamento", richiede che venga eseguito l'assegnamento di un valore.

Parametro (2.5) [Parameter].

Un mezzo per passare un messaggio ad una procedura o ad una funzione del punto in cui il programma chiama la procedura/funzione stessa.

Parametro effettivo (2.5) [Actual Parameter].
Un parametro effettivo è una <variabile> o <espressione> passata nella chiamata di una procedura o funzione; esso sostituisce il parametro *formale* che compare nella dichiarazione della procedura o funzione.

Parametro formale (2.5) [formal parameter].
Un identificatore dichiarato parametro nell'intestazione di una procedura o funzione è detto parametro "formale". Quando la procedura o la funzione saranno poi chiamate, il parametro sarà sostituito da un parametro "effettivo".

Parametro valore (4.7). [Value parameter] oppure [Call-by-value parameter].
Un *parametro valore* è una variabile locale il cui valore è posto uguale al valore del parametro effettivo quando inizia l'esecuzione di una procedura o funzione.

Parametro variabile (4.7) [Variable parameter] oppure [Call-by-name parameter].
Nella dichiarazione di una procedura o funzione gli identificatori di parametri variabile sono preceduti dall'identificatore riservato "VAR". Un parametro variabile è un nome di "comodo" che è usato in fase di compilazione, ma che sarà sostituito dall'identificatore del parametro effettivo quando è chiamata la procedura o funzione.

Parola (5.4) [word, Word Size].
Una raccolta di bits che formano il contenuto di un registro di memoria. La dimensione della parola è il numero di bits contenuti in una parola.

Parola riservata (1.8) [Reserved word].
Una parola di un linguaggio di programmazione che ha un significato predefinito o che ha un significato speciale per il compilatore.

Passare (2.5) [Pass].
Il valore od il nome di un parametro è detto che viene "passato" (come un messaggio) come parte del processo di chiamata di una procedura o funzione.

Passo-passo (1.10) [Single step].
Il fatto che il programma esegua un'istruzione alla volta.

Peso (5.8) [Place value].
La potenza della "base" del sistema numerico che corrisponde alla posizione della cifra. Per esempio, il peso della cifra 2 nel numero 234 è 100 nel sistema decimale. Il peso della cifra "1" più a sinistra nel numero binario 0101 è 4 (cioè 2 elevato al quadrato).

Pila (4.2) [Stack].
Un elenco di dati messi in modo tale che il primo dato che può essere tolto è l'ultimo che è stato messo.

Precedenza (2.10) [Precedence].
In un'<espressione> contenente parecchi operatori distinti, le regole di precedenza determinano l'ordine in cui le corrispondenti operazioni vengono eseguite.

Predecessore (9.8) [Predecessor].
Nella definizione di un <tipo> scalare il predecessore di un elemento è l'elemento adiacente che compare appena prima nell'elenco.

Presentare (13.4) [Monitor].
Presentare un programma è l'azione di tracciare il suo flusso di esecuzione mentre gira e ciò può essere fatto visualizzando il contenuto di variabili selezionate senza interrompere l'esecuzione.

Problemi (0.2) [Problems].
I problemi dati nei primi capitoli del libro sono dati affinché siano svolti usando carta e penna. Devono essere distinti dagli esercizi che sono preparati per essere risolti con l'elaboratore.

Procedura (2.3) [Procedure].
Un sottoprogramma, vale a dire una parte superata di un programma che può essere mandata in esecuzione chiamandola per nome.

Programma (0.2) [Program].
Una sequenza di dichiarazioni di variabili seguita da istruzioni eseguibili o istruzioni che specificano una sequenza logica di calcoli che devono essere eseguiti.

Programmazione strutturata (0.3) [Structured programming].
Un approccio ordinato alla programmazione che enfatizza il fatto si spezzare delle sequenze logiche lunghe e complicate in moduli più piccoli, ciascuno dei quali esegue un compito che è concettualmente separato e distinto dalle altre parti del programma.

Puntatore (3.9) [Pointer].
In questo libro il termine "puntatore" è generalmente usato per far riferimento ad una variabile intera, il cui valore punta ad uno specifico elemento di un vettore o di una variabile <stringa>. PASCAL permette l'uso di variabili di <tipo> puntatore che sono però oltre gli scopi di questo libro.

Radice (6.5) [Root].
Termine usato in questo libro per far riferimento al nodo base di un diagramma di struttura o di un'altra struttura ad albero.

Raffinamenti successivi (6.6) [Step-wise refinement].
Il metodo raccomandato per sviluppare un programma è di partire con una prima descrizione e raffinarla quindi aggiungendo progressivamente nuovi (successivi) dettagli.

REAL (5.6)
Un < tipo > predichiarato in PASCAL introdotto per memorizzare numeri in virgola mobile.

RECORD (7.2)
Una struttura dati che può contenere parecchi dati tra loro correlati ma che possono essere di < tipo > diverso.

Recursione (4.2) [Recursion, Recursive].
Una procedura o funzione si dice recursiva se chiama se stessa. La recursione è il processo associato a procedure e funzioni recursive.

Registro (5.4) [Register].
Un gruppo di componenti per la memorizzazione binaria usati assieme. Questo termine è solitamente applicato ai veloci componenti per la memorizzazione interna alla CPU. Anche se la memoria è composta da un gran numero di registri più lenti, il *termine* registro è usato assai raramente e si parla in genere di *locazione*, *elemento* o *cella*.

Ricerca (13.2) [Search].
Il processo di esaminare un insieme di dati in modo sistematico per determinare se un particolare valore può essere presente, e se così localizzare la sua posizione.

Ricerca binaria (13.5) [Binary search].
Un metodo di ricerca nel quale un insieme di dati è diviso in due, quindi la metà contenente l'elemento è divisa in due, quindi il quarto contenente l'elemento cercato è diviso in due, e così via.

Riga (8.8) [Row].
Una linea orizzontale d'entrata in una tabella o di una rappresentazione tabellare di un vettore.

Ritorno (7.4) [Backspace].
Un tasto che si trova su parecchie tastiere. È generalmente usato per indicare

che il cursore deve tornare indietro di una posizione rispetto alla posizione corrente. Non tutti i dispositivi di stampa hanno questa caratteristica. Se manca il tasto, solitamente lo stesso effetto si ottiene con <control – H>.

Ritorno (2.11) [Return].
Dopo aver completato un'elaborazione, una procedura ritorna il controllo (fa continuare il programma) nel punto che segue immediatamente il punto in cui la procedura è stata chiamata. Quando una funzione ritorna, lascia un valore al posto del suo identificatore dentro un'espressione.

Saltare (11.2) [JUMP].
Termine usato per descrivere il processo mediante il quale un elaboratore blocca la sua normale elaborazione sequenziale e bruscamente inizia l'elaborazione da un punto diverso del programma.

Scambio (14.4) [Exchange].
Descrive il processo per cui una variabile A assume il valore di B e B assume il valore di A.

Scandire (3.9) [Scan].
Termine usato per descrivere il processo di esaminare una stringa per determinare se un particolare carattere o configurazione è presente.

Schema a blocchi (3.3) [Flow-chart].
Un diagramma che rappresenta la sequenza logica di elaborazione seguita da un programma. Si seguono le sequenze di azioni in uno schema a blocchi nell'identica maniera in cui si seguono, su una cartina, le strade da città a città.

Schermo (0.5) [Display].
In questo libro il termine schermo sta ad indicare un dispositivo usato per visualizzare le informazioni in uscita da un elaboratore; spesso è simile ad uno schermo televisivo.

Segnale (5.4) [Signal].
Un impulso elettrico che fa si che una parte dell'hardware dell'elaboratore esegua certe azioni.

Semantica (7.4) [Semantics].
Un insieme di regole che descrivono le azioni o i risultati che ci si aspetta quando è usato un costrutto di un linguaggio di programmazione.

Serie (5.10) [Series].
Un'espressione aritmetica contenente un grande (possibilmente infinito) nu-

mero di sotto-espressioni chiamate "termini", i quali sono generati secondo certe regole logiche.

Set (9.2)
In PASCAL un set (insieme) è una variabile costruita in modo simile ad un vettore ma che contiene solo valori booleani. Ciascun elemento del set, quando è TRUE, è detto essere un "membro presente" del SET. Un set può corrispondere al possibile intervallo di valori nei < tipo > scalare, sottocampo o CHAR.

Sintassi (1.8) [Syntax].
Un insieme di regole che descrivono come deve essere scritto un programma corretto. Un modo conciso per descrivere queste regole sono le carte sintattiche.

Sistema (1.2) [System].
Una raccolta, spesso ampia, di entità interagenti. In questo libro, il termine "sistema" è generalmente applicato al gruppo di programmi che danno la possibilità di sviluppare e far girare programmi PASCAL.

Software (1.2)
Una raccolta di grossi programmi che fanno eseguire, alle istruzioni primitive dell'hardware dell'elaboratore, funzioni più complesse ad uso delle persone.

Sottocampo (9.5) (9.2) [Subrange, Subrange Variabile].
Una parte dell'insieme di tutti i valori che un < tipo > di variabile può assumere. Un sottocampo include tutti i valori da un minimo specificato (limite inferiore) ad un massimo (limite superiore).

Stato (5.4) [State].
Lo stato di un elaboratore, o di qualsiasi dispositivo a logica binaria, è descritto dal valore di tutti i bits di tutti i registri. È possibile ritornare in qualsiasi stato, da qualsiasi altro, posizionando di nuovo tutti i bits ai valori di quello stato.

Struttura base (5.3) [Main frame].
Il principale componente hardware di un elaboratore.

Struttura dati (8.2) [Data structure].
Un'organizzazione logica per variabili correlate che riflette le relazioni tra i vari elementi.

Struttura gerarchica (6.5) [Hierarchic Structure].
Altro nome delle strutture ad "albero". In una struttura gerarchica, tutte le connessioni logiche partono da un nodo detto radice, e non sono permesse connessioni in orizzontale.

Successore (9.8) [Successor].

Il successore di un elemento nella definizione di un < tipo > scalare è l'elemento che segue immediatamente quell'elemento nella dichiarazione.

Suggerire (7.3) [Prompt]

Un programma visualizza un breve messaggio per suggerire/invitare l'utente a rispondere con l'immissione desiderata dalla tastiera.

Tabella di verità (3.9) [Truth table].

Una tabella che mostra le relazioni tra i valori di un'espressione booleana ed i valori delle variabili costituenti l'espressione.

Terminale (7.2) [Terminal].

Un dispositivo, solitamente costituito da uno schermo o una stampante più una tastiera, tramite il quale un utente comunica con l'elaboratore.

Termine (5.10) [Term].

Una parola specializzata che si applica ad un elemento usato nelle sintassi delle espressioni aritmetiche e booleane.

Tipo (I.5) [Type].

Descrizione formale del genere di informazioni che possono essere memorizzate sotto forma di variabili in un programma PASCAL.

Top Down (6.6)

Questo termine descrive una metodologia di sviluppo di programmi ed algoritmi nella quale si parte alla radice del diagramma di struttura, e quindi progressivamente si aggiungono nodi dettagliati.

Tracciare (2.4) [Trace].

Si può tracciare il percorso logico dei passi che un programma compie inserendo, in punti strategici, delle istruzioni WRITELN che mostrino i valori di variabili selezionate quando il programma raggiunge quei punti. Il programma può essere temporaneamente fermato per visualizzare i valori da studiare, aggiungendo l'istruzione REAOLN dopo l'istruzione WRITELN.

Transazione (8.8) [Transaction].

Questo termine ha la sua origine nei problemi gestionali. Si applica ad una sequenza di passi di elaborazione che seguono l'immissione di un record e che continua fino a che non è completata qualche emissione direttamente collegata.

Troncamento (5.7) [Truncation].

È il processo per cui i bits meno significativi di un numero binario risultante da

un'operazione aritmetica sono persi per il fatto che tutti i bits non possono essere contenuti nella parola di memoria.

Unione (9.6) [Union].
L'unione di due insiemi è un insieme i cui elementi sono tutti gli elementi dei due insiemi originali.

Unità centrale di elaborazione (5.3) [Central processing unit].
La parte principale dell'hardware di un elaboratore che interpreta ed esegue le istruzioni in linguaggio macchina. Spesso è chiamata "CPU".

Utente (7.3) [User].
Un utente è una persona che sta facendo girare un programma o che in qualche modo lavora con l'elaboratore.

Variabile (2.7) [Variable].
Un nome dato ad una locazione di memoria dove può essere memorizzato, per uso futuro, un dato o un gruppo di dati associati.

Variabile con indice (8.4) [Subscripted variable].
Un riferimento ad un elemento di un vettore, costituito da un identificatore di vettore seguito da uno o più espressioni dell'indice racchiuse tra parentesi quadre.

Variabile di controllo (3.7) [Control variable].
Una variabile di tipo intero, carattere o scalare a cui è dato un valore iniziale ed è poi incrementata o decrementata nel controllo dell'istruzione FOR.

Variabile globale (2.8) [Global variable].
Una variabile dichiarata nel <blocco> del programma principale è detta "globale" per ricordare che vi si può far riferimento dal programma principale come da qualsiasi procedura.

Variabile locale (2.8) [Local variable].
Una <variabile> locale è una variabile dichiarata nello stesso <blocco> in cui è usata.

Variabile scalare (9.2) [Scalar variable].
Una <variabile> PASCAL che può assumere uno qualsiasi di parecchi valori non-numerici rappresentati da un elenco di identificatori nella dichiarazione del <tipo> scalare.

Variante (9.3) [Variant].
Un mezzo che permette di dare significati diversi all'ultimo campo di un record.

Vettore (8.2) [Array].
Una struttura dati che può contenere parecchi "elementi" tutti dello stesso < tipo >.

Virgola mobile (5.6) [Floating-point].
Un numero in virgola mobile è un numero che è memorizzato in modo che il punto decimale (o il punto binario) non è in relazione fissa con i bits di una parola di memoria.

Vuoto (1.9) [Empty].
Vuoto si dice di un'entità descritta da una carta sintattica che può essere affatto priva di contenuto (solitamente opzionale).

Zero in testa (5.9) [Leading Zeroes].
Cifre zero che compaiono alla sinistra della cifra più significativa diversa da zero in un numero.

APPENDICE C

PROCEDURE E FUNZIONI INTERNE

1. ABS(X)
   Funzione che dà il valore assoluto del parametro, intero o reale X. Il risultato è dello stesso < tipo > del parametro effettivo.

2. ATAN(X)
   Funzione che dà come risultato — reale — il valore dell'arcotangente (X), in cui X è reale. Il valore calcolato è reale ed in radianti.

3. CHR(X)
   Funzione che dà come risultato il carattere il cui valore ordinale è X. Per esempio, supponendo che CH è una variabile di < tipo > CHAR, le due seguenti istruzioni sono equivalentiu:

   CH: = ' '; CH: = CHR(32)

4. CLEARSCREEN
   Procedura UCSD che pulisce lo schermo su un terminale grafico di tipo CRT e che porta la tartaruga in posizione naturale al centro dello schermo.

5. CONCAT(S1, S2, ...)
   Una funzione UCSD che dà un risultato di < tipo > STRING. La stringa risultato è ottenuta concatenando tutti i parametri stringa S1, S2, ..., in cui possono essere usati 2 o più parametri effettivi.

6. COPY(SORGENTE, INDICE, DIMENSIONE)
   Funzione UCSD che dà un risultato di < tipo > STRING. Il parametro effettivo SORGENTE è di < tipo > STRING, mentre INDICE e DIMENSIONE sono di < tipo > INTEGER. La stringa di ritorno è ottenuta copiando i primi DIMENSIONE caratteri della stringa SORGENTE, partendo dal carattere SORGENTE[INDICE].

7. COS(X)
   Funzione che dà come risultato – reale – il coseno di X, in cui X è il parametro effettivo reale espresso in radianti.

8. DELETE(SORGENTE, INDICE, DIMENSIONE)
   Procedura UCSD che toglie DIMENSIONE caratteri dalla SORGENTE partendo da SORGENTE[INDICE]. SORGENTE è di < tipo > STRING, mentre INDICE e DIMENSIONE sono di < tipo > INTEGER.

9. EOF(F)
   Funzione booleana che dà TRUE dopo che il carattere < ETX > (fine flusso) è stato letto dal flusso F. In ogni altro caso si ottiene FALSE. Se è omesso l'identificatore F, si assume, per difetto, il flusso INPUT.

10. EOLN(F)
    Funzione booleana che dà il valore TRUE se è stato letto il carattere di fine linea (per difetto < RET > o ritorno carrello) alla fine della più recente istruzione di READ. In ogni altro caso si ottiene FALSE. Se è omesso l'identificatore di flusso F, si assume, per difetto, il flusso INPUT.

11. EXIT(P)
    Procedura UCSD che, come unico parametro, accetta un identificatore di procedure. La più recente attivazione della procedura sarà terminata normalmente. I risultati non sono definiti se P non è attiva, e ne può derivare una fine normale del programma.

12. EXP(X)
    Funzione che dà come risultato – reale –, il valore della costante "e" elevata alla potenza X; X può essere intera o reale.

13. INSERT(SORGENTE, DESTINAZIONE, INDICE)
    Procedura UCSD che inserisce il valore della variabile SORGENTE nella variabile DESTINAZIONE partendo prima del carattere DESTINAZIONE[INDICE]. SORGENTE e DESTINAZIONE sono di < tipo >STRING, mentre INDICE è di < tipo > INTEGER.

14. LENGTH(S)
    S è di < tipo > STRING e la funzione dà come risultato intero il numero di caratteri contenuti in S.

15. LN(X)
    X è intera o reale e la funzione dà come risultato reale il valore del logaritmo naturale di X.

16. LOG(X)

    X è intero o reale e la funzione dà come risultato reale il valore del logaritmo in base 10 di X.

17. MOVE(DISTANZA)

    Procedura UCSD che fa sì che il cursore (la "tartaruga") si muova a DISTANZA unità video lungo la direzione corrente. DISTANZA è di < tipo >INTEGER.

18. MOVETO(XPOS, YPOS)

    Procedure UCSD che fa sì che il cursore (la "tartaruga") si muova sulla posizione (XPOS, YPOS) dello schermo. Questa posizione si trova a XPOS unità video partendo dal centro verso destra, e YPOS unità video sopra il centro. Se XPOS è negativo la tartaruga si sposta verso sinistra. Se YPOS è negativo la tartaruga si sposta verso il basso. Mentre MOVE fa sì che la tartaruga si muova *relativamente* alla posizione corrente, MOVETO la fa muovere in *assoluto* ad una qualsiasi posizione dello schermo.

19. ODD(X)

    Funzione Booleana che ritorna il valore TRUE se il parametro intero X ha un valore dispari, altrimenti ritorna FALSE.

20. ORD(X)

    Funzione Booleana che ritorna come risultato intero il valore ordinale del parametro X. X deve essere di < tipo > Scalare o di < tipo > CHAR o BOOLEAN.

21. PAGE(F)

    Procedura che fa sì che il prossimo WRITE o WRITELN sul flusso F appaia su una nuova pagina, se F è un flusso il cui contenuto deve essere inviato ad una stampante. Nel caso di un terminale interattivo con schermo, PAGE cancella lo schermo, ed il cursore è spostato sull'angolo in alto a sinistra.

22. PENCOLOR(COLOR)

    Una procedura UCSD usata per cambiare il "colore" dell'"inchiostro" usato dal cursore"Tartaruga". Il parametro COLOR è di < tipo > scalare e può assumere uno dei seguenti valori:

    NONE: La Tartaruga non scrive o disturba il contenuto dello schermo quando si muove.

    WHITE: La Tartaruga disegnerà linee luminose sullo schermo quando viene mossa.

    BLACK: La Tartaruga disegnerà linee "spegnendo" i punti luminosi su

uno schermo mappato a bit. (Applicabile solo a dispositivi con mappatura a bit).

23. POS(CONFIGURAZIONE, SORGENTE)
    Una funzione UCSD che ritorna come risultato intero l'indice in SORGENTE che indica dove è stata trovata CONFIGURAZIONE la prima volta. CONFIGURAZIONE e SORGENTE sono di < tipo > STRING. Se CONFIGURAZIONE non è trovato è ritornato in valore 0 (zero).

24. PRED(X)
    Funzione che ritorna come risultato il predecessore del parametro scalare X. Se il valore di X è il limite inferiore dell'intervallo del < tipo > scalare, ne risulterà una fine per indice non valido.

25. ROUND(X)
    Funzione che ritorna come risultato intero, il valore del parametro reale X arrotondato all'intero più vicino in accordo con la seguente definizione:

$$\begin{aligned} \text{ROUND}(X) &= (\text{TRUNC}(X + 0.5)) \text{ se } X > 0 \\ &= (\text{TRUNC}(X - 0.5)) \text{ se } X < 0 \end{aligned}$$

26. SIN(X)
    Funzione che ritorna come risultato reale il valore di sen(X) dove X è un parametro reale il cui valore è un angolo espresso in radianti.

27. SUCC(X)
    Funzione che ritorna come suo risultato il successore di un parametro scalare X. Se il valore di X è uguale al limite superiore dell'intervallo dichiarato del < tipo > scalare, il programma terminerà in modo anomalo per indice non valido.

28. SQR(X)
    Funzione che ritorna come suo risultato il quadrato del parametro intero o reale X. Il risultato sarà dello stesso < tipo > del parametro.

29. SQRT(X)
    Funzione che ritorna come risultato reale la radice quadrata del valore del parametro reale X.

30. TRUNC(X)
    Funzione che ritorna come risultato intero il valore del parametro reale X troncato al più grande intero minore od uguale ad X per X > 0, od il più piccolo intero maggiore o uguale ad X se X < 0.

31. TURN(ANGOLO)

    Una procedura UCSD che fa sì che la direzione della Tartaruga ruoti di ANGOLO gradi (ANGOLO è di <tipo> INTEGER). Se ANGOLO ha un valore positivo la rotazione è antioraria, se negativo oraria.

32. TURNTO(ANGOLO)

    Una procedura UCSD che cambia la direzione della Tartaruga proprio al valore di ANGOLO. TURN fa una variazione *relativa* alla direzione attuale. TURNTO pone la Tartaruga in una direzione assoluta.

33. WHEREAMI(POSX, POSY, DIREZIONE)

    Una procedura UCSD associata ai grafici Tartaruga con tre parametri variabile nei quali sono ritornati i valori della posizione corrente e della direzione della Tartaruga. Tutti e tre i parametri devono essere di <tipo> INTEGER. POSX e POSY sono le distanze dal centro dello schermo. DIREZIONE è un angolo misurato in gradi in senso antiorario dalla direzione orizzontale alla direzione della Tartaruga.

APPENDICE D

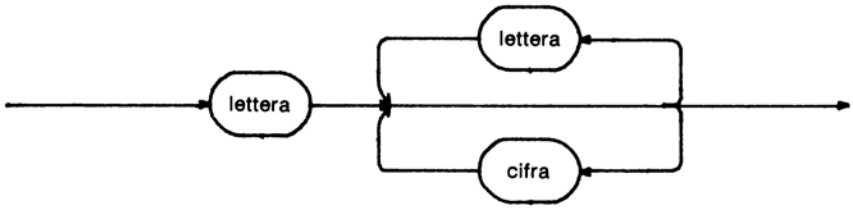
INDICE

Ciascuna voce fa riferimento a <numero capitolo>. <numero sezione>. Per esempio 2.5 significa Capitolo 2 sezione 5. In linea di massima questo indice contiene solo riferimenti a costruttori specifici del linguaggio PASCAL. Si vede, in Appendice B, il glossario per riferimenti ad altri termini.

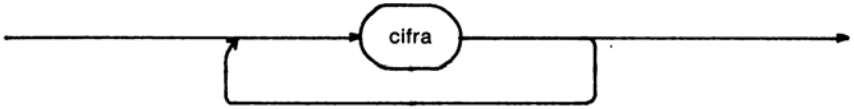
APPENDICE E

CARTE SINTATTICHE

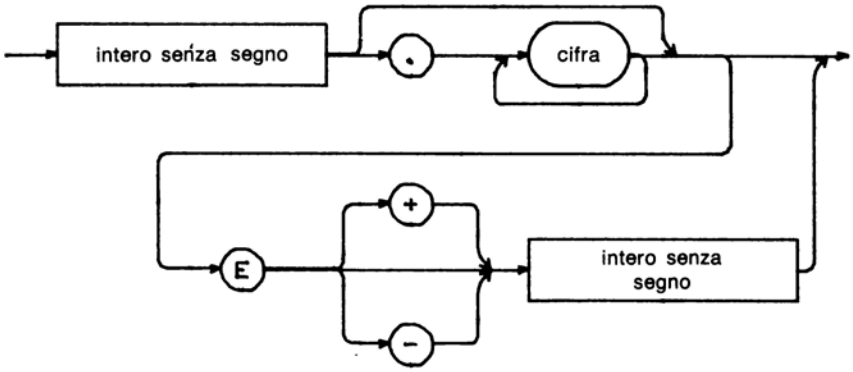
<identificatore>



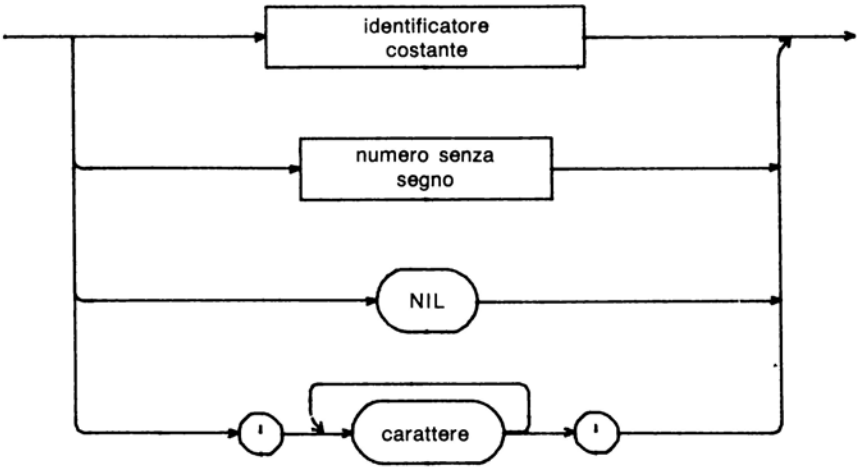
<intero senza segno>



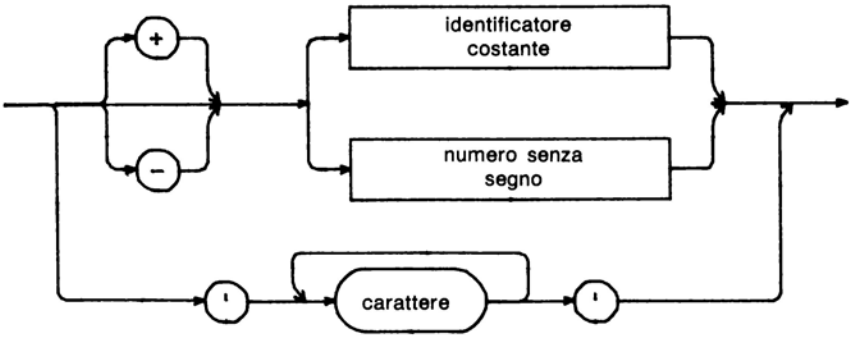
<numero senza segno>



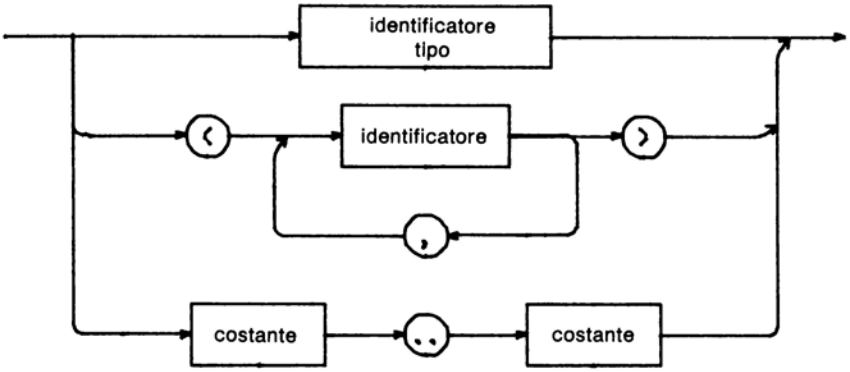
<costante senza segno>



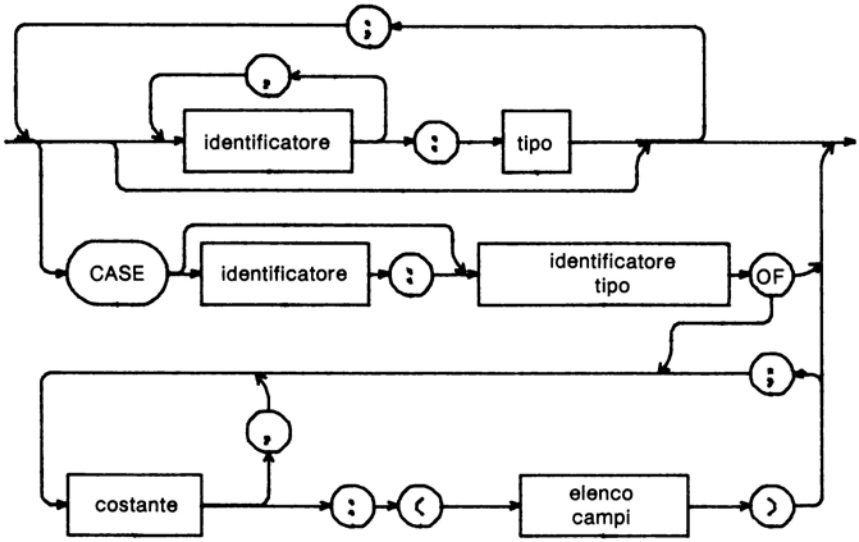
<costante>



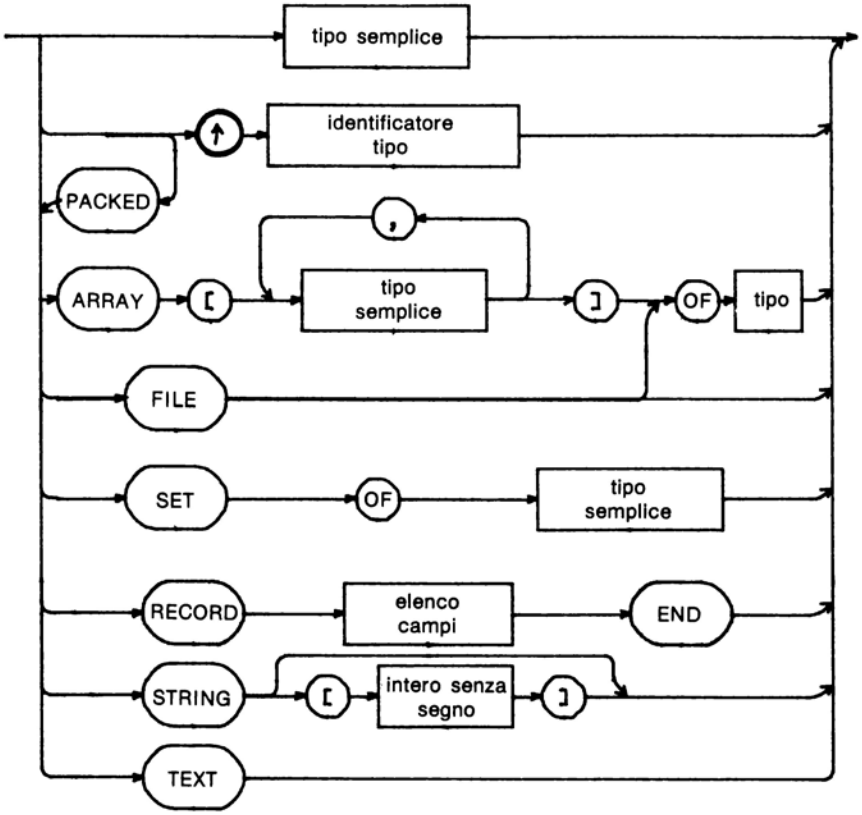
<tipo semplice>



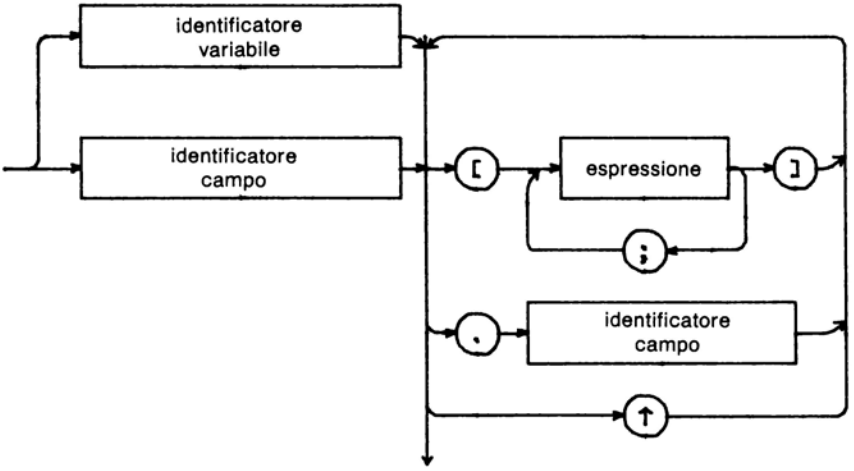
<elenco campi>



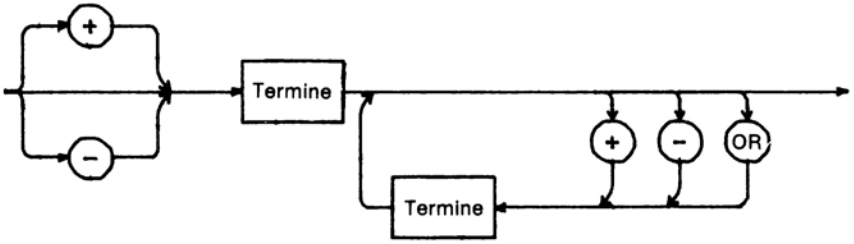
<tipo>


tipo semplice
↑
identificatore tipo
PACKED
,
ARRAY
[
tipo semplice
]
OF
tipo
FILE
SET
OF
tipo semplice
RECORD
elenco campi
END
STRING
[
intero senza segno
]
TEXT

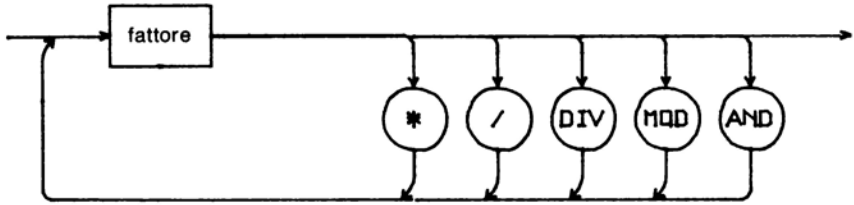
<variabile>



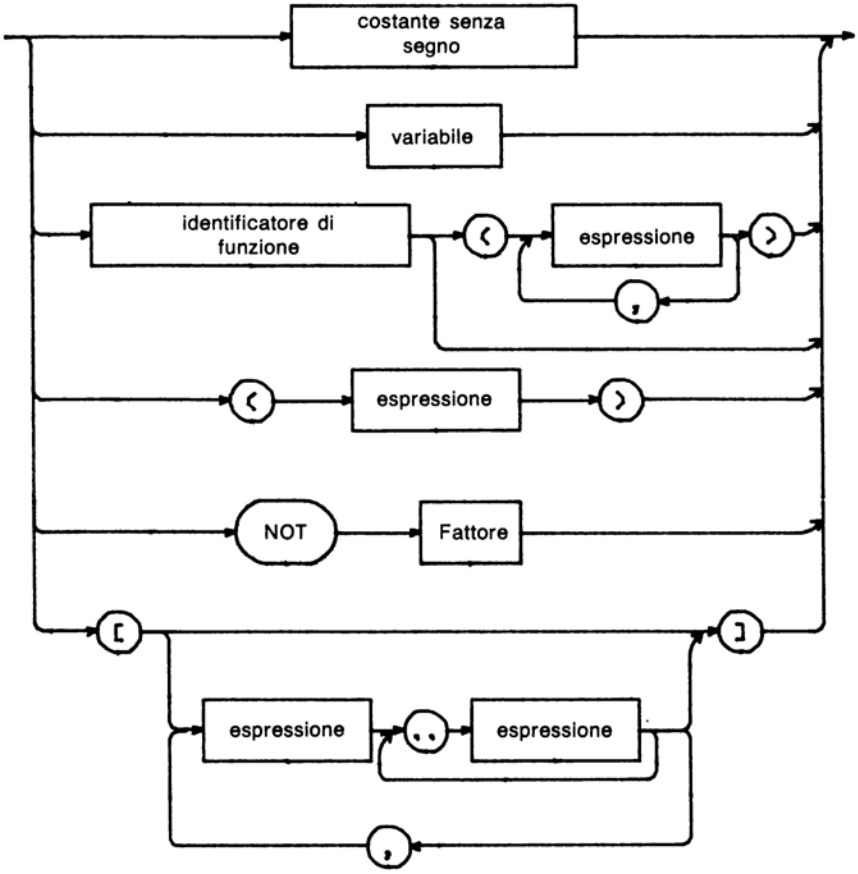
<espressione semplice>



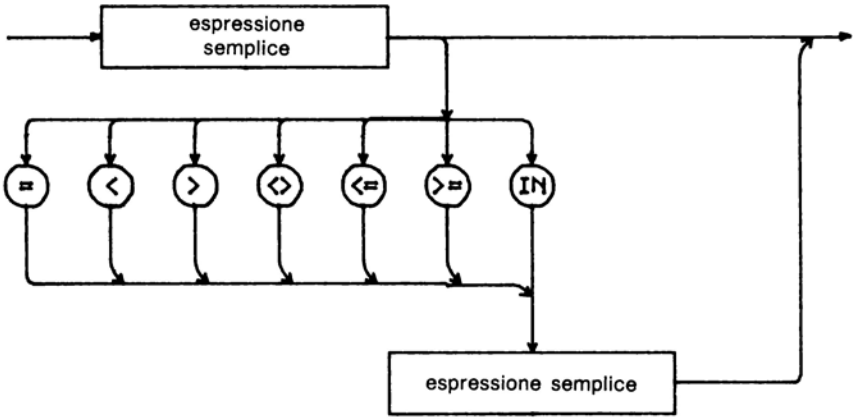
<termine>



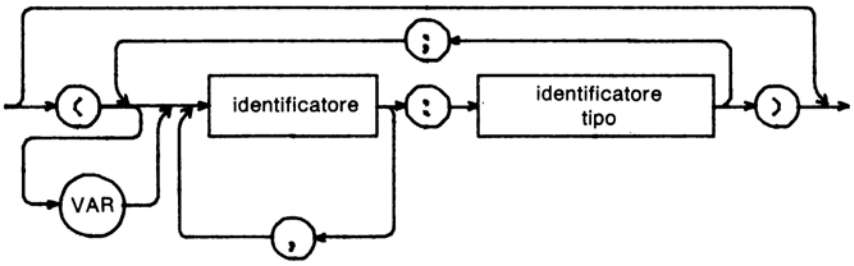
<fattore>

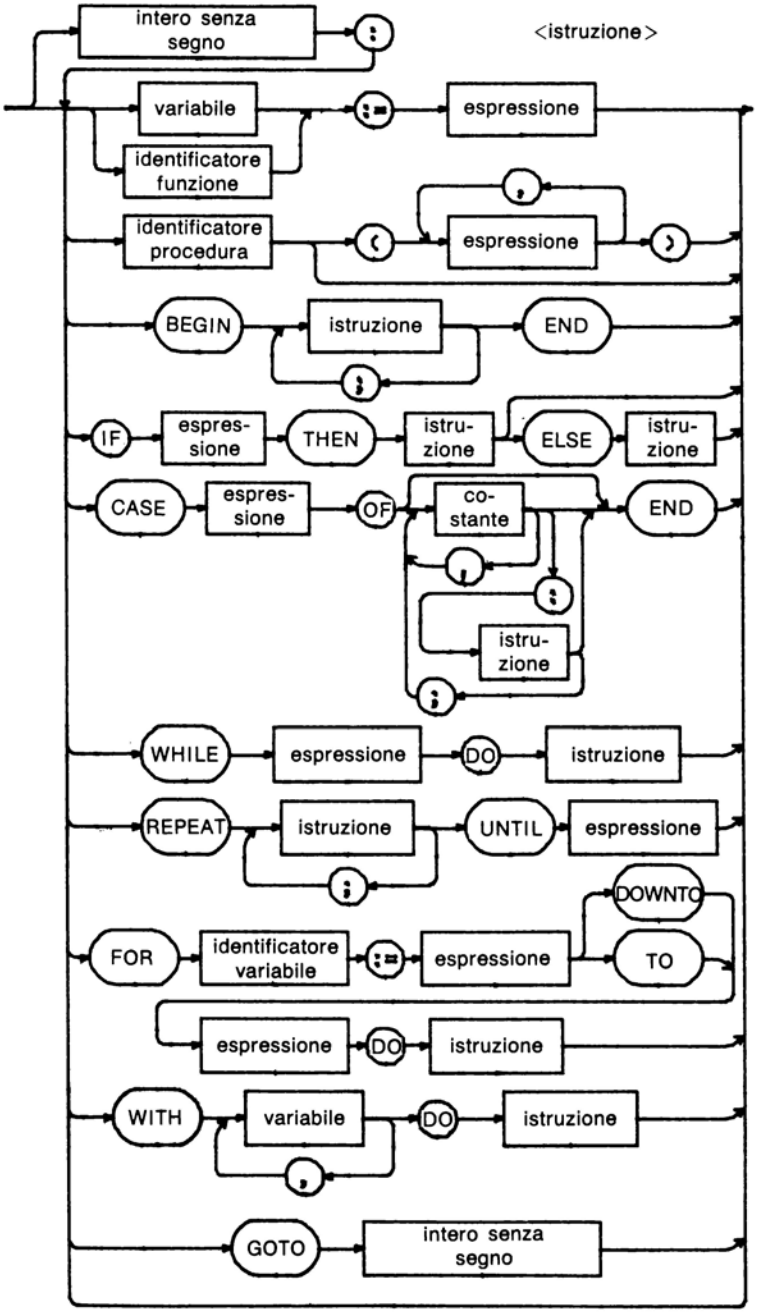

costante senza segno
variabile
identificatore di funzione
(
espressione
)
,
(
espressione
)
NOT
Fattore
[
]
espressione
..
espressione
,

<espressione>

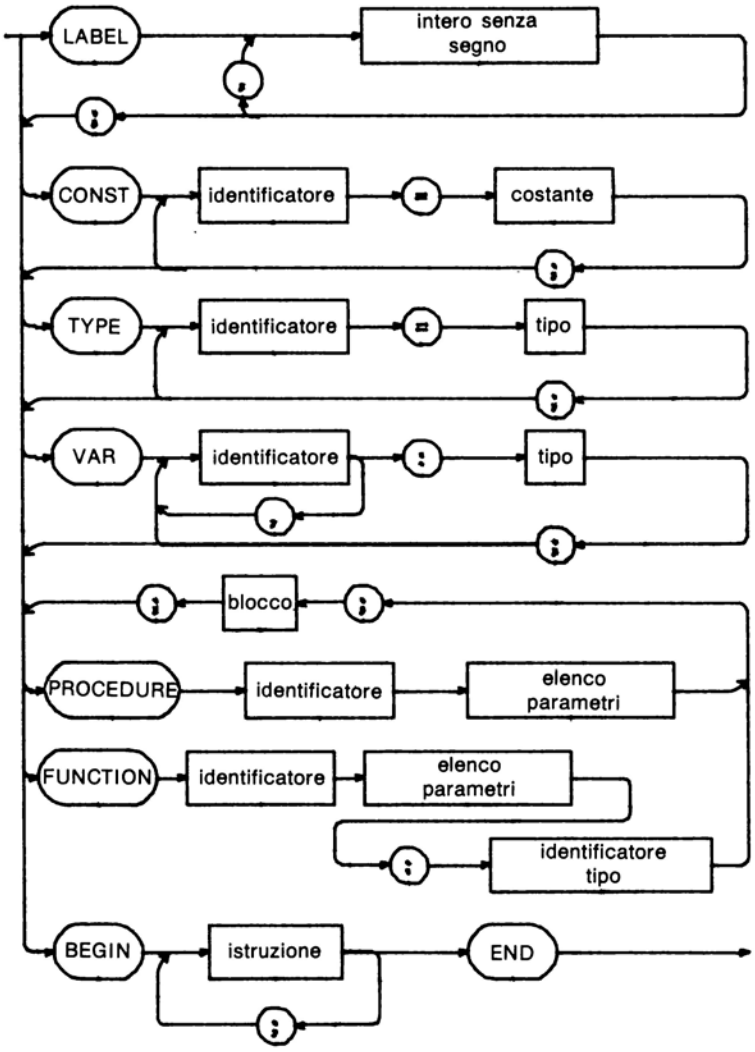


<elenco parametri>




<istruzione>
intero senza segno
:
variabile
:=
espressione
identificatore funzione
identificatore procedura
(
,
)
BEGIN
istruzione
END
;
IF
espres-sione
THEN
istru-zione
ELSE
CASE
OF
co-stante
WHILE
DO
REPEAT
UNTIL
FOR
identificatore variabile
DOWNTO
TO
WITH
GOTO

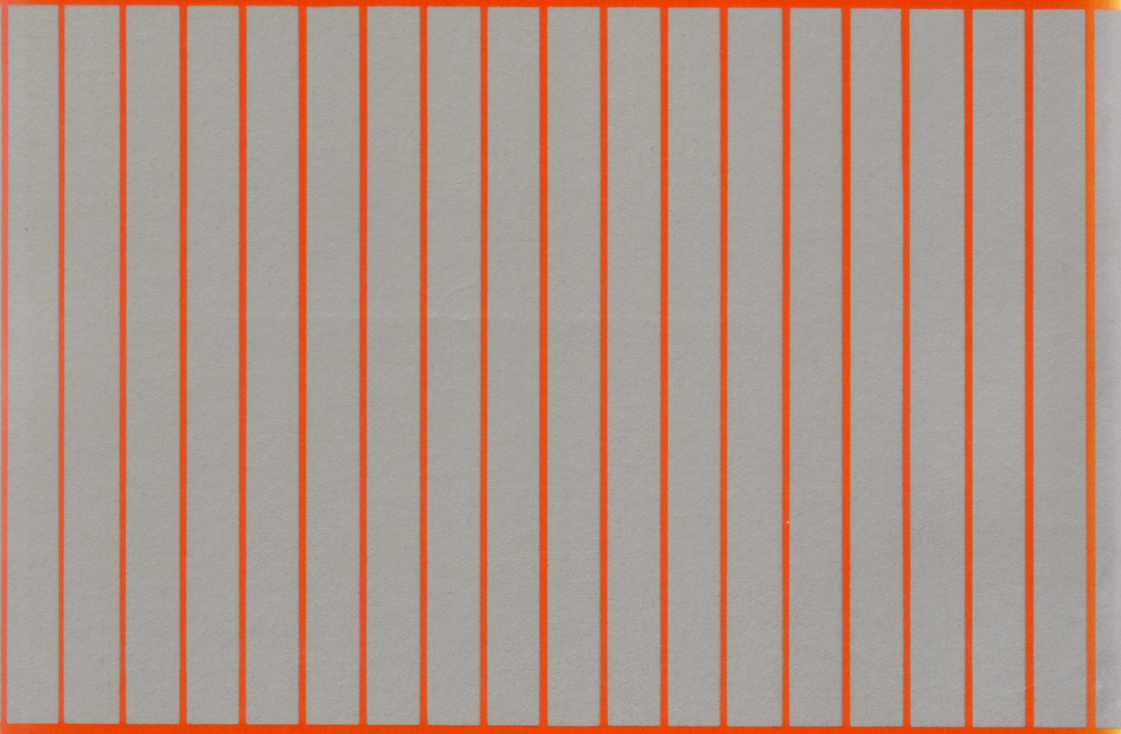
<blocco>



<programma>

L'obiettivo principale di questo libro è di insegnarvi un approccio disciplinato alla soluzione di problemi usando un elaboratore. Come secondo ed inseparabile obiettivo, voi dovete imparare a scrivere programmi. Il libro si occuperà poco di descrivere gli elaboratori, o di esaminare i numerosissimi usi che se ne fa.

Il materiale è stato scelto per essere comprensibile a tutti gli studenti usciti dalle superiori; sostanzialmente non sono necessarie basi di matematica se non alcune nozioni di algebra. A dispetto dell'approccio non numerico, i metodi insegnati sono gli stessi di quelli tradizionalmente usati in problemi basati sulla matematica. Il testo dovrebbe servire, nello stesso modo, a studenti di scienze, lettere, arti o ingegneria per prepararli all'uso dell'elaboratore nei campi che hanno scelto. Coloro che non useranno mai più un elaboratore dovrebbero trarre beneficio nell'usare le stesse metodologie di soluzione di problemi in altri contesti.



L. 28.000 Cod. 512 P ISBN 88-7056-110-0

GRUPPO
EDITORIALE
JACKSON

Kenneth
L. Bowles

SOLUZIONE DI PROBLEMI CON PASCAL

56