Jim Welsh
John Elder

INTRODUZIONE AL PASCAL

ESA

Jim Welsh
John Elder

INTRODUZIONE AL PASCAL

Traduzione dall'inglese di:
Paolo Ciancarini e Daniele Nardi

ESA

Titolo originale: *Introduction to Pascal - Second Edition*
Original English language edition published by
Prentice-Hall International, Hemel Hempstead, Hertfordshire, England

Traduzione italiana di *P. Ciancarini e D. Nardi*

E.S.A. — Edizioni Scientifiche Associate - s.r.l.
00184 Roma - Via della Polveriera, 37 - Tel. (06) 4740952/462678

Prefazione alla traduzione in italiano

Il successo del linguaggio PASCAL, soprattutto nella didattica della programmazione, è sempre maggiore. Esso è infatti un linguaggio di tipo imperativo particolarmente adatto ad introdurre lo studente ai concetti della programmazione: sviluppo sistematico di programmi, scelta dei tipi di dato e delle strutture di controllo, analisi di correttezza e complessità, tecniche di documentazione, ecc.. È per questo che, come molti altri docenti di università italiane e straniere, abbiano adottato il PASCAL nel corso da noi tenuto Facoltà di Ingegneria dell'Università di Roma "La Sapienza".

Nonostante ultimamente siano apparsi vari libri in italiano sul PASCAL, abbiamo ritenuto opportuno proporre la traduzione del testo di Welsh e Elder in quanto lo consideriamo particolamente chiaro e ben organizzato. Gli autori sono riusciti a nostro avviso a raggiungere un duplice obiettivo: da una parte introducono il linguaggio PASCAL fornendo al lettore un manuale preciso, completo e comprensibile; dall'altra parte – attraverso esempi opportunamente scelti per evidenziare le potenzialità del linguaggio e le caratteristiche di ciascun suo costrutto – introducono concetti e metodi della programmazione.

Ringraziamo Paolo Ciancarini e Daniele Nardi per le cure e l'attenzione dedicate alla traduzione del testo e – lavoro tutt'altro che trascurabile – alle ristesura dei programmi, usando quanto più possibile termini mnemonici in italiano. I programmi, tranne quelli riportati in Appendice 2, sono stati compilati ed eseguiti usando il compilatore PASCAL del VAX 11/780 – UNIX.

Ringraziamo la ESA per aver affrontato con energia e capacità la preparazione di questo libro. In particolare il nostro ricordo grato va a Francesco Zuccaccia per l'entusiasmo con cui ha intrapreso questo lavoro.

Carlo Batini
Luigia Carlucci Aiello

Docenti del corso di Programmazione dei calcolatori elettronici
Facoltà di Ingegneria, Università di Roma "La Sapienza"

Roma, 15 Dicembre 1983

Indice

Prefazione alla seconda edizione

Dopo l'uscita della prima edizione di questo libro nel 1979 è stata concordata tra i rappresentanti delle competenti organizzazioni internazionali una nuova definizione del PASCAL. Questa definizione, nota come ISO DIS 7185, è ora in fase di ratifica formale come standard internazionale.

Per lo più la nuova definizione elimina piccole incongruenze presenti nella precedente, ma introduce in due aree altrettante caratteristiche addizionali. Queste riguardano l'uso di parametri procedure e funzioni da una parte e di parametri di tipo array dall'altra. Le modifiche aumentano sostanzialmente l'affidabilità e l'efficienza del PASCAL in tali aree, e sono già state incluse in molte delle principali implementazioni del linguaggio.

La seconda edizione di questo libro descrive il PASCAL esattamente com'è definito dal nuovo standard, ed include la descrizione delle nuove funzionalità e del loro corretto uso. In questa edizione i termini "la definizione del PASCAL" e "il PASCAL standard" vanno riferiti al documento DIS 7185.

Il linguaggio di programmazione PASCAL è stato sviluppato alla fine degli anni sessanta dal Prof. Niklaus Wirth alla Eidgenössische Technische Hochschule di Zurigo. Il suo scopo era di realizzare un linguaggio comprendente un piccolo numero di concetti fondamentali di programmazione sufficienti però ad insegnarla in forma di disciplina logica e sistematica; nello stesso tempo voleva un linguaggio che fosse facile ed efficiente da implementare sulla maggior parte dei calcolatori. Il suo successo nel conseguire questo obbiettivo può essere misurato dalla rapida diffusione dell'uso del PASCAL sia come linguaggio usato nella didattica dei principi di programmazione, sia come linguaggio effettivo per scrivere software di base e applicativo.

Questo libro fornisce un'introduzione completa al PASCAL e può essere usato sia da programmatori novizi sia da coloro che conoscono altri linguaggi di programmazione. In esso viene descritto tutto il linguaggio e ne vengono completamente illustrate le caratteristiche. Il linguaggio descritto non contiene funzionalità peculiari a nessuna particolare implementazione – grande cura è stata dedicata ad isolare ed evidenziare tutte le caratteristiche del PASCAL dipendenti dall'implementazione. Nel testo sono

implicitamente illustrati i principi più generali di programmazione. Lo stile di progettazione usato è consistente con le moderne tecniche di programmazione strutturata e di raffinamento incrementale.

Il materiale si basa su nove anni di corsi tenuti alla Queen's University di Belfast dai due autori, che hanno una grande esperienza nella didattica, nell'uso e nell'implementazione del PASCAL. La struttura del libro è quella adottata nei corsi menzionati e si presta bene sia per apprendere il PASCAL che per imparare a programmare per la prima volta.

Non sono esaminati nei dettagli gli altri aspetti dei sistemi di elaborazione di cui dovrebbe essere a conoscenza l'apprendista programmatore, però il capitolo 1 presenta un riassunto delle nozioni e della terminologia su cui si basano i capitoli successivi. I ca pitoli 3-6 introducono i principali tipi di dato ed istruzioni del PASCAL. Il capitolo 7 riguarda le procedure e le funzioni, l'uso delle quali è da noi considerato fondamentale nel processo di costruzione di un programma. Nei capitoli 9-13 sono introdotte le funzionalità di strutturazione dei dati del PASCAL – array, stringhe, record, insiemi, file e puntatori.

I costrutti del PASCAL sono presentati come segue:

(a) definizione del costrutto mediante la notazione e la terminologia della definizione del PASCAL standard;
(b) spiegazione del suo uso, e di qualsiasi limitazione e problema pratico di cui l'utente deve essere a conoscenza;
(c) illustrazione del suo uso per mezzo di frammenti di programma verificati ed inseriti nel testo.

Una caratteristica distintiva di questo libro è l'inserimento, ove opportuno, di uno o più programmi esemplificativi in ogni capitolo. Il libro contiene in tutto 17 esempi che illustrano l'uso delle varie funzionalità del PASCAL e degli algoritmi di calcolo di base in un contesto reale significativo. Il progetto dei programmi è sviluppato per raffinamento incrementale del problema; alla fine sono riprodotte le stampe del calcolatore comprendenti il listato finale ed i risultati ottenuti.

Quasi tutti i capitoli si concludono con una raccolta di esercizi di programmazione che richiedono l'applicazione delle funzionalità studiate nel capitolo. Tali esercizi richiedono modifiche o estensioni dei programmi mostrati precedentemente oppure la costruzione di nuovi programmi. In appendice 2 sono date le soluzioni di alcuni di questi esercizi.

Il libro vuole costituire sia un ausilio didattico sia un manuale d'uso. Per facilitare quest'ultima funzione include un'appendice di diagrammi sintattici, ed un indice analitico completo. I diagrammi sintattici forniscono un riassunto sintetico dei costrutti linguistici in una forma facile da usare da parte di chi abbia un pò di familiarità con le nozioni base del linguaggio. L'indice analitico elenca tutti i termini formali ed informali usati nel testo indicando il punto in cui vengono definiti e tutti i punti che potrebbero essere d'aiuto nel chiarire il loro significato.

Vogliamo ringraziare il Prof. Tony Hoare per averci incoraggiato per primo a scrivere questo libro, ed inoltre tutte le persone che hanno letto le bozze del manoscritto,

trovato i nostri errori, e dato suggerimenti costruttivi. In particolare ringraziamo Tony Addyman per il suo aiuto nel verificare che questa nuova edizione sia consistente con il nuovo standard del PASCAL. Di tutte le eventuali discrepanze rimanenti siamo completamente responsabili noi.

JIM WELSH
JOHN ELDER

1.
Calcolatori e Programmazione

La programmazione di un calcolatore richiede la comprensione della natura dei calcolatori, dei programmi e dei linguaggi in cui si possono scrivere i programmi. Nei capitoli successivi di questo libro viene spiegato come si possono scrivere programmi nel linguaggio PASCAL. Questo primo capitolo riassume i concetti generali concernenti i calcolatori e la loro programmazione, su cui si basano i capitoli successivi.

Per i lettori che avessero già programmato in altri linguaggi, il capitolo fornisce un riassunto della terminologia e una prima idea della filosofia di programmazione usate nei capitoli successivi.

Per coloro che imparano per la prima volta a programmare, questo capitolo riassume molto brevemente i concetti coi quali occorre familiarizzare per programmare in PASCAL o in qualsiasi altro linguaggio. L'insegnante di un corso può fornire una trattazione più dettagliata di questi argomenti prima o durante lo studio del materiale trattato nel libro.

1.1. IL CALCOLATORE = (creativo ad alto velocità)

Un *calcolatore* è una macchina che può eseguire ad altissima velocità sequenze di operazioni lunghe, complesse e ripetitive. Queste operazioni vengono applicate ad *informazioni* o *dati* forniti dall'utente per produrre ulteriori informazioni o *risultati* che l'utente richiede. La sequenza di operazioni necessarie per produrre i risultati desiderati si chiama *programma.*

I componenti essenziali di un calcolatore sono un *processore,* una *memoria* ed alcuni *dispositivi di input e di output.*

Il *processore* è il cavallo da tiro che esegue la successione di operazioni specificate dal programma. Le operazioni del processore sono molto elementari ma eseguite ad altissima velocità – oltre un milione di operazioni al secondo.

La *memoria* viene usata per conservare le informazioni alle quali sono applicate le operazioni del processore. La memora è di due tipi: *primaria* o *centrale,* e *secondaria* o *di massa.* La memoria centrale permette al processore di acquisire e memorizzare unità di informazione ad una velocità paragonabile alla sua velocità operativa: infatti ogni

operazione richiede normalmente un accesso alla memoria. Per permettere al processore di procedere da un'operazione alla successiva senza ritardo, anche la sequenza di istruzioni del programma che specifica queste operazioni è conservata nella memoria principale. Questa così contiene sia le *istruzioni* che i *dati* sui quali opera il processore.

La quantità di memoria centrale a disposizione del processore è limitata, ed è usata per conservare solo i programmi ed i dati sui quali il processore opera al momento e non si usa, invece, per la memorizzazione permanente dei dati. In alcuni casi la memoria principale può persino non essere abbastanza grande da contenere tutte le istruzioni ed i dati necessari per l'esecuzione di un programma, e perciò i calcolatori sono dotati anche di dispositivi di memoria secondaria come nastri magnetici, dischi o tamburi. Le caratteristiche essenziali di questi dispositivi sono:

(a) la loro capacità è normalmente molto maggiore di quella della memoria principale;
(b) le informazioni si possono conservare in modo permanente, per esempio da un'esecuzione di un programma ad un'altra.

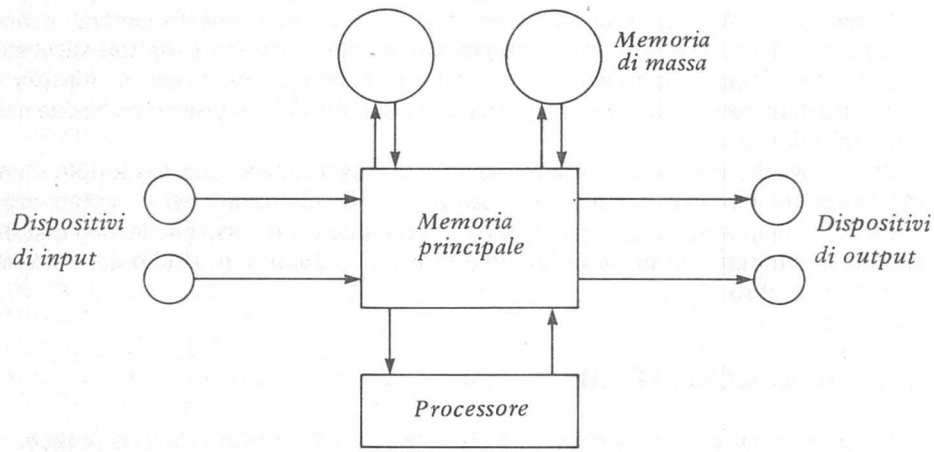


Fig. 1.1 Organizzazione di un calcolatore

Tuttavia la velocità alla quale si possono trasferire le informazioni tra unità centrale e memoria secondaria è molto minore di quella della memoria centrale.

I *dispositivi di input e di output* sono usati per trasferire informazioni dal mondo esterno alla memoria principale *(input)* e da questa a quello *(output)*. Dispositivi di input familiari sono i lettori di schede, che sono in grado di leggere e memorizzare nel calcolatore le informazioni perforate su schede, e le tastiere dei terminali che trasferiscono i caratteri corrispondenti ai tasti premuti nella memoria del calcolatore. Dispositivi di output comuni sono le stampanti e le telescriventi, che stampano le informazioni su tabulati continui, e gli schermi video che mostrano informazioni testuali o

grafiche sullo schermo di un tubo a raggi catodici.

L'organizzazione logica di un calcolatore si può schematizzare come in fig. 1.1.

1.2. STESURA DI UN PROGRAMMA

L'uso di un calcolatore per risolvere un particolare problema implica tre passi essenziali:

(a) specifica del problema, in termini dei dati di input da utilizzare e dei dati di output o risultati da produrre;
(b) progetto di un *algoritmo*, o sequenza di passi con i quali il calcolatore può produrre l'output richiesto dall'input a disposizione;
(c) codifica di tale algoritmo in un programma in un linguaggio di programmazione, come ad esempio il PASCAL.

Il passo (a), la specifica, non è normalmente considerato parte del processo di programmazione, ma una specifica precisa è un prerequisito essenziale di ogni programma ben fatto.

In passato è stata pratica comune separare i passi (b) e (c), definendo prima l'algoritmo in una notazione adatta al progetto, e poi traducendo e codificando tale algoritmo nel linguaggio scelto. Il PASCAL invece, fornisce una notazione che può essere usata sia per il progetto che per la codifica finale del programma richiesto. Usando il PASCAL, perciò, i passi (b) e (c) di solito non sono separati, ma fusi in un unico processo di progettazione/programmazione. Questa tecnica è ben illustrata dai programmi presentati nei capitoli seguenti.

In teoria, una volta scritto il programma, il programmatore ha completato il suo compito, dal momento che l'esecuzione di questo programma da parte del calcolatore dovrebbe produrre i risultati desiderati. In pratica, poiché il compito che il calcolatore deve eseguire è complesso e la capacità del programmatore è limitata, il primo programma scritto può non produrre i risultati richiesti. Il programmatore, perciò, comincia un ciclo di verifica e correzione del suo programma finché non si convince che esso è completamente coerente con le specifiche. Questo processo di ricerca e correzione di errori in un programma è noto come *debugging*. Il processo di debugging di solito viene fatto facendo girare il programma sul calcolatore con opportuni dati di test.

1.3. ESECUZIONE DI UN PROGRAMMA

In un linguaggio "ad alto livello" come il PASCAL il programma è espresso come sequenza di passi elementari di livello adeguato per il programmatore. Il programma viene inoltre preparato in una forma che può essere convenientemente realizzata dal programmatore – come testo scritto o stampato su carta, perforato su un pacco di schede, o battuto alla tastiera di un terminale.

Il programma che il processore del calcolatore esegue deve essere espresso come una sequenza di istruzioni, eseguibili dal processore, più semplici ed "a basso livello", e deve essere conservato nella memoria del calcolatore come sequenza di operazioni

codificate ciascuna delle quali sia immediatamente eseguibile dal processore. La preparazione di un programma in questa forma è estremamente noiosa e faticosa per un programmatore.

Fortunatamente, tuttavia, la traduzione di un testo espresso in un linguaggio ad alto livello in una successione equivalente di istruzioni eseguibili dal processore e contenute nella memoria è un lavoro di routine che può essere eseguito da un programma. Tale programma è messo a disposizione per ciascun linguaggio ad alto livello usabile su un calcolatore, e si chiama *compilatore* per quel linguaggio.

Così un programma scritto come testo in un linguaggio ad alto livello è dato prima come input all'esecuzione del programma compilatore di quel linguaggio. Il compilatore produce nella memoria del calcolatore un programma equivalente eseguibile, il quale può allora essere eseguito per ottenere il risultato desiderato. La fig. 1.2 mostra uno schema di questo processo a due stadi.

Nel tradurre un programma in un linguaggio ad alto livello il compilatore può trovare molti degli errori più elementari commessi nello scrivere il programma. Il compilatore riporta questi errori al programmatore producendo come output il *listato del programma* originale insieme con messaggi che permettono di identificare gli errori trovati durante la sua compilazione. Gli errori di programmazione trovati in questa maniera sono noti come *errori a tempo di compilazione.*

Durante l'esecuzione del programma eseguibile prodotto dal compilatore, possono essere individuate ulteriori situazioni non permesse dalle regole del linguaggio – tali errori sono noti come *errori a tempo di esecuzione.*

Un programma che non produce errori a tempo di compilazione né errori a tempo di esecuzione può ancora non produrre risultati corretti, a causa di errori nell'algoritmo prescelto o nella scrittura del programma. Tali errori sono noti come *errori logici.*

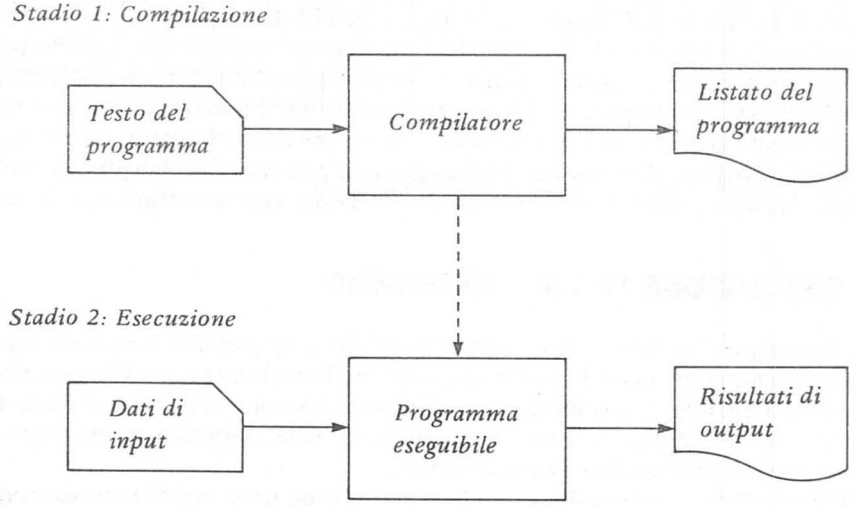


Fig. 1.2 Stadi di esecuzione di un programma

Il debugging, perció, richiede l'eliminazione di tutti gli errori di compilazione, di esecuzione e logici dal programma. A questo scopo il programmatore deve conoscere il linguaggio e *l'implementazione* che sta usando.

1.4. IMPLEMENTAZIONI DI UN LINGUAGGIO

Un vantaggio dell'uso di un linguaggio ad alto livello, come il PASCAL, è che un programma scritto nel linguaggio può essere usato su ogni calcolatore che ha disponibile un compilatore del linguaggio. In teoria il linguaggio, e perciò i programmi scritti in esso, sono indipendenti dal particolare calcolatore – sono perciò detti *indipendenti dalla macchina.*

In pratica esistono molti differenti calcolatori. La presenza di un compilatore di un dato linguaggio su un dato calcolatore è chiamata *implementazione* del linguaggio. Per ragioni pratiche le implementazioni debbono imporre al linguaggio restrizioni aggiuntive rispetto a quelle specificate dalla definizione del linguaggio. Analogamente talvolta le implementazioni richiedono una rappresentazione del testo di un programma leggermente diversa da quella descritta dalla definizione del linguaggio. Lo standard del PASCAL identifica tali aspetti del linguaggio come *definiti* o *dipendenti dall'implementazione,* e così sono chiamati in questo libro. Nel preparare o usare un programma su una data implementazione, il programmatore deve assicurarsi che il programma sia conforme alle regole di quella implementazione, così come alle regole del linguaggio.

Oltre alla capacità di compilare un programma, un'implementazione può fornire altri strumenti al programmatore. Per esempio verificare un errore a tempo di esecuzione o logico non necessariamente identifica la sua causa. Alcune implementazioni possono fornire al programmatore alcuni strumenti adatti all'uopo. Dato che queste caratteristiche sono del tutto dipendenti dall'implementazione, non sono discusse in questo libro, ma naturalmente il programmatore dovrebbe stabilire quali opzioni addizionali gli mette a disposizione l'implementazione alla quale ha accesso.

1.5. OBIETTIVI DELLA PROGRAMMAZIONE

Nel preparare un programma il programmatore deve spesso scegliere tra varie soluzioni alternative. Ogni scelta deve essere fatta in relazione agli obiettivi ed ai vincoli del particolare programma. In questo libro si assume che i seguenti obiettivi siano da perseguire nello sviluppo di qualsiasi programma.

1.5.1. Correttezza

Un obiettivo ovvio nella scrittura di qualsiasi programma è che esso dovrebbe seguire esattamente le specifiche date. Anche troppo spesso, tuttavia, a causa della complessità del problema e della comprensione e attenzione inadeguate dedicate dal programmatore, un programma non aderisce perfettamente alle specifiche. Il programmatore dovrebbe in ogni momento essere attento alla correttezza del programma e al-

la sua rispondenza allo scopo.

Un fattore chiave nel conseguire la correttezza è la *semplicità*. Se si sceglie l'algoritmo o la tecnica più semplice a disposizione è probabilmente più facile per un programmatore vedere se un programma è coerente o no con le specifiche, ed è più difficile inserire errori nella fase di scrittura. Nella programmazione è del tutto inutile ogni forma di complicazione.

Naturalmente alcuni programmi sono inerentemente complessi. In questi casi il programmatore deve adottare un metodo sistematico che controlli e limiti la complessità relativa ad ogni stadio. Questo metodo è illustrato nei programmi sviluppati nel libro.

1.5.2. Chiarezza

Un programma è necessariamente complesso quanto l'algoritmo che esso descrive. Tuttavia, è importante che il modo in cui l'algoritmo viene descritto dal programma non sia più complicato del necessario. La chiarezza del programma è uno strumento di valido aiuto sia per il programmatore stesso, durante il progetto e durante il debugging del programma, sia per coloro che possono dover leggere e correggere il programma in qualche fase successiva.

La chiarezza del programma si ottiene all'incirca allo stesso modo che per qualsiasi altro testo, come un saggio od un libro. Essa richiede:

(a) una divisione logica del testo in parti significative (capitoli, paragrafi, etc.) che rifletta le distinzioni tra le materie descritte, e la loro presentazione in una sequenza logica che ne rispecchi le relazioni reciproche;
(b) una scelta attenta dei costrutti usati in ciascuna parte per esprimere il più precisamente possibile il significato inteso;
(c) lo sfruttamento delle possibilità grafiche come le righe bianche e l'indentazione, per mettere in evidenza la relazione tra le parti componenti il testo.

Un programmatore dovrebbe essere tanto scrupoloso nell'uso di queste tecniche, quanto ogni altro autore di testi. In molti casi l'utilità di un programma è determinata tanto dalla chiarezza del suo testo quanto dalla qualità dell'algoritmo che descrive.

1.5.3. Efficienza

Il costo dell'esecuzione di un programma è normalmente misurato in termini

(a) del *tempo* occorrente al calcolatore per eseguire la sequenza di operazioni in oggetto;
(b) della quantità di *memoria* usata dal calcolatore in tale esecuzione.

In parecchi ambienti il programma dovrà competere con altri programmi per l'uso di queste risorse di calcolo, ed è perciò logico minimizzare le necessità di ciascuna di esse da parte del programma.

Il tempo impiegato per l'esecuzione del programma è direttamente proporzio-

nale al numero di operazioni che il processore deve eseguire. Il programmatore dovrebbe perciò scegliere un algoritmo che minimizzi le operazioni necessarie, ed aver cura di evitare, nella scrittura di un algoritmo in forma di programma, ogni operazione ridondante.

La memoria usata da un programma durante la sua esecuzione è determinata dalla quantità di dati che debbono essere conservati e dalla quantità di istruzioni di processore necessarie per definire il programma, dal momento che queste pure debbono essere mantenute in memoria. Per minimizzare la memoria usata dal suo programma un programmatore deve perciò stare attento sia ai dati manipolati che al numero di operazioni specificate dal programma.

Per certi programmi, o parti di essi, l'uso efficiente di tempo o memoria può essere critico, mentre per altre assai meno. Il programmatore deve essere cosciente di tali necessità di efficienza nella fase di scrittura del programma. In questo libro sono indicati i casi in cui esiste una differenza nell'efficienza di costrutti PASCAL alternativi, cosicché un programmatore possa fare una scelta più corretta in relazione alle necessità del suo programma.

2.
Notazioni e Concetti Fondamentali

2.1. FORMA DI BACKUS-NAUR ESTESA

Un programma è costruito, in prima approssimazione, come una sequenza di simboli o caratteri che formano il "testo". (1) è un semplice programma scritto nel linguaggio di programmazione PASCAL.

```
program addizione (input, output);
var primo, secondo, somma: integer;
begin
    read (primo, secondo);                                    (1)
    somma : = primo + secondo;
    write (somma)
end.
```

Come per i linguaggi naturali, ogni linguaggio di programmazione ha associato un insieme di regole rigidamente definito che descrive come costruire un programma valido per il linguaggio. Queste regole servono in primo luogo per garantire al programmatore la correttezza e l'effetto del suo programma, e poi per permetterne la comprensione da parte del calcolatore e di chiunque lo legga.

Le regole del linguaggio sono costituite da due parti, note come *sintassi* e *semantica*. Le regole sintattiche definiscono il modo in cui le parole (o *vocabolario*) del linguaggio si possono combinare per formare "frasi". Le regole semantiche attribuiscono il significato a queste combinazioni di parole. Di solito le regole semantiche sono stabilite in maniera meno formale rispetto alle regole sintattiche. Queste ultime, per il linguaggio PASCAL, si possono descrivere per mezzo di un formalismo noto col nome di *forma di Backus-Naur estesa*, o semplicemente EBNF (Extended Backus-Naur Form).

Ad esempio, la forma di un programma PASCAL è definita dalla seguente regola EBNF:

programma = intestazione-programma blocco ".".

Questa regola va letta: "Un *programma* è definito da una *intestazione-programma*, se-

guita da un *blocco* seguita da un punto "."". Poi altre regole definiscono le forme consentite di una *intestazione-programma* e di un *blocco*. Nell'esempio precedente di programma la prima riga di simboli è la *intestazione-programma*, mentre i simboli successivi da **var** a **end** formano un *blocco*. In una regola EBNF l'occorrenza di un simbolo del linguaggio tra due coppie di apici, per esempio "." , indica il simbolo stesso. Ogni regola termina con un punto.

Una categoria sintattica può avere talvolta parecchie forme alternative. Ad esempio la regola

cifra = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

definisce una *cifra* in PASCAL come uno dei dieci caratteri 0,1,..,9 con una lista di alternative. La regola sopraindicata si legge: "Una *cifra* è definita da uno "0" o da un "1" o da un "2" o ....".

La chiusura di un costrutto sintattico tra parentesi graffe { } è usata per indicare la ripetizione del costrutto tra parentesi zero o più volte. Ad esempio la regola

intero-senza-segno = *cifra* { cifra }.

definisce un *intero-senza-segno* come una sequenza di una o più delle cifre sopradefinite. Così i seguenti sono esempi di *interi-senza-segno*:

1 27 97300

La chiusura di un costrutto sintattico tra parentesi quadre [ ] indica che il costrutto tra parentesi è opzionale, per esempio le regole.

intero = [*segno*] *intero-senza-segno*.

segno = "+" | "−".

definiscono un *intero* come un *intero-senza-segno* preceduto, eventualmente, da un simbolo "+" o da un simbolo "−".

Le parentesi tonde ( ) si possono usare, quando è necessario, per raggruppare categorie sintattiche; per esempio, le regole per gli *interi* si possono riesprimere nel seguente modo

intero = *intero-con-segno* | *intero-senza-segno*.
intero-con-segno = ("+" | "−") *intero-senza-segno*.

In generale le categorie sintattiche definite dalle regole EBNF si indicano con parole singole (eventualmente collegate da trattini) che ne diano una conveniente descrizione in italiano.

Nel corso del libro introdurremo, insieme con le nuove caratteristiche del linguaggio, le loro definizioni EBNF. Queste definiscono in modo conciso l'intero insieme della costruzioni di ogni categoria sintattica. Alcune costruzioni possono includere altre categorie sintattiche non ancora completamente definite al momento della loro introduzione. Lo studente che impara il linguaggio può ignorare le categorie non definite poiché incontrerà nel seguito la loro trattazione. L'indice delle definizioni sintattiche che si trova alla fine del libro permette di trovare la definizione e la spiegazione di tutte le categorie sintattiche. Qualche volta la sintassi del PASCAL è definita per

mezzo di *diagrammi sintattici.* Questi specificano graficamente in modo conciso le sequenze di simboli permesse in ciascuno dei principali costrutti del PASCAL, e sono utili a coloro che hanno familiarità con il linguaggio per controllare la validità o la forma consentita di particolari costrutti.

L'appendice 1 contiene una descrizione della sintassi del PASCAL in questa forma.

2.2. IL VOCABOLARIO DEL PASCAL

Qualsiasi linguaggio, sia esso parlato o di programmazione, fa uso di un vocabolario. L'italiano, per esempio, nella sua forma scritta, consiste di parole, numeri e simboli di punteggiatura. Il vocabolario del linguaggio di programmazione PASCAL consiste di lettere, cifre e simboli speciali. Le frasi del linguaggio sono quindi costruite a partire da questo vocabolario in accordo con la sintassi del PASCAL.

Secondo la definizione standard del PASCAL, una lettera può essere una delle 26 lettere dell'alfabeto anglosassone in maiuscolo o in minuscolo, cioè ci sono 52 alternative nella categoria sintattica *lettera.*

lettera = *"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|*
"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|
"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|
"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".

In pratica, però, alcuni dei calcolatori su cui è stato implementato il PASCAL permettono l'uso delle sole lettere maiuscole, nel qual caso il programmatore non può usare le minuscole. In questo libro useremo liberamente queste ultime semplicemente per chiarezza tipografica e per una più facile lettura del testo. I programmi risultanti si possono rendere compatibili con le implementazioni del PASCAL che accettano le sole lettere maiuscole, sostituendo tutte le lettere minuscole con le corrispondenti lettere maiuscole. Le versioni finali dei programmi presi in esame nei capitoli successivi sono state riprodotte con metodi fotografici dai listati stampati con il calcolatore, che mostrano la rappresentazione in lettere maiuscole.

Una *cifra* in PASCAL è una delle dieci cifre arabe, cioè

cifra = *"0"| "1"| "2"| "3"| "4"| "5"| "6"| "7"| "8"| "9"* .

Si noti che la lettera *"O"* e la cifra *"0"* sono due caratteri del tutto diversi.

Il numero di simboli speciali richiesti dal PASCAL per la punteggiatura e per vari altri scopi è piuttosto elevato. Molti di questi simboli speciali sono rappresentati da parole invece che da caratteri speciali, come mostrato in (2).

simbolo-speciale = *"+"| "−"| "*"| "/"| "="| "<>"| "<"| ">"|*
"<="| ">="| "("| ")"| "["| "]"| ":="| "."| "'"|
","| ":"| ";"| ".."| "↑"| **"div"** | **"mod"**| **"nil"**| **"in"**|
"or"| **"and"**| **"not"**| **"if"**| **"then"**| **"else"**|

"case"| "of"| "repeat"| "until"| "while"| "do"| (2)
"for"| "to"| "downto"| "begin"| "end"| "with"|
"goto"| "const"| "var"| "type"| "array"| "record"|
"set"| "file"| "function"| "procedure"| "label"|
"packed"| "program" .

In questo libro le parole usate per rappresentare i simboli speciali vengono evidenziate per mezzo dei caratteri in grassetto, come **questi.** Comunque la definizione del PASCAL stabilisce che queste sono parole riservate e non possono essere usate per nessun altro scopo, perciò non è necessario scriverle con sottolineature o altri segni distintivi. I listati dei programmi, riprodotti con metodi fotografici, illustrano questa forma più comune.

2.3. NUMERI

I numeri in PASCAL si possono rappresentare in due forme–interi o reali.

Un *numero-intero* è un numero intero che può essere positivo, negativo o zero. Il numero è scritto come una sequenza di cifre di qualsiasi lunghezza, e può eventualmente essere preceduto da un segno (+ o −).

numero-intero = [*segno*] *intero-senza-segno.*
segno = "+" | "−" .
intero-senza-segno = *cifra* { *cifra* } .

Esempi di interi validi sono:

6
0
−6
+7000000

I seguenti non sono interi validi per la ragione specificata a fianco:

6,437,271	un intero non può contenere elementi che non siano cifre,
−6.0	un intero non può contenere un punto decimale.

Sebbene la precedente definizione permetta interi di qualsiasi dimensione, in pratica ogni implementazione del PASCAL specifica un intervallo di interi consentiti. Ciò è necessario perché il massimo numero di cifre che si può usare come intero dipende dalle dimensioni della locazione di memoria del calcolatore in cui viene memorizzato l'intero.

I numeri al di fuori dell'intervallo di interi consentito, o i numeri con parte frazionaria si possono rappresentare come *reali.* Un numero reale può essere scritto in due diverse forme. Nella prima il numero si scrive con un punto decimale e può essere preceduto da un segno. In ogni caso, il punto decimale deve essere preceduto da almeno

una cifra e seguito da almeno una cifra.

Esempi di numeri reali validi sono:

```
  0.0
  0.873
-74.1
 73.36789
```

I seguenti non sono numeri reali validi per le ragioni indicate:

0.	nessuna cifra dopo il punto decimale,
.736	nessuna cifra prima del punto decimale,
2,736.45	la virgola non è permessa.

Nella seconda forma di rappresentazione il numero reale è espresso da un numero intero o da un numero decimale moltiplicato per una potenza intera di 10 (questa forma è nota col nome di forma scientifica o esponenziale). Il numero intero o decimale è seguito immediatamente dalla lettera **E** (maiuscola o minuscola) ed un intero con o senza segno (detto esponente o *fattore-scala*).

Esempi di numeri reali validi sono:

```
  0E0
  8.73E+02
 -741E-1
0.7336789E2
```

Modi accettabili di scrivere il numero reale 253.0 sono i seguenti:

```
253.0    2.53E2    25.3E+01    253E0    2530E-1
```

La definizione sintattica di un numero reale è indicata in (3).

```
numero-reale = [ segno ] reale-senza-segno.
reale-senza-segno = intero-senza-segno "." sequenza-cifre
                    [ "E" fattore-scala ] | intero-senza-segno          (3)
                    "E" fattore-scala.
sequenza-cifre = cifra { cifra } .
fattore-scala = intero.
```

L'intervallo dei numeri reali consentito dalle implementazioni è anch'esso limitato, ma di solito è molto più grande di quello degli interi. Comunque, la rappresentazione dei numeri reali in un calcolatore è soltanto una approssimazione mentre gli interi sono rappresentati in modo esatto. Perció, quando è possibile, un numero si dovrebbe rappresentare come intero per ottenere la massima precisione nelle operazioni che lo coinvolgono. Una discussione più completa sui numeri reali viene fatta nel Cap. 3.

2.4. IDENTIFICATORI

Il PASCAL richiede che i vari oggetti usati in un programma, come i dati e certe parti di testo del programma, abbiano dei nomi per mezzo dei quali possano essere identificati. Questi nomi si chiamano *identificatori* e vengono creati dal programmatore.

Il PASCAL consente un notevole grado di flessibilità nella scelta degli identificatori. Un identificatore consiste di una lettera seguita da un numero qualsiasi di lettere e cifre, cioè

identificatore = *lettera* { *lettera* | *cifra* } .

Perció, in teoria, un identificatore può essere di lunghezza qualsiasi (noi incoraggeremo l'uso di identificatori significativi e, perció, in alcuni casi lunghi) ma, in pratica, molte implementazioni impongono una restrizione sulla lunghezza delle righe di testo del programma e quindi sulla lunghezza massima degli identificatori. Se l'implementazione consente l'uso di lettere maiuscole e minuscole, entrambe possono essere usate negli identificatori, ma il PASCAL standard richiede che il significato del programma non venga alterato cambiando il formato delle lettere. I seguenti identificatori vengono perció considerati identici:

nome *NOME* *Nome*

Esempi di identificatori validi sono

I
ufo
PC49
unnomedavverolungo

I seguenti sono identificatori non validi per le ragioni indicate

1abc	un identificatore deve iniziare con una lettera,
dopo-domani	un identificatore non deve contenere trattini,
$100	un identificatore deve iniziare con una lettera.

Come già osservato, alcuni dei simboli speciali del PASCAL sono parole riservate che non possono essere usate per altri scopi. Perciò parole come **array** e **begin** non si possono usare come identificatori.

Sebbene il programmatore abbia una scelta illimitata per i nomi degli identificatori, è pratica comunemente accettata scegliere identificatori il cui significato nel programma è suggerito dal loro nome. Se in un programma si utilizzano valori per descrivere temperature e pressioni è molto più indicativo per chi legge il programma rappresentare tali valori con identificatori come *temperatura* e *pressione* piuttosto che con *t* e *p*, che sono più brevi ma di significato meno ovvio. Un identificatore, per quanto il nome scelto per esso possa essere significativo, non determina la quantità che identifica, per quanto riguarda le regole del linguaggio. Questa informazione deve essere fornita per mezzo di una *dichiarazione*. Le forme consentite per le dichiarazioni degli identificatori verranno spiegate al momento della descrizione dei vari oggetti che un identificatore può denotare in PASCAL.

abs	*ln*	*reset*
arctan	*maxint*	*rewrite*
boolean	*new*	*round*
char	*odd*	*sin*
chr	*ord*	*sqr*
cos	*output*	*sqrt*
dispose	*pack*	*succ*
eof	*page*	*text*
eoln	*pred*	*true*
exp	*put*	*trunc*
false	*read*	*unpack*
get	*readln*	*write*
input	*real*	*writeln*
integer		

Tabella 2.1 Identificatori standard

Alcuni identificatori, detti *identificatori standard*, sono predichiarati in ogni implementazione del PASCAL standard. Essi descrivono quantità standard e funzionalità fornite dal linguaggio, come le funzioni trigonometriche ed aritmetiche. La tabella 2.1 mostra una lista completa degli identificatori standard. A differenza delle parole riservate, che non si possono assolutamente usare come identificatori il programmatore può riutilizzare qualunque identificatore standard per descrivere un oggetto che egli desidera introdurre nel suo programma. Ad ogni modo, nel far questo, egli si priva della possibilità di usare la funzionalità del PASCAL standard denotata da quell'identificatore e crea una potenziale confusione per chi successivamente leggerà il programma. Perciò si sconsiglia l'uso di questa pratica.

2.5. STRINGHE

Una sequenza di caratteri racchiusi tra apici forma una stringa. Questa si usa all'interno di un programma per denotare la sequenza di caratteri stessa. Se la stringa di caratteri contiene un apice quest'ultimo va scritto due volte. Quindi la definizione sintattica di una stringa è

stringa = "'" *carattere-stringa* { *carattere-stringa* }"'" .
carattere-stringa = *caratteri-escluso-apice* | "''".

Esempi di stringhe sono

'Esempio di stringa.'
'?'
'Il suo nome e'' Paolo Rossi'
''''

2.6. COMMENTI

Un programma PASCAL è espresso come una sequenza di identificatori, numeri, stringhe e simboli speciali in accordo con la descrizione fornita nei paragrafi precedenti. Nella stesura del testo di un programma il programmatore gode di una notevole libertà nel posizionamento di questi identificatori, numeri e simboli all'interno delle righe del testo e tra esse.

In generale tra gli identificatori, i numeri ed i simboli si possono inserire un numero arbitrario di spazi o di caratteri di fine linea. Comunque gli spazi o i caratteri di fine linea non si possono inserire all'interno di un identificatore, di un numero o di un simbolo speciale ed almeno uno spazio o un carattere di fine linea deve essere inserito tra gli identificatori, i numeri e le parole riservate che altrimenti risulterebbero indistinguibili.

Negli esempi di programma presentati nel libro cercheremo di mostrare che un'attenta impaginazione del testo del programma ed una accurata scelta degli identificatori contribuiscono a rendere un programma autoesplicativo. Ad ogni modo, rimane necessaria e desiderabile per il programmatore la possibilità di aggiungere del testo esplicativo del programma, o *commento*, per chiarire a coloro che lo leggeranno in seguito le azioni da esso realizzate.

Nel PASCAL standard qualsiasi sequenza di caratteri racchiusa tra parentesi graffe { } e non contenente una parentesi graffa chiusa } , forma un *commento*. I commenti possono apparire in un qualsiasi punto del programma dove è permesso uno spazio o un carattere di fine riga, ma non hanno significato alcuno per quanto riguarda la sua esecuzione. Il loro unico scopo è di permettere al programmatore di esprimere più chiaramente il significato del suo programma con l'inserimento di note esplicative in linguaggio naturale.

esempio: { questo è un commento scritto in italiano }

Non vi è alcuna limitazione sulla lunghezza di un commento–esso può consistere di pochi caratteri come di parecchie centinaia di righe di testo.

2.7. SIMBOLI ALTERNATIVI

In molti calcolatori i simboli di parentesi graffa { } non si possono usare, nel qual caso il PASCAL consente l'uso di (* invece di { e di *) invece di } come convenzione alternativa per i commenti. L'uso di tale convenzione è illustrato dai listati dei programmi riprodotti fotograficamente nei capitoli successivi.

In alcuni calcolatori non si possono usare anche altri simboli, usati invece nella rappresentazione standard del PASCAL. Per superare questo problema, le seguenti alternative, in sostituzione dei simboli standard, sono possibili su tutte le implementazioni il cui insieme di caratteri comprende quelli considerati.

SIMBOLO	*ALTERNATIVA*
↑	^ oppure @
{	(*
}	*)
[	(.
]	.)

In molti casi le implementazioni potranno perciò accettare più di una rappresentazione di un simbolo, permettendo programmi che usano rappresentazioni miste. In generale, comunque, è consigliabile usare soltanto la rappresentazione standard dei simboli se questa è permessa ed evitare sempre di mescolare diverse rappresentazioni dello stesso simbolo.

2.8. STRUTTURA BASE DEL PROGRAMMA

Concludiamo questo capitolo con un rapido sguardo alla struttura base dei programmi PASCAL in modo tale che il lettore sia in grado di valutare il contesto ed il significato dei concetti che verranno introdotti nei prossimi capitoli.

Un programma deve descrivere la natura dei dati o delle informazioni da manipolare e le azioni da compiere su quei dati. Questi due aspetti della funzione di un programma si riflettono nella sintassi della maggior parte dei linguaggi di programmazione. In PASCAL un programma è formalmente definito come mostrato in (4).

```
programma = intestazione-programma blocco "." .
intestazione-programma = "program" identificatore "(" lista-identificatori ")" ";".
blocco = parte-dichiarazioni parte-istruzioni.
parte-dichiarazioni = [parte-dichiarazione-etichette]
                      [parte-definizione-costanti]
                      [parte-definizione-tipi]                    (4)
                      [parte-dichiarazione-variabili]
                      parte-dichiarazione-procedure-e-funzioni.
lista-identificatori = identificatore { "," identificatore }.
```

Cioè un *programma* consiste di una *intestazione-programma* seguita da un *blocco* ed è concluso da un punto; un *blocco consiste di una parte-dichiarazioni* seguita da una *parte-istruzioni*. Il ruolo preciso delle varie componenti della *parte-dichiarazioni* verrà spiegato nei capitoli successivi. La *parte dichiarazioni* di un programma nella sua forma più semplice definisce i dati utilizzati dal programma. La *parte-istruzioni* definisce poi le azioni che devono essere eseguite su tali dati.

Consideriamo di nuovo il semplice programma introdotto all'inizio di questo capitolo, riscritto per comodità in (5). Il suo scopo è di leggere due numeri interi e stampare la loro somma. La prima riga è la *intestazione-programma* che attribuisce il nome *addizione* al programma e indica che esso realizzerà un'operazione di input (lettura di informazione da un supporto esterno al programma) e un'operazione di output (scrittura di informazione su un supporto esterno al programma). La seconda riga è la *parte-dichiarazioni* che attribuisce un nome ai tre dati usati nel programma e stabilisce che saranno numeri interi.

```
program addizione (input, output);
var primo, secondo, somma : integer;
begin                                                             (5)
    read (primo, secondo);
    somma := primo+secondo;
    WRITE (SOMMA)
end
```

```
    write (somma)
end.
```

La *parte-istruzioni* del programma è costituita dalla descrizione di tre azioni racchiuse dalla coppia di parole riservate **begin** ed **end**. In primo luogo i due numeri interi vengono letti ed i loro valori registrati come dati *primo* e *secondo*. Questi due valori vengono poi sommati e registrati come dato *somma*. Finalmente, viene scritta in output la somma. Il testo del programma termina con un punto.

ESERCIZI

2.1 Si considerino i seguenti numeri:

275	3,475	7.4	.1475	6000
6E3	275.0	0.001	27365982	0
10E-4	0.074E3	0.1E999	275.	0.0620

Quali sono interi validi in PASCAL?
Quali sono numeri reali validi in PASCAL?
Quali denotano lo stesso valore in PASCAL?
Quali tra quelli che sono validi in PASCAL lo sono anche per l'implementazione alla quale avete accesso?

2.2 Quali dei seguenti identificatori sono validi in PASCAL?

HS	*RAGGI-X*	*alphabetic*	*ALPHABETIC*
ALPHA	*DiGiorgio*	*BEGIN*	*ALPHABETICAL*
omega	*X99999*	*round*	*INPUT*

Quali tra gli identificatori validi lo sono anche per la vostra implementazione?

2.3 Quali delle seguenti righe sono stringhe valide in PASCAL?

'GINO BIANCHI'
'sei + uno = sedici'
'LEI E '' LA MOGLIE DI BIANCHI'
'IL CANE DI SUA MOGLIE'
''''''''''''

2.4 Una sequenza valida di simboli PASCAL è la seguente:

```
if input ↑ = ' '
    then { salta lo spazio } get (input)
    else { conta la sua occorrenza } c [ input ↑ ] := c [ input ↑] + 1
```

Può essere scritta in questa forma per la vostra implementazione?
Se non lo è quali cambiamenti sono necessari?

3.
Tipi di dato e dichiarazioni

3.1. TIPI DI DATO

Un programma manipola informazioni, o *dati,* per ottenere un certo effetto o risultato desiderato. I dati su cui opera un programma sono rappresentati nell'hardware del calcolatore come una sequenza di segnali elettrici. Tuttavia, i linguaggi di alto livello come il PASCAL permettono al programmatore di ignorare la rappresentazione fisica usata dalla macchina, e di esprimere la natura dei dati in termini di *tipi di dato.*

Un tipo di dato definisce un insieme di valori. Il PASCAL, come molti altri linguaggi di alto livello, richiede che ogni dato di un programma abbia un *tipo* associato ad esso, in base alle seguenti considerazioni:

(a) il tipo di un dato determina il *campo dei valori* che esso può assumere, e le specifiche *operazioni* che si possono applicare su di esso;
(b) ogni dato è di un solo tipo;
(c) il tipo di un dato in un programma può essere dedotto esclusivamente dalla sua forma o dal suo contesto, senza alcuna conoscenza dei valori particolari che può assumere durante l'esecuzione;
(d) ogni *operatore* del linguaggio richiede *operandi* di tipo specifico, e produce un risultato di tipo specifico.

Il significato e l'effetto di queste regole sarà descritto e illustrato nei prossimi due capitoli.

Fornendo un insieme di tipi predefiniti, con gli appropriati operatori, il linguaggio permette al programmatore di descrivere le manipolazioni sui suoi dati in termini di questi, piuttosto che della sottostante rappresentazione della macchina. Inoltre, in base ai quattro vincoli di cui sopra, il linguaggio protegge il programmatore da combinazioni illogiche di dati ed operatori, protezione questa che non è presente al livello di macchina.

Nei capitoli successivi vedremo che il PASCAL permette la definizione di tipi altamente strutturati e complessi; in realtà è questa la caratteristica del PASCAL che lo pone in una classe particolare rispetto ai più comuni linguaggi di programmazione.

D'altra parte, ogni tipo di dato è composto in ultima analisi di tipi *non strutturati.* In PASCAL un tipo non strutturato o è definito dal programmatore, oppure è uno dei quattro tipi standard predefiniti (chiamati talvolta tipi *primitivi* del linguaggio), e cioè i tipi *integer, real, char* e *boolean.*

Cominciamo il nostro studio dei tipi di dato non strutturati osservando le proprietà di ciascuno dei quattro tipi primitivi.

3.1.1. Il tipo *integer*

Questo tipo rappresenta l'insieme dei numeri interi, e qualsiasi valore di questo tipo è dunque un numero intero. Tuttavia, come abbiamo visto nel Cap. 2, tutti i calcolatori impongono un limite sulla dimensione del massimo numero intero che possono memorizzare convenientemente, e così ogni implementazione del PASCAL limita i valori interi ad un sottoinsieme dei numeri interi. Così il tipo integer, in pratica, è l'insieme dei numeri interi definiti da una data implementazione, e un valore di tipo integer è uno dei numeri interi nel sottoinsieme definito dall'implementazione. La rappresentazione dei numeri interi è stata descritta nel capitolo precedente.

Il PASCAL definisce una serie di operatori aritmetici che prendono operandi interi e ritornano risultati interi; per esempio,

+	addizione
−	sottrazione
*	moltiplicazione
div	divisione con troncamento
mod	modulo

Questi sono *operatori binari infissi,* cioè sono usati con due operandi scritti ciascuno su un lato dell'operatore: *a* + *b*. Tuttavia, + e − possono anche essere usati come *operatori unari prefissi:* + *a* e − *a*, per denotare identità e inversione (negazione) di segno rispettivamente. Alcuni esempi di operazioni su interi sono riportati nella Tabella 3.1.

Operazione	Risultato
7+3	10
7−3	4
7*3	21
7 **div** 3	2
7 **mod** 3	1
−3	−3
+3	3

Tabella 3.1 Alcuni esempi di operazioni su interi.

Quando gli operandi *a* e *b* hanno lo stesso segno l'operazione *a* **div** *b* dà i normali risultati della divisione tra interi: per esempio

7 **div** 3 = 2
(−7) **div** (−3) = 2

Quando gli operandi sono di segno diverso, e perciò il vero quoziente è negativo, si "tronca" verso lo zero, cosicché

(−7) **div** 3 = −2
7 **div** (−3) = −2

Se *b* è zero si ha un errore.

L'operazione modulo (*a* **mod** *b*) è definita solo per *b* positivo; se *b* è zero o negativo il risultato è un errore. Per *b* positivo il valore di *a* **mod** *b* è definito come il minimo intero non negativo che può essere sottratto ad *a* per ottenere un multiplo intero di *b*. Così

6 **mod** 3 = 0
7 **mod** 3 = 1
(−6) **mod** 3 = 0
(−7) **mod** 3 = 2

Si noti che, per *a* e *b* positivi, il valore di *a* **mod** *b* è il resto della divisione intera *a* **div** *b*, ma per *a* negativo questo non è vero.

Occorre assicurarsi che il risultato delle operazioni di addizione, sottrazione e moltiplicazione di due valori non esca dall'insieme degli interi definito dall'implementazione.

Se questo succede si ha un errore detto di *overflow*. Analogamente la divisione per zero costituisce errore, così come un'operazione di modulo il cui secondo operando sia nullo o negativo. La maggior parte delle implementazioni del PASCAL fornisce qualche strumento mediante il quale durante l'esecuzione di un programma si possono individuare questi errori.

Oltre agli operatori infissi e prefissi di cui sopra, il PASCAL fornisce una serie di *funzioni standard* applicabili a valori interi.

Due di queste sono:

abs — se x è un valore intero, allora la funzione standard *abs (x)* denota il valore assoluto intero di *x*

sqr — il quadrato di un valore intero *x* può essere scritto come *sqr (x)*

esempio:

abs (7) dà 7
abs (−6) dà 6
sqr (3) dà 9
sqr (−4) dà 16.

Il quadrato di un valore intero *x* può chiaramente essere scritto come *x* * *x*. L'uso di *sqr (x)* migliora comunque la chiarezza del programma, e può anche migliorare la sua

efficienza, in certe implementazioni.

Altre funzioni standard relative agli interi ed ad altri tipi di dato sono introdotte nei paragrafi seguenti.

Così come gli altri tipi non strutturati (diversi da *real*), in PASCAL il tipo *integer* definisce una successione ordinata di valori – da un minimo ad un massimo, entrambi definiti dalla particolare implementazione del linguaggio. Ciascun valore, eccetto il minimo, ha un *predecessore,* e ciascuno eccetto il massimo ha un *successore.* Tali tipi sono chiamati tipi *ordinali.* Per gli interi il successore di qualsiasi valore si ottiene aggiungendo 1 e il predecessore sottraendo 1. Per tutti i tipi ordinali il PASCAL fornisce le funzioni standard *succ* e *pred,* definite come segue

succ (x) dà il successore immediato di *x,* se esiste
pred (x) dà il predecessore immediato di *x,* se esiste.

Così, per un intero *x,*

$succ\ (x) = x + 1$
$pred\ (x) = x - 1$

NB

3.1.2. Il tipo *real*

I valori di questo tipo costituiscono l'insieme dei numeri reali. Ancora una volta, c'è però un limite al massimo numero reale che un calcolatore può convenientemente rappresentare e così, in ciascuna implementazione del PASCAL, il tipo *real* consiste dei valori reali di qualche insieme definito dall'implementazione. La rappresentazione dei reali è stata descritta nel capitolo precedente.

Il PASCAL fornisce una serie di operatori aritmetici che prendono operandi reali e producono valori reali:

\+ addizione
– sottrazione
* moltiplicazione
/ divisione.

OPERATORI

Come nel caso degli interi questi sono operatori binari infissi, però + e – possono anche essere usati per denotare identità ed inversione di segno. Alcuni esempi di operazioni su reali sono riportati in Tabella 3.2.

Operazione	Risultato
2.1+1.4	3.5
2.1−1.4	0.7
2.1*1.4	2.94
2.1/1.4	1.5

Tabella 3.2 Esempi di operazioni su reali

Va fatto notare che la rappresentazione dei numeri reali all'interno di un calcolatore non è esatta, e che l'esecuzione di operazioni su questi valori approssimati può produrre imprecisioni sempre maggiori. Lo studio della stima e dell'effetto di tali imprecisioni viene affrontato dall'analisi numerica, e non sarà discusso ulteriormente qui. Il lettore deve comunque tenere presente che, per esempio, per valori reali x e y la relazione

$$(x/y) * y = x$$

può non valere sempre nell'aritmetica del calcolatore.

Le funzioni standard *abs* e *sqr* introdotte nel sottoparagrafo precedente per argomenti interi possono essere usate anche con argomenti reali per produrre risultati reali; per esempio

abs (−6.4) produce 6.4
sqr (3.1) produce 9.61

È da notare che *real* non è un tipo ordinale, cosicché le funzioni standard *succ* e *pred* non sono definite per argomenti reali.

Il PASCAL fornisce anche sei funzioni matematiche standard, che accettano e restituiscono tutte argomenti reali. Sono le seguenti:

ln (x)	restituisce il logaritmo naturale di x, cioè $\log_e x$, per $x > 0.0$.
exp (x)	restituisce l'esponenziale di x, cioè e^x.
sqrt (x)	restituisce la radice quadrata di x, per $x \geqslant 0.0$
sin (x)	restituiscono le funzioni trigonometriche seno e coseno di x, dove x è espresso in radianti.
cos (x)	
arctan (x)	restituisce l'angolo in radianti nell'intervallo 0-π, la cui tangente è x.

Il risultato dell'uso di *ln* con argomento non positivo, o di *sqrt* con argomento negativo, provoca un errore.

I valori di tipo *integer* si possono pensare come aventi valori equivalenti di tipo *real*. Come regola generale, in PASCAL un valore intero può essere usato ovunque è usabile un valore reale.

Così gli operatori binari +, − e * possono essere usati con un operando intero ed un operando reale, per produrre un risultato reale. Analogamente, le funzioni matematiche standard possono essere usate con argomenti interi ottenendo risultati reali; per esempio:

4.72*2	dà 9.4
4 − 1.5	dà 2.5
sqrt (9)	dà 3.0

Naturalmente se entrambi gli operandi degli operatori +, −, * sono interi il risultato è intero, come visto sopra. Si noti comunque che l'operatore reale di divisione, /, produce sempre un risultato reale, anche se entrambi gli operandi sono interi:

13/5 dà 2.6

In realtà, per le regole del PASCAL, ogni valore intero trovato quando è atteso un reale è convertito automaticamente al valore reale equivalente. L'inverso non è mai vero – se è atteso un valore intero al suo posto non può essere usato un reale.

In PASCAL esistono due funzioni standard per la conversione di valori reali in interi. (Funzioni che convertono un valore di un tipo in un valore di un altro tipo sono dette *funzioni di trasferimento).* Queste sono le funzioni di *troncamento* e di *arrotondamento,* definite come segue:

trunc (x) ha argomento reale e ne produce la parte intera.

Così

trunc (7.1) dà 7
trunc (−7.1) dà −7

round(*x*) ha argomento reale e produce l'intero più vicino.

Per *x* non negativo, *round (x)* è equivalente a *trunc* (*x* + 0.5), mentre per *x* negativo, è equivalente a *trunc* (*x* − 0.5); per esempio,

round (7.6) dà 8
round (7.1) dà 7
round (−7.6) dà −8
round (−7.1) dà −7

Nell'uso delle funzioni *trunc* e *round* occorre fare attenzione che il risultato desiderato non trabocchi oltre l'intervallo permesso degli interi, altrimenti si ha un errore. Sebbene l'intervallo dei reali ammessi sia molto più grande di quello ammesso per gli interi, va ricordato che un limite esiste, e che può verificarsi comunque un overflow, cioè un tentativo di calcolare un valore reale fuori limite. Le implementazioni di solito forniscono degli strumenti che segnalano tale condizione di overflow durante l'esecuzione di operazioni reali.

3.1.3. Il tipo *char*

Ogni sistema di elaborazione ha associato un insieme di caratteri, per mezzo dei quali comunica col suo ambiente; questi caratteri, cioè, sono di solito disponibili sui dispositivi di input ed output (come i lettori o le stampanti), e usati per trasferire informazioni in forma leggibile al e dal calcolatore. Sfortunatamente la maggior parte dei calcolatori forniscono insiemi di caratteri leggermente diversi, sebbene in generale rendano comunque disponibile un sottoinsieme standard comprendente le 26 lettere, le 10 cifre, il carattere di spazio ed altri simboli di punteggiatura.

In PASCAL il tipo *char* è definito come l'insieme dei caratteri disponibili nel sistema di elaborazione che esegue il programma. La documentazione relativa ad una particolare implementazione del PASCAL normalmente specifica l'insieme dei caratteri disponibili.

In un programma PASCAL un particolare valore di tipo *char* è denotato racchiudendo il carattere tra apici; per esempio,

```
'x'  '4'  '?'  ' '  '+'
```

Per rappresentare il carattere apice, questo si scrive due volte:

```
''''
```

Così i valori di tipo *char* sono denotati da stringhe contenenti un solo carattere (o due apici). Le stringhe sono state introdotte nel Cap. 2.

Il PASCAL fornisce due funzioni di trasferimento standard, che associano all'insieme dei caratteri un insieme di valori interi consecutivi non negativi, a partire dallo zero (detti numeri ordinali dell'insieme di caratteri), e viceversa.

ord (c)	è il numero ordinale del carattere *c* nell'insieme dei caratteri
chr (i)	è il carattere che ha numero ordinale i, se esiste; altrimenti si verifica un errore.

Ovviamente *ord* e *chr* sono funzioni inverse, e

```
chr (ord (c)) = c
ord (chr (i)) = i
```

sarà sempre vero.

L'associazione tra caratteri ed ordinali è definita dall'implementazione. Lo standard del PASCAL richiede che, in tutte le implementazioni, ai caratteri denotanti cifre decimali sia associato un insieme coerente di numeri ordinali, cosicché

```
ord ('0') = ord ('1')-1
ord ('1') = ord ('2')-1
        ⋮
ord ('8') = ord ('9')-1
```

In molte implementazioni anche alle maiuscole ed alle minuscole (ove disponibili) sono associati insiemi coerenti di numeri ordinali, in modo che

```
ord ('A') = ord ('B')-1          ord ('a') = ord ('b')-1
ord ('B') = ord ('C')-1          ord ('b') = ord ('c')-1
        ⋮                                ⋮
ord ('Y') = ord ('Z')-1          ord ('y') = ord ('z')-1
```

Tuttavia, questo non è imposto dallo standard del PASCAL, e l'utente dovrebbe controllare che ciò valga per la propria implementazione, prima di scrivere programmi dipendenti dai numeri ordinali associati ai caratteri. Lo standard richiede che i valori ordinali delle lettere siano ordinati, cioè che

```
ord ('A') < ord ('B')            ord ('a') < ord ('b')
ord ('B') < ord ('C')            ord ('b') < ord ('c')
        ⋮                                ⋮
```

Questa condizione è sufficiente per il confronto ordinato di caratteri, spiegato nel

prossimo paragrafo.

L'associazione di valori del tipo *char* su un insieme di interi consecutivi permette che esso sia considerato un tipo ordinale, e che le funzioni standard *succ* e *pred* accettino argomenti di tipo *char*:

pred (c) = *chr (ord (c) − 1)*
succ (c) = *chr (ord (c) + 1)*

3.1.4. Il tipo *boolean*

Un valore *booleano* è uno dei valori logici di verità rappresentati dagli identificatori standard del PASCAL *true* e *false*.

Il PASCAL fornisce degli operatori standard che prendono valori booleani come operandi, producendo un risultato booleano. Tali operatori sono

and	*and* logico
or	*or inclusivo* logico
not	*negazione* logica

OPERATORI BOOLEANI

p	*q*	*p* **and** *q*	*p* **or** *q*	**not** *p*
false	*false*	*false*	*false*	*true*
false	*true*	*false*	*true*	*true*
true	*false*	*false*	*true*	*false*
true	*true*	*true*	*true*	*false*

Tabella 3.3 Azione degli operatori **and, or** e **not.**

La tabella 3.3 mostra i risultati dell'applicazione di tali operatori ad argomenti booleani *p e q*.

Valori booleani possono essere prodotti anche per applicazione di *operatori relazionali* ad operandi di altri tipi. Il PASCAL fornisce i sei operatori relazionali matematici:

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

OPERATORI RELAZIONALI

Una relazione consiste di due operandi separati da un operatore relazionale. Se la relazione è soddisfatta ha il valore *true*, altrimenti il valore *false*, cioè il risultato di una relazione è un valore booleano. Gli operatori di cui sopra possono essere applicati a qualsiasi coppia di operandi dello stesso tipo non strutturato. (Il loro uso con operandi di altri tipi è discusso più avanti).

Per operandi interi e reali gli operatori relazionali hanno il significato tradizionale.

Così

7 = 11	dà *false*
7 < 11	dà *true*
3.4 < = 5.9	dà *true*

Per operandi di tipo *char* il risultato è determinato applicando la relazione matematica di numeri ordinali corrispondenti agli operandi. Così, per ogni operatore relazionale R e operandi *c*1, *c*2 di tipo *char,*

*c*1 R *c*2

è equivalente a

ord (*c*1) R *ord* (*c*2)

Va notato che le restrizioni specificate dallo standard PASCAL per gli ordinali di tipo *char* assicurano la validità delle seguenti relazioni sui caratteri:

'0' < '1'	'A' < 'B'	'a' < 'b'
'1' < '2'	'B' < 'C'	'b' < 'c'
⋮	⋮	⋮

Un'analoga convenzione di ordinamento è adottata per il tipo *boolean* stesso, che è un tipo ordinale per cui valgono

ord (false) = 0	*false = pred (true)*
ord (true) = 1	*true = succ (false)*

Con tale convenzione gli altri operatori *booleani* possono essere espressi in termini degli operatori relazionali:

l'operatore di *implicazione materiale* è denotato dall'operatore < =
l'operatore di *equivalenza* è denotato dell'operatore =
l'operatore di *or esclusivo* è denotato dall'operatore < >.

La tabella 3.4 conferma che questi operatori relazionali corrispondono ai detti operatori logici.

p	*q*	*p* < = *q*	*p* = *q*	*p* < > *q*
false	*false*	*true*	*true*	*false*
false	*true*	*true*	*false*	*true*
true	*false*	*false*	*false*	*true*
true	*true*	*true*	*true*	*false*

Tabella 3.4 Altri operatori booleani

Il PASCAL fornisce tre funzioni standard che producono risultati booleani – tali funzioni sono note come *predicati.* Due di queste, *eoln* e *eof,* sono studiate nel Cap. 5. La terza è definita come segue:

odd (x) per argomento intero *x,* dà *true* se *x* è dispari, *false* altrimenti.

3.1.5 Tipi enumerati

Abbiamo dunque visto i quattro tipi standard, o primitivi, forniti dal PASCAL. Questi tipi di dati possono essere usati per la descrizione di ogni dato numerico, alfabetico o logico. Il programmatore può però accorgersi che nessuno di questi tipi fornisce valori adatti a descrivere i dati che intende introdurre nel suo programma. Per esempio, può voler costruire un programma i cui dati sono i quattro semi di un mazzo di carte (fiori, quadri, cuori e picche) o i sette giorni della settimana (da domenica a sabato). Nessuno di questi è descrivibile con naturalezza in termini dei tipi standard del PASCAL. Fortunatamente, il PASCAL permette al programmatore la definizione dei propri tipi di dato, nella forma adatta per ogni particolare applicazione.

Un *tipo-enumerato* è definito semplicemente elencando gli identificatori dai quali saranno denotati i valori del tipo. Gli identificatori sono racchiusi tra parentesi tonde e separati da virgole.

tipo-enumerato = "(" *lista-identificatori* ")".
lista-identificatori = *identificatore* { "," *identificatore* }.

I semi di un mazzo di carte possono essere definiti come tipo enumerato:

(fiori, quadri, cuori, picche)

Elementi di questo tipo possono assumere solo valori denotati dagli identificatori *fiori, quadri, cuori, picche.*

I giorni della settimana potrebbero essere definiti come tipo enumerato:

(domenica, lunedi, martedi, mercoledi, giovedi, venerdi, sabato)

L'ordine in cui sono enumerati i valori del tipo definisce pure il loro ordinamento agli effetti dell'applicazione di operatori relazionali; così

fiori < *quadri* dà *true*
lunedi > *venerdi* dà *false.*

I sei operatori relazionali descritti precedentemente possono essere tutti applicati a valori di tipi enumerati producendo, come sempre, risultati booleani.

Tutti i tipi enumerati sono per definizione ordinali, e le funzioni standard *succ* e *pred* sono definite sui loro valori. Il successore di un valore di tipo enumerato è il valore successivo nell'enumerazione ordinata (se tale valore esiste), mentre il predecessore è il valore precedente nell'enumerazione (se tale valore esiste).

Per esempio, coi tipi definiti prima, sono vere le seguenti relazioni:

succ (lunedi) = *martedi*
pred (lunedi) = *domenica*

Nel caso di valore massimo di un tipo enumerato *succ* non è definito – analogamente, *pred* non è definito per il valore minimo, per esempio *pred (domenica)* e *succ (sabato)* producono un risultato indefinito.

La funzione standard *ord* (definita precedentemente per il tipo *char*) è altresì applicabile ad argomenti che sono valori di un tipo enumerato. Tale funzione associa ai valori di un tipo enumerato gli interi da 0 a $N-1$, nell'ordine (dove N è il numero di valori, o *cardinalità*), del tipo enumerato. Così

ord (fiori) = 0
ord (quadri) = 1
ord (cuori) = 2
ord (picche) = 3

Si noti che il tipo standard *boolean* è equivalente ad un tipo enumerato della forma *(false, true).*

3.1.6. Tipi intervallo

Quando un dato assume valori in un insieme che è un intervallo dei valori descritti da qualche tipo ordinale esistente, il suo tipo può essere definito come sottointervallo del tipo *ospite.* Il tipo intervallo è definito indicando i *limiti* inferiore e superiore dei valori del sottointervallo, separati dal simbolo " . . ".

tipo-intervallo = costante " . . " costante.

Tutte le possibili forme di una *costante* sono definite nel paragrafo successivo. Così

'A' .. 'Z'	è un sottointervallo del tipo *char*
lunedi .. martedi	è un sottointervallo del tipo enumerato introdotto precedentemente
0 .. 9	
1900 .. 1999	sono sottointervalli del tipo *integer*

In realtà il tipo *integer* è esso stesso un sottointervallo – l'intervallo definito dall'implementazione, dell'insieme infinito dei numeri interi.

nota (a) non è permesso definire un sottointervallo del tipo *real,* poiché non è un tipo ordinale;

nota (b) nella maggior parte delle implementazioni, il sottointervallo *'A' .. 'Z'* definisce come propri valori tutte e sole le 26 lettere maiuscole dell'alfabeto ma, per le ragioni spiegate prima, ciò non è sempre vero. In certe implementazioni questo sottointervallo può definire più di 26 valori, ed includere caratteri non alfabetici.

Un intervallo di questo tipo permette di indicare che i valori assunti da qualche dato specifico sono compresi in un certo sottointervallo. D'altra parte, ogni valore assunto è visto come appartenente al tipo ospite, cosicché gli operatori e le funzioni definite per il tipo ospite sono applicabili al (valore del) dato di un tipo intervallo. Per esempio, gli operatori dell'aritmetica degli interi possono essere applicati ad operandi di tipo sottointervallo degli interi, mentre *succ* e *pred* possono essere applicati ad operandi di qualsiasi tipo intervallo. In generale, comunque, il valore prodotto per applicazione

di un'operazione a degli operandi di un tipo intervallo non appartiene necessariamente allo stesso intervallo. Il risultato di tale operazione è sempre da considerarsi appartenente al tipo ospite.

3.2. DICHIARAZIONI DEI DATI

I dati manipolati da un programma possono essere classificati in due insiemi: quelli il cui valore rimane inalterato durante l'esecuzione di un programma, e quelli i cui valori sono cambiati dall'esecuzione. I primi sono noti come *costanti,* gli altri come *variabili.* Gli elementi di ciascuna classe condividono le proprietà di tipo come descritto nel paragrafo precedente. Il modo in cui i dati si introducono in un programma PASCAL, però, dipende dalla classe di appartenenza, come spieghiamo adesso.

3.2.1. Costanti e definizioni di costante

Nei precedenti paragrafi abbiamo visto come si possono denotare valori particolari di tipi di dato non strutturati. Per esempio

123 denota un particolare valore di tipo *integer;*
12.75 denota un particolare valore di tipo *real;*
'A' denota un particolare valore di tipo *char;*
true denota un particolare valore di tipo *boolean;*
cuori denota un particolare valore di un tipo enumerato.

Un dato costante di tipo non strutturato si può quindi denotare scrivendo esplicitamente il suo valore nel punto del programma dove è usato. Non sempre questa è la migliore soluzione, come si vede nel seguente esempio.

Un programma matematico può far uso frequente del valore reale 3.1415926 che è un approssimazione del valore della costante matematica π . Scrivere questo valore esplicitamente in ogni punto presenta i seguenti svantaggi:

(a) la sequenza reale di cifre è molto meno significativa del simbolo π ;
(b) scrivere ripetutamente per esteso questa sequenza di cifre è noioso, e soggetto ad errori – può capitare di scrivere 3.1415926 o 3.1415962 o 3.1451926 senza che la discrepanza sia notata;
(c) decidere di cambiare, per esempio, la precisione del valore usato per π implica cambiare ogni occorrenza della sequenza di cifre.

La soluzione ideale per il programmatore è l'uso del simbolo π stesso, definendo il valore che denota in un solo punto del programma.

Dato che lettere greche come π non sono disponibili nei sistemi di elaborazione, un identificatore significativo e nemonico, come *pi,* è un sostituto accettabile – soluzione permessa dal PASCAL.

In PASCAL un identificatore può denotare un particolare valore costante, per mezzo di una *definizione-costante.* Dopo di che ogni occorrenza dell'identificatore è equivalente ad un occorrenza esplicita del valore stesso in quel punto. Le definizioni di costante sono raggruppate insieme nella *parte-definizione-costanti,* che ha la seguente forma:

parte-definizione-costanti = "**const**" *definizione-costante* ";"
{ *definizione-costante* ";" }.
definizione-costante = *identificatore* "=" *costante*.
costante = *numero-intero* | *numero-reale* | *stringa*
[*segno*] *identificatore-costante*.

Il seguente è un esempio di *parte-definizione-costanti:*

```
const  pi        = 3.1415926;
       e         = 2.7182;
       asterisco = '*';
```

Il PASCAL fornisce tre identificatori standard, che possono essere usati senza alcuna definizione precedente, e che denotano valori costanti standard. Due di questi, *false* e *true,* denotano i valori del tipo *boolean.* Il terzo denota una costante definita dall'implementazione

maxint

tale che ogni valore intero nell'intervallo chiuso da − *maxint* a + *maxint* è un valore del tipo predefinito *integer* per quell'implementazione.

In questo libro il termine *identificatore-costante* è usato per denotare:

(a) un identificatore definito da una *definizione-costante,*
(b) un identificatore introdotto come uno dei valori di un tipo enumerato,
(c) uno degli identificatori standard *false, true, maxint.*

La categoria sintattica

identificatore-costante = *identificatore*.

è introdotta solo per denotare la classe degli identificatori così definiti.

3.2.2. Definizioni di tipo

I tipi primitivi del PASCAL si possono usare in un programma senza doverli definire prima − ogni riferimento ad uno di essi è indicato dal corrispondente identificatore *integer, real, char, boolean.* È spesso necessario, o conveniente, che i tipi definiti dal programma anch'essi siano denotati da identificatori, così detti *identificatori-tipo.*

Ogni identificatore di tipo è introdotto per mezzo di una *definizione-tipo.* Le definizioni di tipo sono raggruppate insieme in una *parte-definizione-tipi* che ha la seguente forma:

parte-definizioni-tipi = "**type**" *definizione-tipo* ";" { *definizione-tipo* ";" }.
definizione-tipo = *identificatore* "=" *tipo*.
tipo = *tipo-semplice* | *tipo-strutturato* | *tipo-puntatore* | *identificatore-tipo*.
tipo-semplice = *tipo-enumerato* | *tipo-intervallo*.

Le possibili alternative *tipo-strutturato* e *tipo-puntatore* sono oggetto di capitoli successivi. La (1) è un esempio di *parte-definizione-tipi* che include tipi semplici.

```
type   giornidellasettimana = (domenica, lunedi, martedi, mercoledi,
                               giovedi, venerdi, sabato);          (1)
       giornidelmese        = 1..31;
       anno                 = 1900..1999;
       giornolavorativo     = lunedi..venerdi;
```

Con queste definizioni tali tipi possono essere denotati dagli identificatori *giornidellasettimana, giornidelmese, anno* e *giornolavorativo,* allo stesso modo in cui i tipi standard sono denotati da *integer, real, char, boolean.*

In questo libro il termine *identificatore-tipo* è usato per denotare

(a) un identificatore introdotto con una definizione di tipo,
(b) uno degli identificatori standard forniti in PASCAL per denotare tipi standard, cioè *integer, real, char, boolean* e *text.* (Il tipo *text* è spiegato nel Cap. 12).

La categoria sintattica

identificatore-tipo = identificatore.

è usata per denotare la classe degli identificatori così definiti.

La sintassi di *definizione-tipo* permette che un identificatore di tipo sia definito in termini di un altro definito altrove:

```
type   t1 = ..... ;
        :
       t2 = t1;
```

Con tale definizione si dice che l'identificatore di tipo *t2* denota lo stesso tipo denotato da *t1*.

3.2.3. Dichiarazioni di variabile

Le *variabili,* cioè i dati i cui valori sono cambiati dall'esecuzione del programma, sono sempre denotate da identificatori. Ogni *identificatore-variabile* è introdotto da una *dichiarazione-variabile,* che specifica anche il tipo dei valori che la variabile può assumere. Le dichiarazioni delle variabili sono raggruppate in una *parte-dichiarazione-variabili,* che ha la seguente forma:

*parte-dichiarazione-variabili = "***var***" dichiarazione-variabile ";"*
{ dichiarazione-variabile ";" }.
dichiarazione-variabile = lista-identificatori ":" tipo.

Date le definizioni di tipo del sottoparagrafo precedente, il seguente è un esempio di *parte-dichiarazione-variabili:*

```
var   oggi                    : giornodellasettimana;
      questanno, prossimoanno : anno;
      giornilavorati          : 0..5;
      inpartenza              : boolean;
```

L'identificatore *oggi* denota un dato variabile che in ogni momento può assumere uno dei sette valori *domenica, lunedì, ..., sabato; questanno* e *prossimoanno* denotano dati distinti che possono assumere valori interi nell'intervallo 1900..1999.

Si noti che

(a) più variabili possono essere dichiarate con la stessa dichiarazione, purché siano dello stesso tipo;
(b) il tipo delle variabili dichiarate può essere definito esplicitamente nella dichiarazione di variabile stessa – se questo è il solo punto del programma in cui è usato il tipo, allora è superfluo introdurre un identificatore per dare il nome al tipo con una definizione di tipo;
(c) si dice che due variabili hanno lo stesso tipo se sono state dichiarate con la stessa dichiarazione, oppure se sono state dichiarate (in diverse dichiarazioni) con lo stesso identificatore di tipo, o con diversi identificatori di tipo denotanti lo stesso tipo.

D'ora innanzi la categoria sintattica

identificatore-variabile = identificatore.

sarà usata per denotare la classe degli identificatori introdotti attraverso una dichiarazione di variabile.

3.3. UNICITÀ E ORDINE DI DEFINIZIONE DEGLI IDENTIFICATORI

Le definizioni di costante, i tipi enumerati, le definizioni di tipo e le dichiarazioni di variabile hanno tutti l'effetto di associare agli identificatori le costanti, i tipi o le variabili, per denotare i quali sono usati. (Altri modi per associare identificatori ad altri oggetti del programma sono descritti nei capitoli seguenti).

Il PASCAL richiede che l'associazione di ogni identificatore con l'oggetto che denota deve essere unica per tutto il suo *campo d'azione* all'interno del programma. Il concetto di campo d'azione di un identificatore è discusso nella sua interezza nel Cap. 7. Per il momento si può assumere che l'associazione di ciascun identificatore sia unica per tutto il programma.

Così le definizioni

```
type  intervallo      : 1..100;
var   x, y            : intervallo;
      y,z, intervallo : real;
```

sono scorrette, poiché gli identificatori *intervallo* e *y* sono entrambi definiti due volte (nello stesso campo d'azione).

La sintassi del PASCAL richiede pure che la *parte-definizione-costanti,* la *parte-definizione-tipi* e la *parte-dichiarazione-variabili* di un programma o blocco, appaiano esattamente in quest'ordine (in riferimento alla definizione del Cap. 2), cioè come in (2).

```
blocco = ...
         [ parte-definizione-costanti ]
         [ parte-definizione-tipi ]                    (2)
         [ parte-dichiarazione-variabili ]
         ....
```

Ciò impone che gli identificatori di costante siano definiti prima di poter essere usati nella *parte-definizione-tipi,* che gli identificatori di tipo siano definiti prima di poter essere usati nella dichiarazione di variabili, che gli identificatori di variabile siano dichiarati prima di poter essere usati per descrivere le azioni da intraprendere sulle variabili, ecc. Questa proprietà di definizione prima dell'uso nel testo del programma aumenta di parecchio la leggibilità del programma stesso. Il vincolo sintattico definire-prima-di-usare non è però sufficiente – un identificatore di costante può essere usato per definirne un altro nella stessa *parte-definizione-tipi.* Il PASCAL impone perciò un vincolo generale di definizione-prima-dell'uso per tutti gli identificatori, con una significativa eccezione per i tipi puntatore, per i quali vedremo che tale regola sarebbe impossibile da osservare. Così non sono permesse le seguenti definizioni:

```
const  xmin   = −xmax;
       xmax   = 100;
type   ycampo = xcampo;
       xcampo = xmin..xmax;
```

perché *xmax* e *xcampo* sono entrambi usati prima di essere definiti, mentre sono accettabili le seguenti definizioni equivalenti:

```
const  xmax   = 100;
       xmin   = −xmax;
type   xcampo = xmin..xmax;
       ycampo = xcampo;
```

ESERCIZI

3.1 Quali delle seguenti combinazioni di operatori ed operandi sono valide in PASCAL?

67 **div** 8	12*2.75	67 **mod** 8	(−3) **div** 8
succ (66.3)	*sqr* (2.5)	*'A' < 'Z'*	*pred* ('7')
67 / 8	6.7 **div** 8	*sqr* (10 **E** 4)	*sqr* (100001)
sqrt (6.25)	*sqrt* (4)	*succ (true)*	*exp* (1.0 **E** 60)

Scrivere il risultato di ogni combinazione valida. I risultati sono tutti definiti sulla vostra implementazione?

3.2 Esaminare la seguente sequenza di definizioni e dichiarazioni PASCAL:

```
const  lineamax          = 64;
       richiestastampa   = true;
type   posizionelinea    = 1..lineamax;
       spaziatura        = (singola, doppia, tripla);
var    questocar, ultimocar   : char;
       questaposizione        : posizionelinea;
       spaziaturaattuale      : spaziatura;
```

Elencare tutti gli *identificatori-costante*;
gli *identificatori-tipo*;
gli *identificatori-variabile*
che ci sono.

3.3 Scrivere una *parte-definizione-costanti* che definisca i seguenti identificatori di costante, coi valori appropriati:

pollicipermetro *simbolocorrente*
gradiperradiante *limitedivelocita*

3.4 Scrivere una *parte-definizione-tipi* che definisca tipi adatti per valori di età, sesso, altezza, peso e stato civile di una persona.

3.5 Questa è una tipica descrizione di una casa offerta da un'agenzia immobiliare: "Tre camere da letto, due soggiorni, riscaldamento centrale a nafta, rimessa, ...". Scrivere una *parte-dichiarazione-variabili* che dichiari le variabili necessarie, ciascuna col tipo appropriato.

4.

Istruzioni, Espressioni ed Assegnazioni

4.1. ISTRUZIONI

Nei Capp. 2 e 3 sono stati introdotti i concetti di dato e tipo di dato, ed il modo di definire nei programmi PASCAL gli elementi costanti e variabili – la *parte-dichiarazioni.* Le manipolazioni che il programma compie sui suoi dati vengono definite dalla *parte-istruzioni.* In essa le azioni da realizzare vengono espresse come sequenze di *istruzioni,* in cui ogni istruzione specifica la corrispondente azione. Il PASCAL è un linguaggio di programmazione *sequenziale,* nel senso che le istruzioni vengono eseguite sequenzialmente nel tempo, cioè una dopo l'altra e mai simultaneamente.

L'esecuzione sequenziale delle istruzioni si riflette nella struttura della *parte-istruzioni,* che è la seguente

parte-istruzioni = "**begin**" *istruzione* { ";" *istruzione* } "**end**".

Nel Cap. 2 è stato presentato un programma che contiene un semplice esempio di *parte-istruzioni,* formata da tre istruzioni da eseguire nell'ordine dato, come mostrato in (1).

```
begin
  read (primo, secondo);                    (1)
  somma := primo + secondo;
  write (somma)
end
```

Il PASCAL consente diverse forme di istruzioni, sia semplici che strutturate, per mezzo delle quali si possono specificare le azioni:

istruzione = [*etichetta* ":"] (*istruzione-semplice* | *istruzione-strutturata*).

Le *etichette* delle istruzioni hanno una importanza relativa nella costruzione di programmi PASCAL e, fino alla loro trattazione nel cap. 8, vengono ignorate. Le *istruzioni-strutturate* si usano per descrivere azioni composte in termini delle istruzioni com-

ponenti, e sono discusse nel Cap. 6. Per ora consideriamo l'*istruzione-semplice,* di cui esistono quattro forme possibili

istruzione-semplice = [*istruzione-assegnazione* | *istruzione-procedura* | *istruzione-goto*].

Una istruzione che non contiene simboli prende il nome di *istruzione-vuota,* e indica nessuna azione. Come vedremo, si usa in alcuni punti del programma per indicare che non deve essere eseguita alcuna azione.

L'*istruzione-procedura* e l'*istruzione-goto* vengono discusse nei capitoli successivi.

L'*istruzione-assegnazione* si usa per assegnare un particolare valore ad una variabile. Il valore viene specificato per mezzo di una *espressione,* di cui innanzitutto consideriamo le forme possibili.

4.2. ESPRESSIONI

Un'*espressione* è una regola per il calcolo di un valore. Essa è costituita da uno o più operandi, combinati per mezzo degli operatori già definiti nel Cap. 3. Un operando può essere uno qualsiasi dei seguenti:

un valore costante;
il valore corrente di una variabile;
il risultato di una chiamata di funzione;
un insieme.

Per ora le uniche chiamate di funzione che consideriamo sono quelle relative alle funzioni standard, definite nel Cap. 3. L'uso dei limiti di array conforme viene discusso nel Cap. 9 e l'uso degli insiemi nel Cap. 11.

Come nel caso delle normali espressioni matematiche, un'espressione PASCAL è valutata secondo le regole di *precedenza degli operatori.* Ad ogni operatore viene assegnata una *precedenza.* Nella tabella 4.1 è riportata la precedenza degli operatori del PASCAL.

Precedenza	Operatori
4	**not**
3	*** / div mod and**
2	**+ − or**
1	**= <> > < >= <=**

Tabella 4.1 Precedenze tra operatori.

Perció le regole per la valutazione di un'espressione sono:

(a) se in una espressione tutti gli operatori hanno la stessa precedenza, la valutazione delle operazioni procede strettamente da sinistra a destra;

(b) quando sono presenti operatori con precedenza diversa vengono valutate per prime le operazioni con precedenza più alta (sulla base della regola da sinistra a destra), poi, delle rimanenti, quelle a precedenza più alta, e così via;
(c) le regole (a) e (b) possono non essere rispettate se si usano le parentesi – in tal caso vengono valutate per prime le operazioni tra parentesi, applicando all'interno delle parentesi le regole di precedenza già viste;
(d) l'ordine di valutazione degli operandi di un operatore binario, cioè con due operandi, dipende dall'implementazione – come vedremo nel Cap. 7, questo va ricordato quando si usano alcune funzioni.

L'effetto di queste regole di valutazione viene meglio descritto da alcuni esempi.

(a) 6*2+4*3
Poiché gli operatori * sono a precedenza più alta, questa espressione è equivalente a (6*2) + (4*3), che dà come risultato 24.
(b) Volendo sommare 4 e 2, e moltiplicare il risultato per 3, si può usare la seguente espressione

```
(4+2)*3
```

Si noti che

```
4+2*3
```

produce un risultato scorretto perché l'operazione *, essendo a precedenza più alta, viene valutata per prima.
(c) 10 **mod** 3*4
In questo caso i due operatori sono con uguale precedenza, per cui vengono valutati da sinistra a destra ed il risultato è 4.

Alcune operazioni booleane, tra cui **and** e **or** possono essere valutate senza che entrambi gli operandi siano stati valutati. Ad esempio

false **and** *b*
b **or** *true*

danno come risultato i valori *false* e *true,* rispettivamente, indipendentemente dal valore di *b.* Ad ogni modo, nella definizione del PASCAL non è specificato se in questi casi devono essere valutati entrambi gli operandi. In alcune implementazioni il secondo operando *b* in espressioni come

a **and** *b*
a **or** *b*

viene valutato anche se il primo, *a*, è *false* o *true,* rispettivamente; in altre non è così. Come vedremo nei programmi che seguono, bisogna stare attenti ad evitare espressioni dipendenti dal metodo di valutazione usato.

Le regole date nel Cap. 3 determinano il tipo del risultato di ciascun operatore che compare in un'espressione a partire dal tipo dei suoi operandi. Il tipo di un'espressio-

ne completa viene determinato applicando queste regole a ciascun operatore e ai suoi operandi, nell'ordine in cui devono essere realizzate le operazioni. Ad esempio,

(a) 2 + *sqrt* (*sqr* (3) + 11*5)

Prima viene valutato il contenuto delle parentesi più esterne. Al loro interno *sqr* (3) viene valutato per primo, ottenendo un risultato intero, poi 11 * 5, che dà di nuovo un risultato intero; questi due risultati vengono sommati dando un risultato intero (64).

Poi viene valutata *sqrt* (64), ottenendo un risultato reale (8.0); finalmente, 2 è sommato a 8.0, e dà un risultato reale (10.0).

(b) In alcuni casi l'omissione delle parentesi non provoca valori scorretti, ma in pratica produce espressioni non legali; per esempio,

(3 > 0) **and** (3 < 10)

dà il valore *true*. Se però si omettono le parentesi, cioè

3 > 0 **and** 3 < 10

l'espressione è scorretta. Le regole di precedenza impongono che l'operazione **and** venga eseguita prima, ma 0 **and** 3 non è un'espressione legale, perché **and** richiede operandi di valore booleano.

Il lettore dovrebbe studiare le espressioni della Tabella 4.2 e convincersi, applicando le regole per la valutazione delle espressioni, che queste producono i valori indicati.

Espressione	Valore
3*17−193 **mod** 19	48
193 **mod** 19 **div** 3*127	127
(−4 + 23*2) **div** 3	14
(7+3) < = 10	*true*
(3 < 7) **or** (3 < = 7)) **and** (7 < > 10)	*true*
(37 + 15 **mod** 4)/*sqr* (5)	1.6
Trunc (3.2*4) **div** 3	4

Tabella 4.2 Alcune espressioni e loro valore.

La notazione PASCAL per le espressioni è meno flessibile di quella usata in matematica, ed occorre fare attenzione nel tradurre espressioni matematiche in PASCAL. Nella Tabella 4.3 sono indicate alcune espressioni matematiche ed il modo di esprimerle in PASCAL.

Espressione matematica	Espressione PASCAL
$\frac{x+2}{y+4}$	$(x + 2)/(x + 4)$
$x\,(y + w\,(3 - v))$	$x * (y + w * (3 - v))$
$\frac{x\,y}{w+2}$	$x * y / (w + 2)$
$a + \frac{b}{c}$	$a + b / c$

Tabella 4.3 Alcune espressioni matematiche e le loro corrispondenti in PASCAL

Per quanto riguarda le espressioni occorre puntualizzare alcuni aspetti importanti:

(a) ad ogni variabile usata in una espressione deve essere già stato attribuito in qualche modo un valore, altrimenti la valutazione dell'espressione provoca un errore;
(b) non si devono mai scrivere due operatori aritmetici uno a fianco all'altro: per esempio, $a * -b$ non è un'espressione legale e va scritta $a * (-b)$;
(c) non si può mai omettere il segno di moltiplicazione quando tale operazione è richiesta;
(d) non c'è niente di male nell'uso di parentesi ridondanti. Perciò, in caso di dubbi sulle regole di precedenza, conviene usare le parentesi per costruire le espressioni.

Il metodo di costruzione delle espressioni, e le regole di precedenza degli operatori che determinano la loro valutazione, sono definiti dalla sintassi (2) (queste definizioni sintattiche riassumono semplicemente le regole già viste, e vengono presentate in (2) soltanto per poter fare riferimento ad esse).

espressione = *espressione-semplice* [*operatore-relazionale espressione-semplice*].
espressione-semplice = [*segno*] *termine* {*operatore-addizione termine*}.
termine = *fattore* { *operatore-moltiplicazione fattore* }.
fattore = *variabile* | *costante-senza-segno* | *designatore-funzione* | *limite* | *insieme* | "(" *espressione*")" | "**not**" *fattore*.
costante-senza-segno = *intero-senza-segno* | *reale-senza-segno* | *stringa* | *identificatore-costante* | "**nil**". (2)
operatore-relazionale = "=" | "<>" | "<" | ">" | "<=" | ">=" | "**in**".
operatore-addizione = "+" | "−" | "**or**".
operatore-moltiplicazione = "*" | "/" | "**div**" | "**mod**" | "**and**".

L'occorrenza di una variabile in una espressione indica che nella valutazione dell'espressione viene usato come operando il valore corrente della variabile. Come abbiamo detto nel Cap. 3, una variabile è denotata da un identificatore specificato in

una *dichiarazione-variabile.* Gli identificatori introdotti per dichiarazione di variabile dei tipi di dato considerati nel Cap. 3, denotano quelle che abbiamo definito *variabili-semplici.* La definizione sintattica completa di una variabile denotata all'interno della *parte-istruzioni* è

variabile = variabile-semplice | variabile-componente | variabile-puntata.
variabile-semplice = identificatore-variabile.

Nei capitoli successivi vedremo che, in alcuni casi, le variabili sono costituite da più *componenti,* ciascuno dei quali può essere considerato a buon diritto una variabile con una corrispondente notazione.

4.3. L'ISTRUZIONE DI ASSEGNAZIONE

Una *istruzione-assegnazione* ha la seguente forma:

istruzione-assegnazione = (variabile | identificatore-funzione) ":=" espressione.

La seconda forma di assegnazione si usa nella costruzione di funzioni, che è discussa nel Cap. 7. Il simbolo ":=" è noto come *operatore di assegnazione,* e di solito si legge "diventa".

L'effetto dell'esecuzione di un'*istruzione-assegnazione* è la valutazione dell'espressione alla destra dell'operatore di assegnazione e l'assegnazione del valore risultante alla variabile che si trova sulla sinistra.

Ad esempio,

$i := 3$

significa rimpiazza il valore corrente della variabile i con il valore 3;

$i := j$

significa rimpiazza il valore corrente della variabile i con il valore corrente della variabile j.

Alle variabili di tutti i tipi che abbiamo introdotto finora, siano essi primitivi, enumerati, o intervallo, si possono assegnare nuovi valori. In ogni caso, ad una variabile può essere assegnato soltanto un valore di un tipo appropriato. Perciò, la variabile a sinistra dell'operatore di assegnazione ed il valore dell'espressione a destra devono essere di tipo identico, con le seguenti eccezioni:

(a) una espressione costituita soltanto da una variabile di un tipo intervallo viene considerata del corrispondente tipo ospite;

(b) ad una variabile di un tipo intervallo può essere assegnato un valore del corrispondente tipo ospite, purché il valore appartenga all'intervallo richiesto;

(c) ad una variabile reale può essere assegnato un valore intero, come specificato dalla regola di sostituzione di reali per interi del Cap. 3.

Un valore che soddisfa queste condizioni si dice *compatibile per assegnazione* con il ti-

po della variabile alla quale deve essere assegnato. La compatibilità per assegnazione nel caso di tipo insieme e di tipo stringa è discussa nei Capp. 9 e 11 rispettivamente.

Il risultato dell'assegnazione di un valore al di fuori dell'intervallo richiesto ad una variabile di tipo intervallo è un errore. La maggior parte delle implementazioni fornisce strumenti per individuare questi errori durante l'esecuzione.

Si consideri la seguente sequenza di istruzioni di assegnazione:

```
a := 3.52;
a := b * b * b;
j := j + 1;
p := (j > 0) or (c <> ' ');
c := '.'
```

Perché queste siano istruzioni di assegnazione legali la variabile *a* deve essere di tipo *real*, *b* e *j* possono essere di tipo *integer* o *real*, *p* di tipo *boolean* e *c* di tipo *char* (o loro intervalli).

Come esercizio, il lettore dovrebbe verificare che la seguente sequenza di istruzioni di assegnazione, applicate alle variabili intere *x*, *y* e *w*, le lascia tutte con valore finale 100:

```
x := 27;
y := 343;
w := x + y - 300;
x := w div 10 + 23;
y := (x + w) div 10 * 10;
x := x + 70 + y mod 10;
w := w + x - 70
```

ESERCIZI

4.1 Inserire le parentesi in modo che chiariscano il significato delle seguenti espressioni, e poi calcolarne il valore:

```
6.75 - 12.3 / 3
6 * 11 - 42 div 5
175 mod 15 div 3 * 65
13 + 7 * 5 - 4 * 5 div 2
11 mod 4 div 2 <> 0
('A' <= 'Z') or ('9' <= '8') and ('A' < 'I')
```

4.2 Assumendo che *X*, *Y* e *Z* siano variabili intere, quali valori avranno dopo l'esecuzione della seguente sequenza di istruzioni?

```
X := 50;
Y := 340;
Z := X + Y - 190;
X := 17;
Y := X + Z;
Z := X + 200;
Z := X + Z - 200;
```

4.3 Scrivere le espressioni booleane che determinano se

(a) il valore di una variabile intera i è nell'intervallo da 1 a 100, estremi inclusi;
(b) il valore di una delle due variabili intere j e k è un multiplo dell'altro;
(c) l'anno y del ventesimo secolo è bisestile.

4.4 (a) Scrivere una sequenza di istruzioni di assegnazione che permetta di scambiare il valore di due variabili reali x ed y, usando altre variabili se occorre.

(b) *X, Y* e *Z* sono variabili intere. Scrivere le istruzioni di assegnazione per assegnare la somma, il prodotto e la loro media alle variabili chiamate *somma, prodotto* e *media.* Di che tipo avete assunto le variabili *somma, prodotto e media?*

(c) *C, D* ed *U* sono variabili dichiarate nel seguente modo:

C, D, U : '0'..'9'

cioè assumono i valori *c, d* ed *u* che sono i caratteri delle cifre decimali. Scrivere un'istruzione di assegnazione che assegni ad una variabile intera *I* il numero decimale denotato dai caratteri

cdu

nell'ordine.

5.

Input ed output dei dati

5.1. TRASFERIMENTO DI INFORMAZIONI AL E DAL PROGRAMMA

Scopo di ogni programma è la manipolazione di dati – un programma per fare le paghe opera su dati che descrivono gli impiegati e le ore che essi hanno lavorato; un programma per la risoluzione di sistemi di equazioni lineari deve conoscere i coefficienti di ciascuna equazione. Questi dati potrebbero in realtà essere inclusi nel programma, ma allora, per poter calcolare le paghe della settimana successiva, o risolvere un'altro sistema di equazioni, sarebbe necessario cambiare il programma. Ciò che serve, in effetti, è un modo per informare il programma sui dati da manipolare in ogni particolare occasione, cioè un mezzo per inserire i dati nel programma durante la sua esecuzione, da qualche dispositivo esterno al calcolatore. Allora gli stessi programmi potrebbero essere usati per eseguire il calcolo delle paghe di ogni settimana, o per risolvere parecchi sistemi di equazioni lineari.

Esistono molti dispositivi esterni – i lettori di schede leggono le informazioni perforate su schede, i lettori di nastro leggono quelle su nastro, le tastiere delle telescriventi presentano al calcolatore le informazioni battute sui tasti. Questi, e molti altri, sono mezzi per dare dati di ingresso ad un programma.

Analogamente, vogliamo vedere i risultati prodotti da un programma – un programma di paghe deve produrre prospetti stampati leggibili dagli impiegati, ed assegni da incassare; sarà necessario poter esaminare la soluzione di un sistema di equazioni lineari. Occorre, perciò, un mezzo per comunicare i risultati di un programma al mondo esterno. Di nuovo, esistono dispositivi, come le stampanti di linea, le telescriventi, gli schermi video, sui quali possono essere mostrate le informazioni. Tali dispositivi sono mezzi di uscita dei dati da un programma.

Tutti i dispositivi menzionati sinora hanno due proprietà in comune. Innanzitutto essi forniscono tutti i caratteri dell'insieme accettato dal calcolatore con cui comunicano; in secondo luogo le loro informazioni sono strutturate come successioni di linee (per esempio, una scheda perforata costituisce una linea).

In PASCAL l'input (ingresso) e l'output (uscita) non sono considerati in termini di dispositivi reali, come lettori o stampanti. Invece il programma è considerato un processo che ottiene informazioni da un *canale di input,* e spedisce informazioni su un *ca-*

nale di output, entrambi appartenenti a dispositivi esterni, la cui natura non è ulteriormente specificata. Ogni implementazione del PASCAL assume l'esistenza di questi due canali, organizzati come sequenze di linee di caratteri. Così un programma PASCAL non contiene alcuna indicazione sui reali dispositivi usati – ciò dipende unicamente dall'ambiente in cui opera l'implementazione.

Il lettore, forse, ha notato che l'intestazione del programma considerato esempio nel Cap. 2 contiene un riferimento all'input ed all'output nella forma

program *addizione (input, output)*;

Questo indica semplicemente che il programma fa uso dei canali di input ed output sui dispositivi standard di ingresso ed uscita forniti dall'istallazione dell'elaboratore su cui si opera.

Un'intestazione simile sarà usata per ogni programma di questo libro, fino al Cap. 12, ove saranno discussi strumenti più complessi di input ed output, cui corrispondono intestazioni complesse.

Abbiamo detto sopra che i canali di input ed output sono organizzati per sequenze di linee di testo, ciascuna delle quali contiene caratteri appartenenti all'insieme disponibile. Per certi dispositivi la separazione di una linea dalla successiva è indicata da speciali caratteri di "controllo", che non sono valori dell'insieme di caratteri descritto dal tipo primitivo *char* del PASCAL. Tuttavia, il PASCAL permette di controllare la struttura a linee dei canali di input ed output, per mezzo di speciali funzionalità del linguaggio.

5.2. INPUT IN PASCAL

Un'istruzione di input, in PASCAL, presenta una delle seguenti due forme:

read (lista-variabili)
read (input, lista-variabili)

dove una *lista-variabili* è una lista di uno o più *identificatori-variabile,* separati da virgole:

lista-variabili = variabile { *","* *variabile* }.

Ogni variabile nella *lista-variabili* deve essere di tipo *integer, real* o *char,* o di un tipo che è sottointervallo di *integer* o *char.* Notare che l'input diretto di un tipo enumerato, o di un tipo *boolean,* non è permesso.

L'azione eseguita, per entrambe le forme dell'istruzione di lettura, è di ottenere dal canale standard di input il numero richiesto di valori (cioè tanti valori quante sono le variabili nella lista), ed assegnare tali valori, nell'ordine, alle variabili della lista. Per esempio, l'istruzione

read (a)

otterrà il valore successivo dal canale standard di input, ed assegnerà tale valore alla variabile *a.* L'istruzione

read (alfa, beta, gamma)

otterrà tre valori dal canale standard di input, e li assegnerà alle variabili *alfa, beta, gamma,* nell'ordine. Queste due istruzioni possono essere scritte

read (input, a)
read (input, alfa, beta, gamma)

ed avranno esattamentre lo stesso effetto. Così

read (v1, v2, v3, ..., vn)

è equivalente a

read (v1); *read (v2)*; *read (v3)*; ...; *read (vn)*;

Le istruzioni *read (v)* e *read (input, v)* possono essere pensate come istruzioni di assegnazione che ottengono il valore da assegnare a *v* dal canale di input, piuttosto che da un'espressione. Perciò il valore ottenuto deve essere compatibile per assegnazione col tipo della variabile alla quale deve essere assegnato. Il modo in cui è rappresentato il valore sul canale di input dipende dal tipo di *v*.

Ogni valore intero o reale sul canale di input può essere rappresentato come una successione di caratteri di qualsiasi forma permessa per tali numeri (vedi Cap. 2), e può essere immediatamente preceduto da un segno più o meno. Ciascun valore intero o reale può essere preceduto da un numero qualsiasi di caratteri di spazio o fine linea, ma non debbono esserci spazi o fine linea tra segno e numero.

Un valore di tipo *char* sul canale di input è rappresentato semplicemente dal carattere da assegnare. Gli spazi sono significativi, in questo caso, e non vengono ignorati. Lo stesso vale in questo tipo di input per i fine linea – ciascun fine linea è trattato come un carattere, ma il valore assegnato alla variabile è lo spazio.

I dati di input di un programma sono trattati come un flusso continuo sul canale. Ogni istruzione di input che viene eseguita, comincia ad ottenere valori dal punto del canale immediatamente seguente l'ultimo valore ottenuto dall'istruzione di input eseguita precedentemente. La lettura di un valore intero fa sì che siano saltati tutti gli spazi sul canale di input. Viene poi letto il dato successivo, che deve essere un numero intero valido. La lettura di valori reali causa la scansione del canale di input alla ricerca di un intero o di un reale valido. Nel caso di un carattere l'effetto è semplicemente la lettura del carattere successivo sul canale. Perciò i valori sul canale debbono essere sincronizzati con le variabili elencate dall'istruzione di input. Nessun valore del canale di input può essere letto più di una volta.

Come esempio delle forme possibili di dati di input si consideri la seguente istruzione:

read (valorereale, c1, c2, c3, valoreintero)

dove *valorereale* è una variabile di tipo *real, c1, c2* e *c3* sono variabili di tipo *char, valoreintero* è una variabile di tipo *integer.*

I seguenti quattro esempi di canali di input assegnano tutti gli stessi valori (640.0, *'X'*, *'Y'*, *'Z'*, 104) alle cinque variabili.

(a) 640.0*XYZ*104
(b) 64**E**1*XYZ* 104
(c) +6.4**E**+02*XYZ*
104
(d) 640*XYZ* 104

L'effetto dell'istruzione *read* usando una di queste quattro possibilità è dunque equivalente alle cinque assegnazioni

valorereale := 640.0; *c1* := *'X'*; *c2* := *'Y'*; *c3* := *'Z'*;
valoreintero := 104

Quando, leggendo il canale di input, si raggiunge la fine di una linea, allora il valore disponibile per un input di caratteri è lo spazio. In certi casi il programma deve poter accorgersi che è stata raggiunta la fine di una linea. A questo scopo il PASCAL fornisce un predicato, o funzione booleana, standard, che si scrive

eoln

oppure

eoln (input)

Questo ritorna il valore *true* quando è stato letto l'ultimo carattere effettivo della linea corrente di input, altrimenti dà *false*. Così, se si tenta di leggere un carattere quando *eoln (input)* è *true*, il valore ottenuto è il carattere di spazio, ed *eoln (input)* diventa *false* (a meno che ovviamente la linea successiva non consista di zero caratteri, nel qual caso *eoln (input)* ridiventa *true*).

Una forma speciale di istruzione *read* permette di saltare il contenuto di una linea di input. È l'istruzione

readln

o, in altra forma,

readln (input)

che tralascia il resto della linea corrente del canale di input. Dopo che è stata eseguita un'istruzione *readln* il primo carattere disponibile sul canale di input è il primo carattere della linea successiva. Se si vogliono leggere i valori delle variabili *v1*, ..., *vn* e poi saltare all'inizio della linea successiva, si può scrivere

readln (v1, ..., vn)

oppure

readln (input, v1 ..., vn)

Questo è equivalente (a)

read (v1, ..., vn); *readln*

Notare che la lettura di *v1*, ..., *vn* può a sua volta causare la lettura di parecchie linee, e che a linea nuova si salta dopo aver letto le variabili.

Per esempio, supponiamo di avere tre linee sul canale di input, ciascuna contenente almeno due valori interi. Vogliamo assegnare il secondo intero di ciascuna linea alle variabili *x, y,* e *z,* rispettivamente, ed ignorare tutti gli altri valori. Questo si può ottenere introducendo un'altra variabile intera, diciamo *w,* e scrivendo

```
readln (w, x);
readln (w, y);
readln (w, z)
```

Il canale di input di un programma è di solito di lunghezza finita. Ovviamente, un programma non può leggere più dati di quanti ne fornisca il suo canale di input. In certi programmi è perciò necessario poter scoprire quando è stata raggiunta la fine del canale di input. A questo scopo il PASCAL fornisce un predicato, o funzione booleana, standard,

```
eof
```

oppure

```
eof (input)
```

che ritorna il valore *true* se e solo se è stata raggiunta la fine dei dati in input, altrimenti il suo valore è *false.*

Il canale di input è sempre composto da un numero intero di linee. Dunque è in generale sufficiente testare la funzione *eof.*

(a) prima di ogni input (nel caso che il canale di input sia vuoto), e
(b) dopo ogni *readln,* oppure
(c) dopo aver letto un carattere di spazio quando *eoln* è vero.

Si ha un errore se si tenta un operazione *read, readln* o *eoln* quando *eof* è vero. L'uso di *eoln* e di *eof* è illustrato in vari programmi di esempio nei prossimi capitoli.

5.3. OUTPUT IN PASCAL

L'istruzione base di output in PASCAL ha due forme:

```
write (lista-output)
write (output, lista-output)
```

dove la *lista-output* è una lista di valori in uscita. La sua azione in esecuzione è la spedizione di tali valori, nell'ordine, al canale di output standard. Per esempio

```
write (2 * x + 7)
```

provocherà l'output di un valore, mentre

```
write (a, b, c + 1)
```

provocherà l'output di tre valori.

Come per i dati di input l'output generato da un programma PASCAL è trattato come un flusso continuo di informazioni su un canale. Il programma per l'uscita sul dispositivo di output standard può tuttavia strutturare tale flusso in linee. Sul canale di output possono essere generate nuove linee con l'istruzione

writeln (output)

o, semplicemente

writeln

Se si richiede l'output di una lista di valori (sulla corrente linea di output), e poi il salto a linea nuova, si possono usare le istruzioni

writeln (output, lista-output)

oppure

writeln (lista-output).

Si consideri la seguente successione di istruzioni

```
writeln ('A', 'B');
write ('C');
writeln ('D');
writeln;
writeln ('E');
```

L'output risultante dall'esecuzione di tale sequenza sarebbe:

```
AB
CD
E
```

Su certi dispositivi di output le linee di informazioni possono essere raggruppate in pagine, ed è possibile fare in modo che la successiva linea di output sia la prima di una nuova pagina. A questo scopo il PASCAL fornisce un'altra istruzione con due forme alternative

```
page
page (output)
```

il cui effetto è un salto a pagina nuova sul dispositivo di output.

La *lista-output* di un'istruzione *write* o *writeln* enumera i valori da mandare in output, separandoli per mezzo di virgole:

lista-output = *valore-output* { "," *valore-output* }.

Ogni valore di output deve essere di tipo *integer, real, char, boolean* oppure una stringa.

Anche il canale di output, come quello di input, è una sequenza di caratteri. L'uscita di un valore di tipo *char* semplicemente aggiunge quel carattere alla sequenza mandata in output fino a quel momento. Mandare in output una stringa significa scrivere la sequenza di caratteri che compongono la stringa in aggiunta alla precedente sequenza di output. Un valore booleano in output è la stringa *'True'* oppure quella *'False'*.

Un valore intero o reale è mandato in output mediante la corrispondente sequenza di caratteri (contenente cifre, segni, punti decimali, ecc.), che ne denota il valore. Per esempio, l'istruzione

*write (*27+3, ' = ', 30, *'IS'*, 27+3=30*)*

che stabilisce l'output di una lista formata da un valore intero, un carattere, un altro intero, una stringa ed un valore booleano, potrebbe aggiungere la sequenza di caratteri

30= 30 *IS TRUE*

al canale di output.

Il numero esatto di caratteri stampati per ciascun valore, chiamato *ampiezza-campo,* è determinato dal modo in cui il valore è denotato nella lista di output. Nella sua forma più semplice un *valore-output* è semplicemente un'*espressione* che determina il valore da far uscire. L'espressione può però essere qualificata da una o due espressioni, separate dai due punti, che controllano l'ampiezza ed il formato del campo in cui il valore è mandato in output. Così abbiamo

valore-output = espressione [*" : " ampiezza-campo* [*" : " lunghezza-parte-frazionaria*]].
ampiezza-campo = espressione.
lunghezza-parte-frazionaria = espressione.

dove la prima espressione, che determina il valore da mandare in output, deve essere di tipo *integer, real, boolean, char* oppure una stringa. L'*ampiezza-campo* e la *lunghezza-parte-frazionaria* debbono essere espressioni che producono risultati positivi e interi, altrimenti si ha un errore.

Quando un valore di output è scritto senza specificare l'ampiezza-campo viene assunto un campo di lunghezza convenzionale. Tale valore convenzionale è definito dalle implementazioni PASCAL in accordo col tipo del valore da mettere in output. In tutte le implementazioni, però, il valore convenzionale stabilito per il tipo *char* è 1. L'ampiezza di campo determina il numero esatto di caratteri scritti sul canale di output. Se il valore effettivo richiede un numero di caratteri minore di quanto specificato, prima del valore stesso viene messo in output un numero appropriato di spazi.

Se il valore effettivo richiede invece un numero di caratteri maggiore di quanto specificato, nel caso di interi o reali l'ampiezza di campo viene aumentata del minimo indispensabile per rappresentare il numero, mentre una stringa è troncata all'ampiezza specificata eliminando i caratteri più a destra.

La terza forma di valore di output, che include una specificazione di lunghezza della parte frazionaria, può essere usata solo per valori reali. Se non è specificata tale lun-

ghezza un valore reale è mandato in output nella forma in virgola mobile, per esempio,

0.976100**E**+03

Se invece è specificata una lunghezza *n* per la parte frazionaria, il valore sarà stampato in output in virgola fissa, con *n* cifre dopo il punto decimale; per esempio con *n*=2 lo stesso valore di cui sopra diventerebbe

976.10

I vantaggi dell'uso di stringhe di caratteri, e di specificazioni di formato, possono essere dedotti dalla Tabella 5.1, che riporta varie istruzioni con gli output che producono.

Istruzione	Output
write (a)	−0.357000000**E**+02
write (a:1:10*)*	−0.357**E**+02
write (a:5:1*)*	−35.7
write ('a = ', a:5:1*)*	*a* = −35.7

Tabella 5.1 Alcune istruzioni di output e loro effetto

Le funzionalità di input ed output descritte in questo capitolo sono in realtà casi particolari di funzionalità più generali fornite dal PASCAL. Le istruzioni *read, readln, write* e *writeln* sono, in effetti, esempi d'uso di *procedure* (vedi il Cap. 7) fornite come standard in tutte le implementazioni PASCAL, ed il loro uso con i canali standard di input ed output è solo un caso particolare delle funzionalità per la gestione di file testo (vedi il Cap. 12).

PROGRAMMA 1 (Calcolo dell'orario di arrivo)

Si richiede un programma che legga tre interi rappresentanti l'orario di lancio di un razzo espresso in ore, minuti e secondi sul quadrante da 24 ore, legga ancora un intero che rappresenti il tempo di volo del razzo in secondi, usi tali valori per calcolare l'orario di ritorno sulla Terra, stampando infine tale valore in una forma leggibile.

Il metodo usato nel progettare il programma in questo come in altri esempi presentati nel libro consiste nel suddividere il problema specificato in una serie di sottoproblemi espressi in un italiano comune e sintetico. Questa stessa tecnica è poi usata per suddividere ciascun sottoproblema in altri sottoproblemi. Il metodo è applicato reiteratamente finché ciascun sottoproblema può essere direttamente e facilmente espresso in PASCAL, ed è spesso chiamato "programmazione per raffinamento incrementale".

Considerando che è buona pratica riscrivere ogni dato letto da un programma (così da poter scoprire un qualsivoglia errore nei dati di input), la soluzione al problema può essere definita informalmente come in (1).

```
begin
    leggi e stampa i dati;
    calcola l'orario di arrivo;                    (1)
    stampa l'orario di arrivo
end
```

La prima azione, *leggi e stampa i dati,* consiste di due azioni:

```
leggi e stampa l'orario di partenza;
leggi e stampa il tempo di volo
```

L'introduzione di tre variabili positive intere *ore, minuti* e *secondi* permette che la prima di queste due azioni sia espressa usando istruzioni *read* e *write.* L'istruzione *read* necessaria è semplicemente

```
read (ore, minuti, secondi)
```

L'istruzione *write* stamperà una stringa *'L'' ORARIO DI PARTENZA E'''* e l'orario di lancio nel formato *oo/mm/ss.* Ciò deve apparire su una linea diversa; inoltre lasceremo una linea bianca tra ciascuna delle linee significative di output prodotte dal programma. Così

```
writeln ('L'' ORARIO DI PARTENZA E''', ore:2,'/', minuti:2,'/', secondi:2);
writeln
```

Notare che per ottenere il formato richiesto è stata usata un'ampiezza di campo 2.

Analogamente, *leggi e stampa il tempo di volo* può essere programmato come una semplice serie di istruzioni di input ed output di una variabile *tempodivolo*:

```
read (tempodivolo);
writeln ('TEMPO DI VOLO =', tempodivolo:7, 'SECONDI');
writeln
```

Occupiamoci ora del sottoproblema *calcola l'orario di arrivo.* Il problema richiede che si trovi l'orario di ritorno sulla Terra. Questo è calcolato mediante le sei istruzioni di assegnazione elencate sotto. Notare che l'ordine delle assegnazioni è importante e che un altro ordine potrebbe produrre risultati sbagliati.

```
secondi := secondi + tempodivolo;
minuti := minuti + secondi div 60;
secondi := secondi mod 60;
ore := ore + minuti div 60;
minuti := minuti mod 60;
ore := ore mod 24
```

Infine, *stampa l'orario di arrivo* consiste nella stampa di un'opportuna didascalia seguita dai valori delle variabili *ore, minuti, secondi* in formato opportuno. L'istruzione relativa è

```
writeln ('ORARIO DI ARRIVO PREVISTO =', ore:2,'/', minuti:2,'/', secondi:2)
```

Si può ora comporre il programma PASCAL finale che consiste di un'intestazione di programma, delle dichiarazioni delle variabili, e di una parte istruzioni. L'intestazione deve indicare che il programma usa entrambi i canali standard di input e di output. Le variabili appartengono tutte al tipo sottointervallo degli interi 0..*maxint*; si osservi che hanno identificatori che ne suggeriscono il significato. La parte istruzioni contiene le istruzioni viste sopra, racchiuse da **begin** ed **end,** e seguite da un punto per segnalare la fine del programma. Il programma risultante e un esempio di output sono mostrati nel Listato 1.

LISTATO 1

```
PROGRAM RAZZO (INPUT,OUTPUT);
(* QUESTO PROGRAMMA LEGGE TRE NUMERI CHE RAPPRESENTANO
   L' ORARIO DI LANCIO DI UN RAZZO ESPRESSO IN ORE,
   MINUTI, SECONDI SU UN QUADRANTE DA 24 ORE. LEGGE
   POI UN QUARTO NUMERO CHE DA' IL TEMPO DI VOLO IN
   SECONDI, E STAMPA L' ORARIO DI ARRIVO PREVISTO*)

VAR ORE, MINUTI, SECONDI, TEMPODIVOLO : 0..MAXINT;

BEGIN
   (* LEGGE E STAMPA I DATI *)
   READ(ORE,MINUTI,SECONDI);
   WRITELN('ORARIO DI LANCIO: ',ORE:2,'/',MINUTI:2,'/',
                                          SECONDI:2);
   WRITELN;
   READ(TEMPODIVOLO);
   WRITELN('TEMPO DI VOLO = ',TEMPODIVOLO:7,' SECONDI');
   WRITELN;

   (* CALCOLO DELL' ORARIO DI ARRIVO *)

   SECONDI:= SECONDI + TEMPODIVOLO;
   MINUTI := MINUTI + SECONDI DIV 60;
   SECONDI:= SECONDI MOD 60;
   ORE    := ORE + MINUTI DIV 60;
   MINUTI := MINUTI MOD 60;
   ORE    := ORE MOD 24;
   (* STAMPA DELL' ORARIO DI ARRIVO *)
   WRITELN('ORARIO DI ARRIVO PREVISTO = ',ORE:2,'/',
                                    MINUTI:2,'/',SECONDI:2);
END.
```

```
ORARIO DI LANCIO:  3/47/32

TEMPO DI VOLO =   45678 SECONDI

ORARIO DI ARRIVO PREVISTO = 16/28/50
```

ESERCIZI

5.1 Date tre linee di dati input

```
12.75        24.7
    -33 E 10
0.075
```

e alcune variabili così dichiarate:

```
X, Y, Z : real;
I, J, K : integer;
c : char;
```

qual è l'effetto dell'esecuzione di ciascuna delle seguenti serie di istruzioni di input?

(a) *read (X, Y, Z)*
(b) *readln (X); readln (Y, Z)*
(c) *readln (X, Y, Z); read (I)*
(d) *readln (I); readln (J); readln (K)*
(e) *read (I, c, J, K)*

5.2 Date le variabili intere *cont, min* e *max,* ed una variabile reale *media,* scrivere una serie di istruzioni di output per stampare i loro valori nella seguente forma:

```
NUMERO      INTERVALLO      MEDIA
  xxx          xx..xx         xx.x
```

5.3 Scrivere un programma che legge una somma di denaro, scritta nella forma

ddd.dd

l'aumenta dell'8%, e stampa la somma originale e quella calcolata. Il vostro programma funzionerebbe se l'input fosse nella forma

ddd:dd

e se no, come lo modifichereste?

5.4 Scrivere un programma che legge il costo di un articolo in vendita e la somma di denaro offerta dal cliente, e stampi:

```
COSTO       $ xxx.xx
OFFERTA     $ xxx.xx
RESTO       $ xxx.xx
```

6.
Strutture di controllo di base

La potenza di un calcolatore consiste nella sua capacità di compiere azioni sequenziali, ripetitive e selettive ad altissima velocità. Il PASCAL permette di esprimere tali azioni per mezzo di una notevole varietà di *istruzioni-strutturate*

istruzione-strutturata = *istruzione-composta*|
istruzione-ripetitiva|
istruzione-condizionale|
istruzione-with.

L'*istruzione-with* è un costrutto dedicato ad una particolare azione, ed il suo uso non viene per ora ulteriormente approfondito, mentre le altre istruzioni strutturate sono tutte spiegate in questo capitolo.

6.1. ISTRUZIONI COMPOSTE

Un'*istruzione-composta* ha la forma

istruzione-composta = **"begin"** *sequenza-istruzioni* **"end"**.
sequenza-istruzioni = *istruzione* { **";"** *istruzione* }.

cioè consiste di una sequenza di istruzioni separate da punto e virgola, precedute da **begin** e seguite da **end.** L'esecuzione di un'*istruzione-composta* comporta semplicemente l'esecuzione delle sue istruzioni componenti, nell'ordine in cui sono scritte.

Ad esempio,

begin $a:=b$; $b:=c$; $c:=a$ **end**

è un'*istruzione-composta* formata da tre istruzioni di assegnazione. Abbiamo già visto esempi di istruzioni composte nei programmi dei precedenti capitoli–la *parte-istruzioni* di ogni programma PASCAL è un'istruzione composta.

La figura 6.1 mostra la sequenza di azioni risultante dall'istruzione composta

begin $S1$; $S2$; $S3$ **end**

nella forma di un diagramma di flusso. Sebbene in questo capitolo usiamo i diagram-

mi di flusso per mostrare il flusso del controllo relativo alle istruzioni strutturate PASCAL, ne sconsigliamo l'uso come strumento di progetto di programmi, poiché essi non consentono di affrontare la costruzione di programmi in modo sufficientemente sistematico.

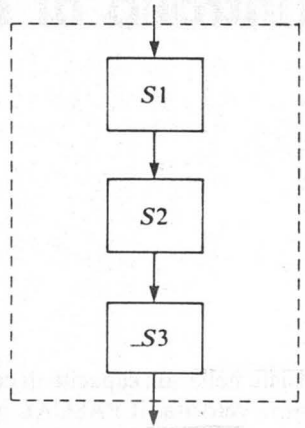


Fig. 6.1 Esecuzione di un'istruzione composta

Il ruolo dei delimitatori dell'istruzione composta, **begin** ed **end,** consiste nel ridurre ad una singola istruzione sintattica la sequenza di istruzioni che essi racchiudono. Perciò la *parte-istruzioni* di un programma è, da un punto di vista sintattico, una singola istruzione. Più spesso i delimitatori si usano per collocare la sequenza di istruzioni che racchiudono sotto il controllo di una istruzione PASCAL ripetitiva o condizionale, come vedremo nei prossimi paragrafi.

All'interno di un'istruzione composta il punto e virgola fa da separatore delle istruzioni, perciò non occorre dopo l'ultima istruzione. Se invece viene messo, si assume l'esistenza di una istruzione nulla, o "vuota", immediatamente prima del simbolo **end.** Una *istruzione-vuota* è un'istruzione che non contiene nessun simbolo e che non ha effetto. Quindi

begin $a := b$; ; $b := c$; $c := a$; **end**

è un'istruzione composta corretta, formata da un'istruzione di assegnazione, un'istruzione vuota, altre due istruzioni di assegnazione e un'ultima istruzione vuota.

6.2. ISTRUZIONI RIPETITIVE

Nei programmi una classe importante di azioni è costituita dai cicli, che permettono la ripetizione di alcune istruzioni, o gruppi di istruzioni, di solito controllata da qualche condizione di terminazione.

Il PASCAL fornisce tre costrutti di ripetizione, che soddisfano le necessità di costruzione di cicli nella maggior parte delle situazioni riscontrabili nei programmi, e sono

istruzione-ripetitiva = *istruzione-while*|
istruzione-repeat|
istruzione-for.|

6.2.1. L'*istruzione-while*

La forma sintattica dell'*istruzione-while* è

istruzione-while = **"while"** *espressione* **"do"** *istruzione.*

L'*espressione* deve produrre un valore di tipo *boolean.* L'espressione viene valutata ripetutamente e, se rimane vera, l'*istruzione* viene eseguita di seguito alla valutazione dell'espressione. La ripetizione termina non appena il valore dell'espressione diventa falso.

Il flusso del controllo per l'istruzione

while *e* **do** *S*

è pertanto quello indicato in Fig. 6.2.

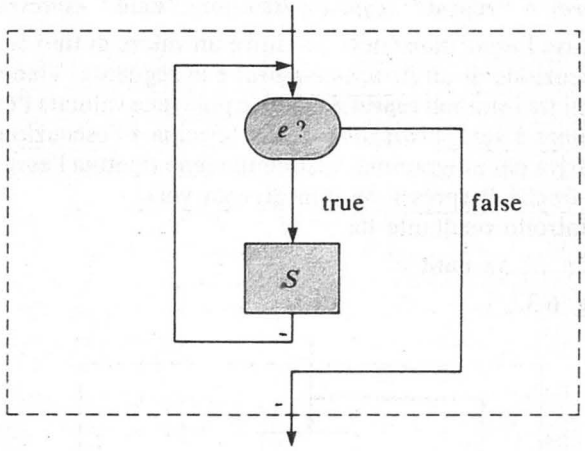


Fig. 6.2 Esecuzione di un'istruzione-while

I seguenti sono esempi di istruzioni-while:

```
(a) while c = ' ' do begin cont := cont + 1; read (c) end
(b) while abs (x - a / x) > 1E-6 do x := 0.5 * (x + a / x)
```

Si noti che se l'espressione, inizialmente, è falsa, l'istruzione non viene eseguita per niente. Oltre a ciò si noti l'uso dei delimitatori dell'istruzione composta, **begin** ed **end,** nell'esempio (a), per inserire una sequenza di istruzioni sotto il controllo dell'*istruzione-while.*

Il programma (1) legge due interi positivi e divide il primo per il secondo, usando solo le operazioni di addizione e di sottrazione.

```
program divisione (input, output);
var x, y, quoziente, resto : 0.. maxint;
begin
    read (x, y);
```

```
    resto := x;
    quoziente := 0;
    while resto > = y do
    begin                                                    (1)
        quoziente := quoziente + 1;
        resto := resto − y
    end;
    writeln (x, ' diviso per ', y, ' uguale ', quoziente, ', resto ', resto)
end.
```

6.2.2. L'*istruzione-repeat*

Una *istruzione-repeat* assume la forma

istruzione-repeat = "**repeat**" *sequenza-istruzioni* "**until**" *espressione.*

Anche in questo caso l'*espressione* deve produrre un valore di tipo *boolean.* L'azione provocata dall'esecuzione di un'*istruzione-repeat* è la seguente. Viene eseguita la sequenza di istruzioni tra i simboli **repeat** ed **until,** e poi viene valutata l'espressione booleana. Se il suo valore è vero, l'*istruzione-repeat* termina e l'esecuzione continua con l'istruzione successiva del programma, altrimenti viene ripetuta l'esecuzione della *sequenza-istruzioni,* finché l'espressione non diventa vera.

Il flusso del controllo risultante da

repeat *S*1; *S*2; ...; *Sn* **until** *e*

è mostrato in Fig. 6.3.

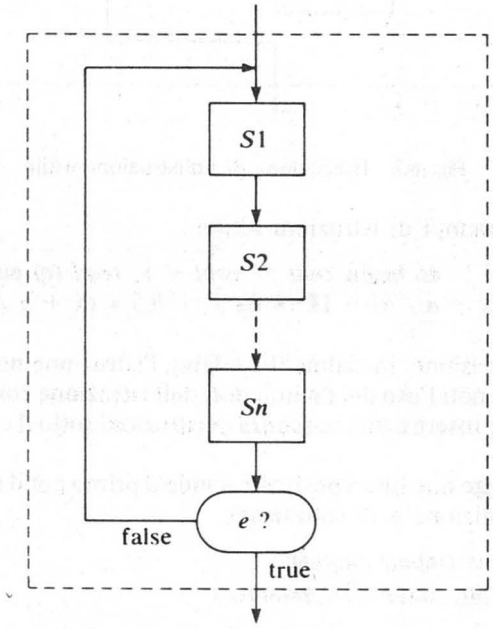


Fig. 6.3 Esecuzione di un'istruzione-repeat

I seguenti sono esempi di istruzione-repeat:

(a) **repeat** *cont* := *cont* + 1; *read (c)* **until** *c* < > ' '
(b) **repeat** *x* := 0.5 * (*x* + *a* / *x*) **until** *abs* (*x* − *a* / *x*) < = 1**E**−6

Si noti che il corpo del ciclo non deve avere la forma di un'*istruzione-composta* (con l'uso di **begin** ed **end**), perché la sintassi stabilisce che il corpo del ciclo è la sequenza di istruzioni delimitata dai simboli **repeat** ed **until**.

La differenza principale tra l'istruzione-repeat e l'istruzione-while è che nella prima il corpo del ciclo (cioè le istruzioni che devono essere eseguite ripetitivamente) viene eseguito almeno una volta, prima della valutazione della condizione di terminazione, mentre nella seconda la condizione di terminazione viene valutata prima, cosicché il corpo del ciclo può non essere affatto eseguito. Qualcuno osserva che questa differenza non giustifica due costrutti distinti per il ciclo e che uno solo di essi, l'istruzione-while, è sufficiente a descrivere entrambe le situazioni. Comunque, oltre a rendere esplicito il fatto che il corpo del ciclo viene eseguito almeno una volta, l'istruzione-repeat è utile per evitare potenziali errori di programmazione. Nelle situazioni di ciclo repeat capita spesso che i valori delle variabili che influiscono sulla condizione di terminazione siano indeterminati o sbagliati prima dell'esecuzione iniziale del corpo di un ciclo. Per esprimere il ciclo con un'istruzione-while il programmatore deve fare prima delle assegnazioni artificiose a queste variabili in modo da rendere falsa la condizione di terminazione, un lavoro che troppo spesso dà luogo ad errori, o addirittura è del tutto trascurato. Per esempio,

repeat *read (ch)* **until** *ch* = '.'

dovrebbe essere espresso

ch := ';'; **while** *ch* <> '.' **do** *read (ch)*

in cui a *ch* viene inizialmente assegnato un qualche valore diverso da '.'.

6.2.3 L'*istruzione-for*

L'*istruzione-while* e l'*istruzione-repeat* sono i costrutti più importanti per esprimere le ripetizioni riscontrabili nella maggior parte dei programmi. Comunque, per operazioni che devono essere ripetute un determinato numero di volte si può usare l'*istruzione-for.*

Un'*istruzione-for* prende la forma

istruzione-for = **"for"** *identificatore-variabile* **":="** *espressione-iniziale* (**"to"**|**"downto"**)
espressione-finale **"do"** *istruzione.*
espressione-iniziale = *espressione.*
espressione-finale = *espressione.*

L'*identificatore-variabile* prende il nome di *variabile di controllo,* e deve essere dichiarato come variabile di un tipo ordinale nel blocco immediatamente contenente l'*istruzione-for.* L'*espressione-iniziale* e l'*espressione-finale* devono dare valori dello stesso tipo ordinale della variabile di controllo.

Il valore iniziale e quello finale determinano una successione di valori crescenti (se

si usa **to**) o decrescenti (se invece si usa **downto**). L'azione dell'*istruzione-for* è di assegnare ciascuno di questi valori, uno alla volta, alla variabile di controllo ed eseguire, dopo ogni assegnazione, l'istruzione che si trova dopo il simbolo **do**. Se il valore iniziale è più grande del valore finale nel caso di **to**, o più piccolo di quello finale, nel caso di **downto**, l'istruzione non viene affatto eseguita.

Nell'interpretare questa definizione sono da sottolineare i seguenti punti:

(a) I valori iniziale e finale determinati rispettivamente dall'espressione iniziale e finale vengono valutati soltanto una volta, all'entrata dell'*istruzione-for*. Essi non vengono rivalutati ad ogni ciclo, e le variazioni delle variabili contenute in queste espressioni che avvengono durante l'esecuzione ripetitiva non influenzano in alcun modo la sequenza di valori assunti dalla variabile di controllo.

(b) Il valore della variabile di controllo viene modificato implicitamente ad ogni ciclo della ripetizione, e nessuna azione volta a modificare il suo valore può apparire all'interno del corpo del *ciclo-for* o all'interno di qualsiasi procedura o funzione (vedi Cap. 7) che *potrebbe* essere chiamata dal corpo del ciclo.

(c) Dopo la conclusione di un'*istruzione-for* il valore della variabile di controllo non è definito, e nei programmi non si dovrebbe fare alcuna assunzione sul suo valore.

Esempi:

```
1. for j:= lineescritte to dimensionepagina do writeln
2. for j:= m downto n do writeln (j, j * j * j)
3. for giorno:= lunedi to venerdi do
   begin
       read (orelavorate); oretotali := oretotali + orelavorate
   end
```

I quattro programmi (2) - (5) realizzano azioni simili, nel senso che ciascuno di essi legge una serie di interi positivi e calcola la loro somma, ma, a causa di condizioni leggermente diverse nei vari casi, in essi vengono usate diverse strutture ripetitive.

(a) Nel caso (2) la lunghezza della sequenza non è nota, ma l'ultimo intero è seguito da un intero negativo. È possibile avere una sequenza senza interi positivi.

```
program somma1 (input, output);
var i, somma : integer;
begin
    somma := 0;
    read (i);
    while i > = 0 do
    begin
        somma := somma + i;                    (2)
        read (i)
    end;
    writeln (somma)
end.
```

(b) Nel caso (3) valgono le stesse condizioni del caso (2), ad eccezione del fatto che si garantisce che c'è almeno un intero positivo nella sequenza.

```
program somma2 (input, output);
var i, somma : integer;
begin
    somma := 0;
    read (i);
    repeat
        somma := somma + i;                    (3)
        read (i)
    until i < 0;
    writeln (somma)
end.
```

(c) Nel caso (4) la sequenza è preceduta da un intero, che specifica il numero di interi nella sequenza.

```
program somma3 (input, output);
var lunghezza, k, i, somma : integer;
begin
    read (lunghezza);
    somma := 0;
    for k := 1 to lunghezza do                 (4)
    begin
        read (i);
        somma := somma + i
    end;
    writeln (somma)
end.
```

(d) Nel caso (5) la sequenza contiene un intero per ogni linea di input. L'uso di *readln* per leggere i valori fa sì che il riconoscimento della fine della sequenza di input si possa controllare per mezzo della funzione *eof*. Notare che *eof* non si può usare altrimenti durante l'input numerico, perché non sono stati letti gli spazi dopo l'ultimo numero.

```
program somma4 (input, output);
var i, somma : integer;
begin
    somma := 0;
    while not eof (input) do                   (5)
    begin
        readln (i);
        somma := somma + i
    end
    writeln (somma)
end.
```

Le istruzioni controllate da un istruzione ripetitiva possono essere di qualsiasi forma ammessa tra quelle definite all'inizio di questo capitolo. In particolare, il corpo dell'istruzione ripetitiva può contenere un'altra istruzione ripetitiva, nel qual caso le istruzioni ripetitive si dicono *annidate*. Quest'annidamento delle istruzioni ripetitive viene illustrato in (6), che legge un intero n e calcola la somma della serie

$$1^1 + 2^2 + 3^3 + \ldots + n^n.$$

```
program sommadipotenze (input, output);
var n, x, potenza, i, somma : 0 ..maxint;
begin
    read (n);
    somma := 0;
    for x := 1 to n do
    begin                                                    (6)
        potenza := 1;
        for i := 1 to x do potenza := potenza * x;
        somma := somma + potenza
    end;
    writeln (somma)
end.
```

Come secondo esempio dell'annidamento di cicli si consideri (7), che legge il contenuto della stringa di input e lo fornisce in uscita, mantenendo la stessa struttura delle linee dell'input.

```
program copia (input, output);
var  c : char;
begin
    while not eof (input) do
    begin { copia una linea }
        while not eoln (input) do
        begin
            read (c);
            write (c)                                        (7)
        end;
        readln;
        writeln
    end
end.
```

PROGRAMMA 2 (Tabulazione di voti d'esame)

Una classe è formata da 50 studenti, ciascuno dei quali studia 5 materie. I voti ottenuti da ogni studente in ciascuna materia vengono dati in input ad un calcolatore, in modo che ad ogni linea corrispondono i dati di uno studente. Ogni linea contiene il nome dello studente, che è lungo al più 30 caratteri e termina con un

punto, seguito da 5 voti da 0 a 10 separati da spazi; per esempio

MARY SMITH. 7 7 10 6 4
HOWARD BLACKBURN. 4 7 5 4 5

Si desidera un programma che legge questo input e crea una tabella contenente la valutazione complessiva degli studenti; in essa devono comparire il nome di ogni studente, la sua votazione totale incolonnata, e una riga di istogramma, che dia una misura della valutazione. Per esempio, gli input mostrati potrebbero produrre i seguenti output:

MARY SMITH. 34 **********
HOWARD BLACKBURN. 25 **********

Il programma ha chiaramente la struttura di un ciclo, che viene eseguito una volta per ogni studente:

```
for studente := 1 to 50 do
    leggi e stampa i dati di uno studente
```

Il processo *leggi e stampa i dati di uno studente* si può spezzare in una sequenza di passi più semplici, come in (8).

```
begin
    leggi e stampa il nome;
    leggi e stampa i voti;                                    (8)
    allinea l'input e l'output per lo studente successivo
end
```

L'azione *leggi e stampa il nome* è visibilmente un semplice ciclo di lettura e stampa, carattere per carattere, che termina con la lettura di un punto, come indicato in (9).

```
repeat
    read (caratteresuccessivo);                               (9)
    write (caratteresuccessivo)
until caratteresuccessivo = '.'
```

L'azione *leggi e stampa i voti* si può spezzare nella sequenza (10).

```
leggi i voti e calcola il totale;
stampa il voto totale;                                        (10)
stampa la riga dell'istogramma
```

Quindi *leggi i voti e calcola il totale* si può scrivere come indicato in (11).

```
totale := 0;
for materia := 1 to 5 do
begin
    read (voto);                                              (11)
    totale := totale + voto
end
```

Però, per stampare il voto totale allineato in una certa colonna, dobbiamo sapere quante posizioni di stampa occupa il nome dello studente. Perciò modifichiamo il ciclo *leggi e stampa il nome,* in modo da stampare un numero di spazi tale che risultino stampati comunque 30 caratteri, come descritto in (12).

```
lunghezzanome := 0;
repeat
    read (caratteresuccessivo);
    write (caratteresuccessivo);
    lunghezzanome := lunghezzanome + 1
until caratteresuccessivo = '.';                    (12)
while lunghezzanome < 30 do
begin
    write (' ');
    lunghezzanome := lunghezzanome + 1
end
```

In questo modo, per stampare il totale con l'ultima cifra nella 40-esima posizione, scriviamo

```
write (totale : 10)
```

La stampa di una riga dell'istogramma del profitto degli studenti consiste nella scrittura di una sequenza di caratteri identici, * ad esempio, la cui lunghezza sia proporzionale al voto totale raggiunto. Siccome il voto totale, così com'è, può variare tra 0 e 50, lo riduciamo di un fattore di scala 2/5, per ottenere in stampa una riga di lunghezza adeguata. Per rendere la stampa più leggibile, il primo '*' dell'istogramma viene separato con due spazi dal voto totale che lo precede. La stampa della riga dell'istogramma si può, perciò, scrivere

```
write ('  ');
for stella := 1 to round (0.4 * totale) do write ('*')
```

Alla fine, si riallinea la posizione dell'input e dell'output per lo studente successivo

```
readln;
writeln
```

Siamo ora in grado di assemblare i vari frammenti nel programma finale, con le opportune dichiarazioni delle variabili introdotte. Il programma risultante, ed un esempio dell'output che produce, sono presentati nel Listato 2.

LISTATO 2

```
PROGRAM VOTI (INPUT,OUTPUT);

VAR STUDENTE            : 1..50;
    LUNGHEZZANOME       : 0..30;
    PROSSIMOCARATTERE   : CHAR;
    TOTALE              : 0..50;
    MATERIA             : 1..5;
```

```
    VOTO                : 0..10;
    STELLA              : 0..20;

BEGIN
   FOR STUDENTE:= 1 TO 50 DO
   BEGIN
      (* LEGGI E STAMPA NOME *)
      LUNGHEZZANOME:= 0;
      REPEAT
      READ(PROSSIMOCARATTERE);
      WRITE(PROSSIMOCARATTERE);
      LUNGHEZZANOME:= LUNGHEZZANOME 9+ 1
      UNTIL PROSSIMOCARATTERE = ',';
      WHILE LUNGHEZZANOME < 30 DO
      BEGIN
         WRITE(' ');
         LUNGHEZZANOME:= LUNGHEZZANOME + 1
      END;
      (* LEGGI E STAMPA VOTI *)
      (* LEGGI E SOMMA VOTI *)
      TOTALE:= 0;
      FOR MATERIA:= 1 TO 5 DO
      BEGIN
         READ(VOTO);
         TOTALE:= TOTALE + VOTO
      END;
      (* STAMPA IL TOTALE ALLINEANDOLO *)
      WRITE(TOTALE:10);
      (* STAMPA LINEA DI ISTOGRAMMA *)
      WRITE(' ');
      FOR STELLA:= 1 TO ROUND(0.4 * TOTALE) DO
         WRITE('*');
      (* ALLINEA INPUT ED OUTPUT *)
      READLN;
      WRITELN
   END
END.
```

```
ALAN LEWIN.                    22  *********
BETTY CALLAGHAN.               19  ********
CHRIS FARMER.                  17  *******
ERNEST ATKINS.                 28  ***********
```

6.3. ISTRUZIONI CONDIZIONALI

Spesso l'esecuzione di una istruzione deve dipendere dal verificarsi di una condizione; oppure, ad un certo punto, si deve scegliere, in base ad una condizione, di eseguire una tra varie istruzioni possibili. Il PASCAL offre a questo scopo due forme di istruzione

istruzione-condizionale = istruzione-if | istruzione-case.

6.3.1. L'*istruzione-if*

L'*istruzione-if* consente l'esecuzione condizionale di una istruzione, oppure la scelta tra l'esecuzione di due istruzioni. Le due forme corrispondenti sono

istruzione-if = **"if"** *espressione* **"then"** *istruzione* [**"else"** *istruzione*]

In entrambe le forme l'*espressione* deve produrre un risultato booleano.

L'azione corrispondente alla forma più breve consiste nella valutazione dell'*espressione* e, se e solo se il suo valore è *true,* nell'esecuzione dell'*istruzione.* In entrambi i casi l'esecuzione continua con l'istruzione successiva all'*istruzione-if.*

Nella forma più lunga, viene valutata l'*espressione* e, se il suo valore è *true,* viene eseguita la prima *istruzione* (cioè l'istruzione dopo il simbolo **"then"**), altrimenti la seconda (cioè quella dopo il simbolo **"else"**). Di nuovo, in entrambi i casi l'esecuzione continua con l'istruzione successiva all'*istruzione-if.* Queste strutture di controllo sono illustrate in Fig. 6.4.

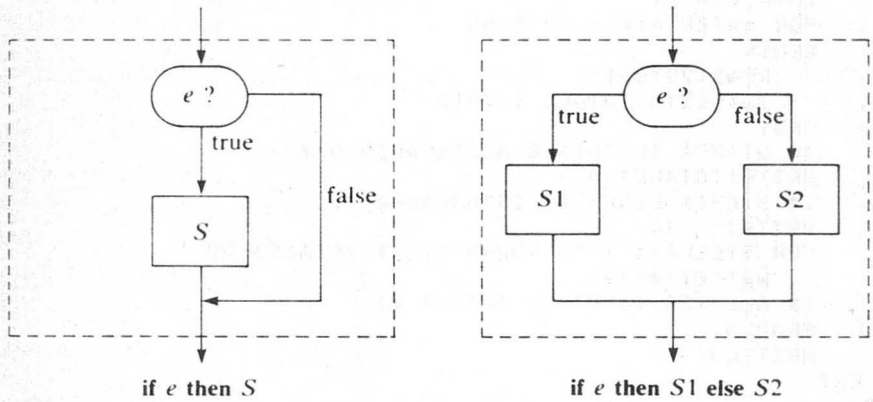


Fig. 6.4. Strutture di controllo delle istruzioni-if.

Quando si usa la forma **else** bisogna stare attenti a non mettere un punto e virgola prima di **else**, perché questo comporterebbe la conclusione in quel punto dell'*istruzione-if,* cioè

if *e* **then** *S*1; **else** *S*2

è sbagliata perché "**else** *S*2" non è un'istruzione legale in PASCAL.

Esempi di *istruzioni-if* sono mostrati in (13)-(16).

```
if ch = '?' then domanda := true                                    (13)

if eof (input)
   then writeln (' dati tutti letti')                               (14)
   else read (ch)
```

```
if questomese = Dic
    then begin
            questomese := Gen;
            questoanno := questoanno + 1
         end
    else questomese := succ (questomese)
```

(15)

```
if sqr (b) < 4 * a * c
  then write ('equazione con radici complesse')
  else begin
            d := sqrt (sqr (b) - 4 * a * c);
            writeln ((-b + d) / (2 * a), (-b - d) /(2 * a))
       end
```

(16)

Si noti che una sequenza di istruzioni che deve essere controllata da un'*istruzione-if* si può raggruppare in un'unica istruzione, usando i delimitatori **begin** ed **end** per formare un'*istruzione-composta.*

Il programma mostrato in (17) legge due interi positivi e calcola il loro massimo comun divisore.

```
program massimocomundivisore (input, output);
var a, b : 1.. maxint;
begin
    read (a, b);
    while a <> b do
        if a < b
            then a := a - b
            else b := b - a;
    writeln (a)
end.
```

(17)

Le istruzioni la cui esecuzione è controllata da un'*istruzione-if* possono includere altre *istruzioni-if,* formando istruzioni condizionali più complesse. In tal caso le istruzioni si dicono *annidate* ; per esempio

```
if x < 0
    then segno := true
    else if x > 0
            then segno := false
            else write ('il valore e " zero')
```

Un'*istruzione-if* annidata della forma

if *e*1 **then if** *e*2 **then** *S*1 **else** *S*2

può apparire ambigua perché non è chiaro se l'esecuzione di *S*2 è controllata dal valore di *e*1 o di *e*2. L'ambiguità viene risolta considerando questa istruzione equivalente a

```
if e1
    then begin
            if e2
              then S1
              else S2
         end
```

In questo modo l'esecuzione di *S*2 risulta dipendente dal valore di *e*2. Se, al contrario, si vuole che l'esecuzione di *S*2 sia dipendente dal valore di *e*1, bisogna scrivere

```
if e1
  then begin
          if e2 then S1
       end
  else S2
```

Il programma (18) legge un valore intero (che si assume positivo e minore di 100) e stampa uno dei seguenti messaggi:

(a) la parola *'buzz'* se il valore contiene la cifra 7, altrimenti
(b) la parola *'buzz-buzz'* se il valore è un multiplo di 7, altrimenti
(c) il valore stesso

```
program buzzbuzz (input, output);
var i : 1..100;
begin
    read (i);
    if (i div 10 = 7) or (i mod 10 = 7)                    (18)
       then writeln ('buzz')
       else if i mod 7 = 0
               then writeln ('buzz-buzz')
               else writeln (i)
end.
```

PROGRAMMA 3 (Analisi di un triangolo)

Si richiede un programma che legge tre interi positivi, che rappresentano le lunghezze dei tre lati di un triangolo, in ordine crescente, e stampa i dati di input, e una delle seguenti descrizioni del tipo di triangolo da essi definito:

non è un triangolo
triangolo equilatero
triangolo isoscele
triangolo scaleno

ed infine l'area del triangolo (se si tratta di un triangolo).

In prima approssimazione il programma richiesto si può scomporre in due passi:

begin
leggi e stampa la lunghezza dei lati;
determina e stampa le informazioni sul triangolo
end

Il passo *leggi e stampa la lunghezza dei lati* si può esprimere immediatamente con istruzioni *read* e *write* che usano tre variabili intere *a, b, c,* per rappresentare la lunghezza dei tre lati in ordine crescente.

Il passo *determina e stampa le informazioni sul triangolo* si può spezzare così

if *è triangolo*
then *determina e stampa il tipo e l'area*
else *write ('non e " un triangolo')*

Tre segmenti formano un triangolo se e solo se la lunghezza del segmento più lungo è minore della somma delle lunghezze degli altri due segmenti. Siccome *a, b, c,* rappresentano i lati in ordine crescente di lunghezza la condizione *è un triangolo* è esprimibile immediatamente

$$a + b > c$$

Determinare la natura del triangolo è ancora facilmente esprimibile in termini di queste tre lunghezze ordinate:

if $a = c$
then *il triangolo è equilatero*
else if $(a = b)$ **or** $(b = c)$
then *il triangolo è isoscele*
else *il triangolo è scaleno*

L'area di un triangolo di lati *a, b, c* è data dalla formula

$$area = \sqrt{s\,(s-a)\,(s-b)\,(s-c)}$$

in cui *s* è la semisomma dei lati *a, b, c,* ed è perciò facile da programmare, usando valori reali per s e per l'area, in quanto possono avere parti frazionarie.

Possiamo a questo punto costruire il programma finale, che è riportato nel Listato 3, con un esempio dell'output prodotto.

LISTATO 3

```
PROGRAM TRIANGOLI(INPUT,OUTPUT);

VAR   A,B,C   : 1..MAXINT;
      S,AREA  : REAL;

BEGIN

   (* LEGGI E SCRIVI I LATI IN ORDINE CRESCENTE *)

   READ(A,B,C);
```

```
   WRITE(A,B,C,' ... ');
   IF A + B > C
      THEN BEGIN

                (* DETERMINA LA NATURA DEL TRIANGOLO *)

                IF A = C
                   THEN WRITE('UN TRIANGOLO EQUILATERO')
                   ELSE IF (A = B) OR (B = C)
                           THEN WRITE('UN TRIANGOLO ISOSCELE')
                           ELSE WRITE('UN TRIANGOLO SCALENO');

                (* DETERMINA L' AREA *)

                S:= 0.5 * (A + B + C);
                AREA:= SQRT(S * (S - A) * (S - B) * (S - C));
                WRITELN(' DI AREA',AREA:8:2)
           END
      ELSE WRITELN('NON E'' UN TRIANGOLO')
END.
```

```
      7     7     9 ... UN TRIANGOLO ISOSCELE DI AREA   24.13
```

6.3.2. L'*istruzione-case*

Una forma complessa di selezione, che capita frequentemente nei programmi, e che perciò richiede particolare attenzione, è la *selezione* di una azione tra molte, in base al valore di una certa espressione. Questo si potrebbe esprimere per mezzo di un'*istruzione-if* annidata: ad esempio

```
if ch = 'I'
    then n := 1
    else if ch = 'V'
            then n := 5
            else if ch = 'X'
                    then n := 10
                    else if ch = 'L'
                            then n := 50
```

Un modo più elegante di esprimere in PASCAL un'azione di questo genere è di usare l'*istruzione-case,* che ha la seguente sintassi:

istruzione-case = "**case**" *espressione* "**of**" *corpo-case* {"," *corpo-case* } ["; "] "**end**".
corpo-case = *lista-etichette-case* ":" *istruzione*.
lista-etichette-case = *costante* { ";" *costante* }.

L'*espressione* che segue il simbolo **case** prende il nome di *selettore,* e deve dare un valore di tipo ordinale. Le costanti che si trovano davanti all'*istruzione* nei vari *corpi-case* si chiamano *etichette di case* e devono essere dello stesso tipo non strutturato del selettore.

L'azione di un'*istruzione-case* consiste nella valutazione dell'*espressione* selettore, e poi nell'esecuzione dell'*istruzione* etichettata con il valore risultante. Dopo di che l'esecuzione procede con l'istruzione che segue l'*istruzione-case.* Se nessuna *etichetta-case* corrisponde al valore del selettore, si ha un errore.

La Fig. 6.5 mostra la forma generale dell'*istruzione-case* e la risultante struttura di controllo.

```
case e of
c1     : S1;
c2,c3  : S2;
c7     :   ;
c5     : S3
end
```

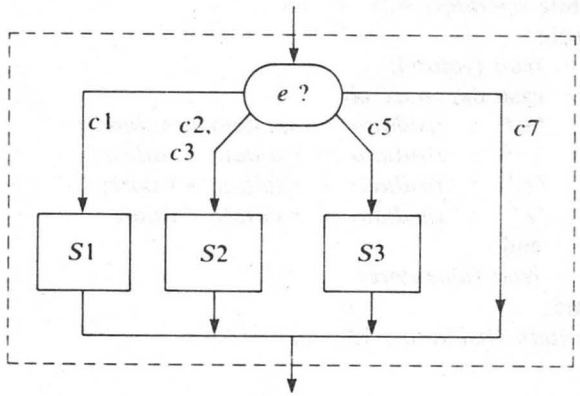


Fig. 6.5 Struttura dell'istruzione-case

L'istruzione considerata all'inizio di questo sottoparagrafo si può ora riscrivere come in (19).

```
case ch of
'I' : n := 1;
'V' : n := 5;
'X' : n := 10;                                    (19)
'L' : n := 50
end
```

Si noti che due o più etichette-case possono essere associate con la stessa istruzione, nel qual caso sono separate da virgole (e non occorre che siano scritte secondo un ordine particolare); ad esempio

```
4, 7, 2, 1, 15 : x := y + 2
```

Anche l'ordine delle istruzioni etichettate all'interno di una istruzione-case è irrilevante. Ciascun valore che può assumere l'espressione selettore deve apparire una ed una sola volta nell'*istruzione-case* come etichetta. Se nessuna azione deve essere realizzata in corrispondenza di qualche valore, a questi va associata un'istruzione vuota.

Il programma (20) simula il comportamento di un calcolatore tascabile. Legge infatti una sequenza di valori numerici separati da operatori aritmetici (+, −, *, /) e calcola

il valore dell'espressione definita da questa sequenza. Il risultato del calcolo è mandato in output quando viene letto l'operatore '='. Così, l'input 7 + 5 * 3 / 8 = produce il risultato 4.5000.

```
program calcolatoretascabile (input, output);
var risultato, valore : real;
    operatore : char;
begin
    read (risultato);
    read (operatore);
    while operatore <> '=' do
    begin
        read (valore);                                          (20)
        case operatore of
        '+' : risultato := risultato + valore;
        '-' : risultato := risultato - valore;
        '*' : risultato := risultato * valore;
        '/' : risultato := risultato / valore
        end;
        read (operatore)
    end;
    writeln (risultato : 12 : 4)
end.
```

PROGRAMMA 4 (Calcolo della data di domani)

Costruire un programma che, dati tre interi i cui valori rappresentano un giorno compreso tra il 1 gennaio 1900 ed il 31 dicembre 1999, fornisca in output i valori che rappresentano il giorno seguente.

I dati di input sono tre interi che rappresentano il giorno, il mese e l'anno di una data; per esempio, 2 12 1978 rappresenta il 2 dicembre 1978. In (21) è indicata la struttura del programma.

```
begin
    leggi la data;
    metti in output la data;                                    (21)
    aggiorna la data a quella del giorno successivo;
    metti in output la data
end.
```

La data si può rappresentare con tre variabili

giorno, una variabile del tipo intervallo 1..31;
mese, una variabile del tipo intervallo 1..12;
anno, una variabile del tipo intervallo 1900..2000.

L'operazione *leggi la data* è semplicemente

```
read (giorno, mese, anno)
```

L'output di questa data potrebbe avere la forma

```
write ('giorno seguente a', giorno : 2, '/', mese : 3, '/', anno : 4)
```

Esprimendo astrattamente l'operazione di aggiornamento come

```
if ultimogiornodelmese
  then primogiornodelmesesuccessivo
  else giornosuccessivodiquestomese
```

abbiamo bisogno, prima di tutto, di trovare il numero dei giorni del mese denotato dal valore di *mese*. Se si introduce una variabile *giornidelmese*, questo può essere calcolato usando un'*istruzione-case*.

```
case mese of
1,3,5,7,8,10,12 : giornidelmese := 31;
4,6,9,11 : giornidelmese := 30;
2 : if (anno mod 4 = 0) and (anno <> 1900)
      then giornidelmese := 29
      else giornidelmese := 28
end
```

L'operazione di aggiornamento diventa

```
if giorno = giornidelmese
  then begin
         giorno := 1;
         if mese = 12
           then begin
                  mese := 1;
                  anno := anno + 1
                end
           else mese := mese + 1
       end
  else giorno := giorno + 1
```

L'output della nuova data è

```
writeln ('e' ' ', giorno : 2, '/', mese : 3, '/', anno : 4)
```

Il programma risultante, ed un esempio dell'output che produce, sono riportati nel Listato 4.

LISTATO 4

```
PROGRAM GIORNOSUCCESSIVO(INPUT,OUTPUT);

VAR   GIORNO          : 1..31;
      MESE            : 1..12;
      ANNO            : 1900..2000;
      GIORNIDELMESE   : 28..31;
```

```
BEGIN
   READ(GIORNO,MESE,ANNO);
   WRITE('IL GIORNO SEGUENTE',GIORNO:3,'/',MESE:2,'/',ANNO:4);

   (* TROVA IL NUMERO DI GIORNI DEL MESE *)

   CASE MESE OF
      1,3,5,7,8,10,12 : GIORNIDELMESE:= 31;
          4, 6, 9, 11 : GIORNIDELMESE:= 30;
                    2 : IF (ANNO MOD 4 = 0) AND (ANNO <> 1900)
                           THEN GIORNIDELMESE:= 29
                           ELSE GIORNIDELMESE:= 28
      END;

   (* AGGIORNA *)

   IF GIORNO = GIORNIDELMESE
      THEN BEGIN
              GIORNO:= 1;
              IF MESE = 12
                 THEN BEGIN
                         MESE:= 1;
                         ANNO:= ANNO + 1
                      END
                 ELSE MESE:= MESE + 1
           END
      ELSE GIORNO:= GIORNO + 1;
   WRITELN(' E''',GIORNO:3,'/',MESE:2,'/',ANNO:4)
END.


IL GIORNO SEGUENTE 31/12/1982 E'  1/ 1/1983
```

ESERCIZI

6.1 Scrivere un programma che legge un intero positivo N e tabula il fattoriale dei numeri da 1 a N nella forma

I	1	2	3	4	...	*N*
FATTORIALE (I)	1	2	6	24	...	

{ *fattoriale (i)* = 1 * 2 * 3 * ... * *(i − 1)* * i }

Riscrivere il programma per tabulare tutti i fattoriali il cui valore è minore del massimo valore intero ammissibile per la vostra implementazione del PASCAL (cioè *maxint*).

6.2 Una società di credito edilizio presta denaro agli acquirenti di case, che poi pagano mensilmente l'uno per cento sulla somma avuta in prestito. Questo pagamento copre sia la restituzione del capitale che gli interessi dovuti, i quali ammontano all'otto per cento per anno, calcolato mensilmente. Scrivere un programma che legge l'ammontare di un prestito e tabula la serie di pagamenti richiesti, in una tabella di quattro colonne in cui sono riportati il numero del pagamento, l'interesse dovuto per quel mese, la restituzione del capitale per quel mese, ed il capitale ancora da restituire. La tabella dovrebbe anche contenere l'indicazione relativa al pagamento finale non standard richiesto per completare la restituzione.

6.3 Scrivere un programma che stampi la piramide di cifre in Fig. 6.6,

```
1
121
12321
1234321
123454321
12345654321
1234567654321
123456787654321
12345678987654321
```

Fig. 6.6

6.4 (a) Modificare il Programma 1 in modo tale che, se l'orario di arrivo previsto non è nello stesso giorno del decollo, l'output assuma la seguente forma:

ORARIO DI ARRIVO PREVISTO = hh/mm/ss DOPO *d* GIORNI

Se l'arrivo è nello stesso giorno del decollo l'output deve rimanere invariato.

(b) Modificare ulteriormente il programma in modo che se l'arrivo è il giorno successivo al decollo, l'output prenda prenda la forma

ORARIO DI ARRIVO PREVISTO = hh/mm/ss
IL GIORNO SUCCESSIVO

6.5 Alcune lettere sono formate da sole linee rette, per esempio A,E,F,H,..., mentre altre richiedono linea curve, per esempio B,C,D,G,.... Scrivere un pogramma che legge e stampa una riga di un testo, sostituendo tutte le lettere che contengono tratti curvi con un asterisco. Scrivere il programma usando

(a) un'istruzione-if;
(b) un'istruzione-case.

6.6 Scrivere un programma che legge e stampa un testo e determina il numero di righe, frasi e parole in esso presenti. Assumere che

(a) ogni frase finisce con un punto, e che il punto può essere usato solamente per questo scopo;
(b) le parole sono formate soltanto da lettere.

7.
Procedure e funzioni

7.1 LA NOZIONE DI PROCEDURA

In questo libro il metodo usato nella costruzione dei programmi consiste nel suddividere il problema iniziale in una serie di problemi più semplici. Questa suddivisione è descritta usando una combinazione di istruzioni strutturate PASCAL e di frasi in italiano. Per esempio, in una prima fase del suo sviluppo il Programma 2 era stato espresso come:

```
for studente := 1 to 50 do
    leggi e stampa i dati di uno studente
```

Il processo *leggi e stampa i dati di uno studente* era stato poi ridefinito come istruzione composta di passi più semplici, come in (1).

```
for studente := 1 to 50 do
begin
    leggi e stampa il nome;
    leggi e stampa i voti;                                    (1)
    allinea l'input e l'output per lo studente successivo
end.
```

Ciascuno dei tre sottoproblemi dell'istruzione composta (1) è stato poi ridefinito in termini di altri sottoproblemi o, dove possibile, direttamente in PASCAL. Questo processo di raffinamento incrementale ha portato, alla fine, ad una situazione in cui tutte le parti del progetto del programma risultano espresse completamente in PASCAL.

Quando progettiamo programmi più lunghi e complessi, questo metodo assume un'importanza ancora maggiore. Vero è però, che le azioni isolate ai primi stadi del processo di raffinamento possono passare per parecchi livelli ulteriori di raffinamento, prima di diventare lunghe e complicate sequenze in PASCAL, ciascuna delle quali comprenderà molte linee, o addirittura pagine, di programma. Quando si andrà a leg-

gere il programma, la notevole lunghezza del testo renderà difficile coglierne la struttura – usando i costrutti PASCAL descritti sinora, il solo modo di incorporare esplicitamente i livelli di raffinamento entro il programma finale è per mezzo dei commenti. Quello che serve è, invece, un mezzo per poter suddividere il testo del programma in unità significative corrispondenti ai sottoproblemi identificati durante la costruzione, per poi esprimere brevemente l'azione complessiva del programma in termini di tali unità. Ciò aiuterebbe il lettore a vedere più chiaramente la struttura del programma, rendendogli più semplice non solo la sua comprensione, ma anche le eventuali modifiche necessarie.

Il PASCAL, come molti altri linguaggi, fornisce la possibilità di descrivere i programmi in tal modo. Il programmatore può definire un'azione, o gruppo di azioni, sotto forma di *procedura,* alla quale dà un nome, noto come *identificatore-procedura.* Questa procedura può poi essere invocata in un altro punto del programma per mezzo di un'*istruzione-procedura,* che nomina l'*identificatore-procedura.*

La sintassi di un'*istruzione-procedura* è

istruzione-procedura = *identificatore-procedura* [*lista-parametri-attuali*].
identificatore-procedura = *identificatore.*

Per il momento ci occuperemo solo della forma più semplice di *istruzione-procedura,* quella in forma di identificatore. Si dice che un'*istruzione-procedura chiama* la procedura denotata dall'identificatore. Il suo effetto è l'esecuzione delle azioni definite da quella procedura (e poi di proseguire con l'esecuzione dell'istruzione immediatamente seguente l'*istruzione-procedura*).

Un'*istruzione-procedura* è una delle forme ammesse di *istruzione-semplice,* definite nel Cap. 4, e può dunque essere usata ovunque sia consentito usare un'istruzione. Possiamo così mantenere il raffinamento del Programma 2 come parte attiva del programma stesso scrivendolo come indicato in (2). In (2) *leggiestampanome, leggiestampavoti, allineainputedoutput* sono ora *istruzioni-procedura,* mentre le procedure che definiscono le azioni necessarie sono dichiarate altrove nel programma.

```
begin
    for studente := 1 to 50 do
    begin
        leggiestampanome;
        leggiestampavoti;                                    (2)
        allineainputedoutput
    end
end.
```

La dichiarazione di una *procedura* ha la forma indicata in (3). Una *dichiarazione-procedura* consiste, cioè, di una *intestazione-procedura* ed un *corpo-procedura,* separati da un punto e virgola. In sostanza, il ruolo della *intestazione-procedura* è di associare un nome o identificatore alla *procedura* da dichiarare. Lo scopo e l'uso della *lista-parametri-formali* sono discussi più avanti.

dichiarazione-procedura = *intestazione-procedura* ";" *corpo-procedura*.
intestazione-procedura = "**procedure**" *identificatore* [*lista-parametri-formali*]. (3)
corpo-procedura = *blocco.*

L'azione della *procedura* è definita dal *corpo-procedura,* che ha la forma di un *blocco.* Come abbiamo visto nel Cap. 2, un blocco può cominciare con una parte dichiarazioni non vuota, ma il suo componente essenziale è la parte istruzioni:

blocco = parte-dichiarazioni parte-istruzioni.

Come la *parte-istruzioni* del blocco del programma definisce l'azione del programma, la *parte-istruzioni* di un blocco di procedura definisce l'azione della procedura, cioè l'azione da eseguire ogniqualvolta è chiamata la procedura. Possiamo allora dichiarare la procedura *leggiestampanome* come in (4). Si noti che l'*istruzione-while,* che nel Programma 2 stampava gli spazi seguenti il nome, può ora essere rimpiazzata da un'*istruzione-if.* Le altre due procedure richieste dal Programma 2 si possono dichiarare in modo analogo.

```
procedure leggiestampanome;
var prossimocarattere: char;
    lunghezzanome : 0..30;
begin
    lunghezzanome := 0;                                        (4)
    repeat
        read (prossimocarattere);
        write (prossimocarattere);
        lunghezzanome := lunghezzanome + 1
    until prossimocarattere = '.';
    if lunghezzanome < 30 then write (' ' : 30-lunghezzanome)
end;
```

In PASCAL le dichiarazioni di procedura sono raggruppate in una *parte-dichiarazione-procedure-e-funzioni* che, come visto nel Cap. 2, è il componente finale della *parte-dichiarazioni.* La sua forma è definita in (5). La *parte-dichiarazione-procedure-e-funzioni* consiste di zero o più *dichiarazioni-procedura* o *funzione* seguite da punto e virgola (la dichiarazione e l'uso delle funzioni sono spiegati più oltre).

parte-dichiarazione-procedure-e-funzioni =
    { (*dichiarazione-procedura* | *dichiarazione-funzione*) ";" }. (5)

Così, nel Programma 2 riscritto usando le procedure, le tre procedure sono dichiarate tra la restante dichiarazione di variabile e la parte istruzioni rivista, come in (6). Il riposizionamento delle altre dichiarazioni di variabile è discusso nel prossimo paragrafo.

```
program voti (input, output);
var studente : 1..50;

    procedure leggiestampanome;
    var prossimocarattere: char;
        lunghezzanome : 0..30;
    begin
        lunghezzanome := 0;
```

```
        repeat
            read (prossimocarattere);
            write (prossimocarattere);
            lunghezzanome := lunghezzanome + 1;
        until prossimocarattere = '.';
        if lunghezzanome < 30
          then write (' ' : 30-lunghezzanome)
    end;

    procedure leggiestampavoti;
    var materia : 1..5;
        voto : 0..10;
        totale : 0..50;                                          (6)
        stella : 1..20;
    begin
        { leggi e calcola il totale dei voti }
        totale := 0;
        for materia := 1 to 5 do
        begin
            read (voto);
            totale := totale + voto
        end;
        { stampa voto totale }
        write (totale : 10)
        { stampa istogramma }
        write (' ');
        for stella := 1 to round (0.4 * totale) do write ('*')
    end;

    procedure allineainputedoutput;
    begin
        readln;
        writeln
    end;
begin
    for studente := 1 to 50 do
    begin
        leggiestampanome;
        leggiestampavoti;
        allineainputedoutput
    end
end.
```

Nella versione modificata del programma la *parte-istruzioni* mostra esplicitamente la struttura complessiva concepita per il programma – un ciclo il cui corpo comprende una serie di tre azioni sussidiarie – mentre le dichiarazioni di procedura spiegano in dettaglio come eseguire tali azioni. In questo esempio alcune azioni sono banali, e la

loro espressione in forma di procedura può sembrare una complicazione. L'esempio, tuttavia, mostra chiaramente come in un programma più lungo e complesso i livelli significativi del raffinamento delle azioni del programma si possono esprimere con brevi sequenze di istruzioni, comprendenti chiamate di procedura. I raffinamenti finali delle azioni, che tali istruzioni invocano, sono graficamente separati dalle istruzioni stesse, e anche tra di loro, essendo raccolti in un insieme appropriato di dichiarazioni di procedura.

La dichiarazione di una procedura non causa l'esecuzione del suo corpo, o parte istruzioni – questa è innescata soltanto dall'esecuzione di una corrispondente *istruzione-procedura,* o *chiamata.* Nell'esempio precedente, a ciascuna procedura corrispondeva esattamente una chiamata, ma non è necessariamente così – in un programma possono essere usate due o più istruzioni di chiamata della stessa procedura. Per vedere questo riconsideriamo il programma *buzzbuzz* illustrato nel Cap. 6. Correggiamo il programma, in modo che esso legga un intero n ($0 < n < 100$) e produca in output la successione di interi n, $n+1$, $n+2$, ..., inframmezzata dalle parole *buzz* e *buzz-buzz,* come prima. Al raggiungimento di 100 la successione dovrebbe ricominciare ciclicamente da 1, ed arrivata ad n-1 il programma dovrebbe fermarsi. Questo si può ottenere assai semplicemente con l'uso di una procedura, come in (7).

In questo caso l'uso di due cicli-**for** esprime, semplicemente e chiaramente, il controllo richiesto per generare le appropriate successioni di valori per *i.* Ogni corpo di ciclo applica la stessa azione al valore corrente di *i,* ed è perciò espresso come una chiamata alla stessa procedura *scrivibuzz.*

```
program buzzbuzz (input, output);
var i, n : 1..99;
    procedure scrivibuzz;
    begin
        if (i div 10 = 7) or (i mod 10 = 7)
          then writeln ('buzz ')
          else if i mod 7 = 0
                 then writeln ('buzz-buzz ')
                 else writeln (i)                    (7)
    end;
begin
  read (n);
  for i := n to 99 do scrivibuzz;
  for i := 1 to n-1 do scrivibuzz
end.
```

Usare la stessa struttura di controllo senza una procedura, in questo caso, significherebbe scrivere due copie del corpo della procedura – una per ciascun ciclo-**for**. L'uso di una procedura riduce la lunghezza totale del testo finale del programma. In generale, quando una procedura è chiamata da due o più punti distinti del programma, c'è una notevole riduzione della lunghezza del testo del programma, rispetto a quella dello stesso programma scritto senza procedure. Questa riduzione di lunghezza del testo si riflette, normalmente in una corrispondente riduzione delle dimensioni del pro-

gramma eseguibile contenuto nell'elaboratore. L'uso di una procedura che sia chiamata da due o più punti del programma presenta un triplice vantaggio economico, in quanto esso permette di ridurre

(a) la complessità del problema, misurata in termini di lunghezza del testo del programma,
(b) la probabilità di errori di programmazione,
(c) le dimensioni del programma eseguibile risultante.

Tali risparmi costituiscono la motivazione principale per l'inclusione delle procedure in molti linguaggi.

Al contrario, si potrebbe dire che l'uso di una procedura, che sia chiamata in un punto soltanto del programma, aumenta la lunghezza del testo del programma – a causa della presenza dei simboli in più, occorrenti alla dichiarazione di procedura ed alla sua chiamata. È anche vero che si introduce un appesantimento corrispondente nel programma eseguibile derivato – sia in lunghezza che in tempo di esecuzione. Tali appesantimenti sono controbilanciati dall'aumento di leggibilità e dalla conseguente diminuizione della probabilità di presenza di errori, che l'introduzione di una procedura comporta. In parecchie situazioni tali vantaggi superano di molto i relativi svantaggi. Nel seguito del libro le procedure sono usate liberamente, sia per sottolineare la struttura del programma, sia per riassumere, ove appropriato, un'azione comune a due o più punti del programma.

7.2. STRUTTURA A BLOCCHI E CAMPO D'AZIONE

Nel riscrivere il Programma 2 nella forma (6), le dichiarazioni delle variabili usate in ciascuna procedura sono state localizzate in quella procedura – in generale la definizione sintattica di un *corpo-procedura* come blocco permette la dichiarazione di dati all'interno di una procedura. In realtà, non solo possiamo dichiarare costanti, tipi e variabili, ma anche procedure.

Si dice che ogni identificatore introdotto entro la parte dichiarazioni di un blocco per rappresentare costanti, tipi, variabili e procedure è *locale* al blocco. Il blocco è la sola parte del programma in cui si può far riferimento a tali identificatori, ed è chiamato *campo d'azione (scope)* degli identificatori. Per esempio, supponiamo di dichiarare una variabile *a* in una procedura *P*, come in (8). Allora alla variabile *a* si può far riferimento solo nel corpo della procedura *P* – si dice che *a* è una variabile locale di *P*, e non è conosciuta all'esterno della procedura.

```
procedure P;
var a : integer;
begin                                    (8)
:
end
```

Tuttavia, gli identificatori dichiarati esternamente ad un blocco, cioè nella parte dichiarazioni del blocco racchiudente, sono noti, e ad essi si può fare riferimento all'interno del blocco racchiuso. Tali identificatori sono detti *non-locali.* o *globali,* rispetto

al blocco racchiuso. Per esempio, in (9) alla variabile *b* si può far riferimento all'interno della procedura *P,* ed è perciò detta variabile *non-locale* o *globale* della procedura *P.*

```
var b : integer;
procedure P;
begin                                                    (9)
  :
end
```

Si consideri nuovamente la versione modificata del Programma 2 del paragrafo precedente. Le variabili *caratteresuccessivo* e *lunghezzanome* sono usate solo entro la procedura *leggiestampanome,* e possono essere dichiarate locali a tale procedura. Analogamente, le variabili *totale, materia, voto* possono essere dichiarate locali alla procedura *leggiestampavoti.* La variabile *studente* è usata dalla parte istruzioni del programma stesso e deve perciò essere dichiarata nel blocco del programma. Questo significa che a *studente* si potrebbe far riferimento come variabile globale da qualsiasi procedura, sebbene in realtà nessuna lo faccia.

Tale uso delle dichiarazioni locali presenta tre vantaggi:

(a) rende esplicito che le variabili *prossimocarattere, lunghezzanome, totale, materia, voto* e *stella* sono significative solo entro le procedure in cui sono dichiarate, semplificazione notevole per il lettore del programma;
(b) assicura che certi errori, causati dall'uso inavvertito di queste variabili in altre parti del programma, siano immediatamente scoperti dal compilatore;
(c) come vedremo, aiuta l'implementazione a minimizzare la memoria usata per le variabili.

Negli esempi visti sinora abbiamo usato solo procedure dichiarate nella parte dichiarazioni del blocco del programma. In teoria, però, ogni corpo di procedura è a sua volta un blocco, la cui parte dichiarativa può contenere altre dichiarazioni di procedura, e dunque altri blocchi. Si dice che una procedura dichiarata in un'altra procedura è *annidata* in essa. Per esempio, nel raffinare la procedura *leggiestampavoti* nel Programma 2, avremmo potuto introdurre altre procedure come in (10).

```
procedure leggiestampavoti;
var totale : 0..50;
  procedure leggiesommavoti;
  var materia : 1..5;
      voto : 0..10;
  begin
      totale := 0;
      for materia := 1 to 5 do
      begin
          read (voto);
          totale := totale + voto                        (10)
      end
  end;
```

```
        procedure stampaistogramma;
        var stella : 1..20;
        begin
            write ('   ');
            for stella := 1 to round (0.4 * totale) do write ('*')
        end;
    begin
        leggiesommavoti;
        stampavototale;
        stampaistogramma
    end
```

La struttura complessiva di un programma PASCAL si presenta, così, come un insieme di blocchi, alcuni dei quali sono annidati entro altri, fino ad un livello arbitrario di annidamento. Ciascuno di questi blocchi può introdurre nuovi identificatori nella sua parte dichiarazioni e le *regole di campo d'azione* debbono determinare l'accessibilità di tutti questi identificatori in tutta la struttura.

Negli esempi usati finora gli identificatori erano unici in tutto il programma. Nel Cap. 3 abbiamo detto che l'associazione di ciascun identificatore deve essere unica in tutto il suo intervallo d'uso – tale intervallo, o campo d'azione, è stato ora definito come il blocco in cui è dichiarato l'identificatore. Ovviamente, un identificatore non deve essere dichiarato per due scopi distinti nello stesso blocco. D'altra parte, non c'è ragione per cui lo stesso identificatore non possa essere dichiarato per scopi differenti in blocchi differenti, poiché le regole di campo d'azione chiariscono in ogni punto del programma risultante quale dichiarazione, se esiste, sia valida.

Le regole del campo d'azione degli identificatori in PASCAL sono dunque definite formalmente come segue:

(i) il campo d'azione della dichiarazione di un identificatore è il blocco cui appartiene la dichiarazione, e tutti i blocchi racchiusi in quel blocco, a meno della regola (ii);

(ii) quando un identificatore dichiarato in un blocco *A* è ridichiarato in un blocco *B*, racchiuso da *A*, allora il blocco *B*, e tutti i blocchi da esso racchiusi, sono esclusi dal campo d'azione della dichiarazione dell'identificatore in *A*.

Gli identificatori standard del PASCAL si considerano dichiarati in un blocco immaginario racchiudente l'intero programma.

La regola generale introdotta nel Cap. 3, secondo cui un identificatore va dichiarato prima dell'uso, vale sempre. Va fatto notare, tuttavia, che la regola (ii) significa che se un identificatore *I* è ridichiarato in un blocco *B*, non può essere usato da qualche blocco esterno *A*, nella parte di *B* precedente la sua ridichiarazione, con qualche significato non-locale, perchè *tutto* il blocco *B* è escluso dal campo d'azione della dichiarazione di *I* in *A*.

Lo schema di programma (11) illustra l'applicazione delle regole di campo d'azione, che determinano a quali identificatori ci si riferisce nelle parti istruzioni delle varie procedure, e del programma principale. Il lettore dovrebbe assicurarsi, prima di conti-

nuare, di aver compreso come tali regole individuino gli identificatori accessibili elencati nei commenti inseriti in ogni parte istruzioni.

```
program P;
var i, j : integer;
    procedure Q;
    const i = 16;
    var k : char;
        procedure R;
        var j : real;
        begin
            { questa parte istruzioni può usare
              la variabile locale j : real;
              la costante non-locale i = 16;
              la variabile non-locale k : char;
              le procedure non-locali R, Q;
              tutti gli identificatori standard }
        end;
    begin
        { questa parte istruzioni può usare
          la costante locale i = 16;
          la variabile locale k : char;
          la procedura locale R;
          la variabile non-locale j : integer;
          la procedura non-locale Q;
          tutti gli identificatori standard }
    end;
begin
    { questa parte istruzioni può usare
      le variabili locali i, j : integer;
      la procedura locale Q;
      tutti gli identificatori standard }
end.
```

(11)

Si noti che gli identificatori di procedura sono soggetti alle stesse regole cui sono sottoposti gli altri identificatori, cosicché una procedura può essere usata solo entro il blocco in cui appare la sua dichiarazione (ed in ogni blocco racchiuso da tale blocco). La possibilità che una procedura si riferisca a, e dunque chiami, se stessa (nota come *ricorsione*), è discussa più avanti.

Si noti inoltre l'uso dell'indentazione per indicare l'annidamento dei blocchi di una procedura nel testo del programma. Questa è una buona regola di programmazione, in quanto permette al lettore di apprezzare la struttura procedurale del programma.

Le regole di campo d'azione determinano i limiti d'uso degli identificatori, entro il testo di un particolare programma. Le stesse regole si usano per determinare il ciclo di vita, o *esistenza,* delle variabili. Il PASCAL stabilisce che le variabili dichiarate locali ad una procedura esistono solo durante l'esecuzione della procedura, essendo create

all'ingresso nella procedura, e distrutte all'uscita. Ne consegue che

(a) ad una variabile dichiarata locale ad una procedura non si può far riferimento nella parte di programma che ha chiamato la procedura, poiché la variabile è creata solo all'entrata nella procedura, e cessa di esistere prima che il controllo torni alla parte chiamante;
(b) non c'è alcuna relazione tra i valori delle variabili create da esecuzioni successive della stessa procedura. Il valore di una variabile locale è sempre indefinito, dopo l'ingresso in una procedura (cosicché la prima azione di una procedura, in generale, sarà l'inizializzazione delle sue variabili), poiché ogni volta che è chiamata una procedura, viene creato un nuovo insieme di variabili con valori indefiniti.

Questo limite all'esistenza, o ciclo di vita, delle variabili permette alle implementazioni di minimizzare la memoria usata per rappresentarle nel calcolatore. La memoria usata per rappresentare le variabili locali di una procedura è acquisita solo all'ingresso nella procedura, e ridiventa disponibile per l'uso alla fine della sua esecuzione. Due procedure chiamate in sequenza possono così usare la stessa memoria per le loro variabili locali. Il programmatore aiuta l'implementazione a minimizzare la memoria usata per le variabili, se le rende locali ovunque possibile.

Inizialmente, le regole di campo d'azione e di esistenza degli identificatori possono sembrare complesse all'utente poco avvezzo ai linguaggi strutturati a blocchi. Egli, però, troverà presto in tali regole uno strumento notevole per la costruzione di programmi chiari, senza errori ed economici in termini di memoria. La giusta collocazione delle dichiarazioni per ottenere tali benefici si consegue, solitamente, in modo abbastanza naturale con un raffinamento incrementale, in cui le procedure siano usate liberamente per esprimere lo sviluppo della struttura del programma. A questo scopo occorre osservare i seguenti principi guida:

(a) Quando è possibile occorre dichiarare un identificatore nel blocco in cui è usato. Così si aumenta la chiarezza del programma, si minimizza la probabilità di errori derivanti dall'uso inavvertito degli identificatori in altre parti, e si minimizzano le necessità di memoria per ciò che riguarda le variabili.
(b) Quando un identificatore deve essere dichiarato in due o più blocchi per denotare lo stesso oggetto, va dichiarato nel blocco che racchiude immediatamente tutti gli usi dell'identificatore.
(c) Quando una variabile usata da una procedura deve mantenere il suo valore tra una chiamata della procedura e la successiva, va dichiarata nel primo blocco racchiudente che le assicuri un ciclo di vita tale da abbracciare tutte le chiamate di procedura dipendenti.

Rendere locali gli identificatori, ove possibile, aumenta pure la libertà del programmatore nella scelta degli identificatori da usare. Gli identificatori locali a due procedure, che siano completamente separate l'una dall'altra (cioè l'una non racchiudente l'altra), possono essere scelti del tutto indipendentemente. Così, (12) in PASCAL è una rappresentazione valida di due procedure appartenenti allo stesso programma. Gli oggetti denotati dagli identificatori usati in una procedura non

hanno assolutamente alcuna relazione con quelli con lo stesso nome usati nell'altra.

```
procedure P;
const c = '?';
var d, e : char;
begin
⋮
end;
procedure Q;
type e = 1..100;                                    (12)
var c : boolean;
    d : char;
begin
⋮
end;
```

7.3. PARAMETRI

Come abbiamo visto, una procedura può essere usata per definire un'azione, o una serie di azioni, da eseguire in due o più punti distinti di un programma. Per esempio, vogliamo ordinare i valori di due variabili globali intere x e y, in più punti di un programma, in modo che il valore di x non sia maggiore del valore di y, scambiando i loro valori se necessario. Invece di scrivere l'istruzione (13), in ogni punto del programma ove si richiede l'ordinamento, potremmo dichiarare una procedura (14), e chiamarla per mezzo dell'istruzione di procedura

ordina

nei punti del programma interessati.

```
if x > y then
        begin                                       (13)
            t:=x ; x:=y ; y:=t
        end
```

```
procedure ordina;
var t : integer;
begin
    if x > y then                                   (14)
            begin
                t:=x ; x:=y ; y:=t
            end
end
```

Questa procedura ordina i valori delle due variabili intere x e y. In pratica, spesso è più conveniente dichiarare una procedura che ordini i valori di due variabili intere qualsiasi, non solo di x e y. Allora, se in qualche punto del programma dovremo ordinare x e y, potremo scrivere

ordina (x, y)

mentre se, in un altro punto, vogliamo ordinare i valori di *y* e di un'altra variabile intera *z*, possiamo scrivere

ordina (y, z)

Questo, in PASCAL, è possibile, se si dichiara una procedura che comprenda una *lista-parametri-formali* nella sua intestazione, nel modo seguente:

procedure *identificatore lista-parametri-formali;*
. . .
. . .

La *lista-parametri-formali* definisce un insieme di oggetti fittizi, chiamati *parametri-formali,* in termini dei quali è definito il corpo della procedura. Quando la procedura viene chiamata, questi *parametri-formali* sono rimpiazzati dai corrispondenti *parametri-attuali,* così specificati nell'*istruzione-procedura*:

identificatore-procedura lista-parametri-attuali

Per la procedura *ordina* una dichiarazione appropriata sarebbe la (15), in cui *a* e *b* sono i *parametri-formali,* in termini dei quali è definita la procedura *ordina.* Per ordinare i valori delle variabili *x* e *y* l'istruzione di procedura necessaria è

ordina (x, y)

dove *x* e *y* sono i parametri *attuali* che rimpiazzano *a* e *b,* in questa particolare esecuzione della procedura.

```
procedure ordina (var a, b : integer);
var t : integer;
begin
    if a > b then
            begin                                          (15)
                t:=a ; a:=b ; b:=t
            end
end
```

Più in dettaglio, una *lista-parametri-formali* consiste di una o più sezioni, separate da punti e virgola, e racchiusa da parentesi tonde:

lista-parametri-formali = "(" sezione-parametri-formali
{ ";" sezione-parametri-formali } ")".

Ogni *sezione* introduce uno o più parametri formali della stessa *classe.* Ci sono quattro *classi* di parametri formali in PASCAL, cioè

(a) parametri *per valore*;
(b) parametri *per variabile*;
(c) parametri *procedura*;
(d) parametri *funzione.*

La classe di un parametro formale è determinata dalla forma della *sezione-parametri-formali* cui appartiene, come indicato in (16).

```
sezione-parametri-formali = sezione-parametri-per-valore|
                            sezione-parametri-per-variabile|
                            sezione-parametri-procedura|
                            sezione-parametri-funzione.                    (16)
sezione-parametri-per-valore = lista-identificatori " : " tipo-parametro.
sezione-parametri-per-variabile = "var" lista-identificatori " : " tipo-parametro.
sezione-parametri-procedura = intestazione-procedura.
sezione-parametri-funzione = intestazione-funzione.
tipo-parametro = identificatore-tipo| schema-array-conforme.
```

In ciascuna sezione sono dichiarati uno o più identificatori, che denotano i parametri formali richiesti; inoltre, eccetto che per la classe delle procedure, è specificato il tipo corrispondente. Notare che un *tipo-parametro* (che non sia uno *schema-array-conforme)* può essere specificato soltanto da un identificatore di tipo già dichiarato in un blocco che lo racchiude.

La *lista-parametri-attuali* usata in un'istruzione di procedura prende una forma corrispondente alla *lista-parametri-formali* della procedura. Essa consiste di un numero appropriato di *parametri-attuali,* separati da virgole:

```
lista-parametri-attuali = "(" parametro-attuale
                          { "," parametro-attuale } ")".
```

La forma di un *parametro-attuale* è determinata dalla classe del corrispondente *parametro-formale.* Esistono quattro possibili forme, descritte in (17):

```
parametro-attuale = valore-attuale | variabile-attuale|
                    procedura-attuale | funzione attuale.
valore-attuale = espressione.                                              (17)
variabile-attuale = variabile.
procedura-attuale = identificatore-procedura.
funzione-attuale = identificatore-funzione.
```

La corrispondenza tra le liste di parametri attuali e formali di un'istruzione di procedura e della dichiarazione della procedura deve seguire le seguenti regole:

(i) il numero dei parametri nelle due liste deve essere lo stesso;

(ii) ogni parametro attuale corrisponde al parametro formale che occupa la stessa posizione nella lista-parametri-formali;

(iii) i parametri attuali e formali corrispondenti debbono essere consistenti come descritto, per ciascuna classe, nei paragrafi seguenti. Le classi di parametri più usate sono quelle per variabile e per valore il cui *tipo-parametro* è denotato da un identificatore di tipo. Ora ci occupiamo di queste, mentre le classi di parametri procedura e funzione sono discusse più avanti. I parametri array conformi sono studiati nel Cap. 9.

7.3.1. Parametri per variabile

Un parametro formale per variabile è usato per denotare un parametro attuale il cui valore può essere alterato dall'esecuzione della procedura. Il parametro attuale corrispondente deve perciò essere una variabile dello stesso tipo. L'esecuzione di una particolare istruzione procedura determina il parametro attuale coinvolto e, per tutta la seguente esecuzione del corpo della procedura, ogni operazione riguardante il parametro formale è applicata direttamente alla variabile attuale.

In alcuni casi la procedura può usare il valore esistente di un parametro attuale per variabile, prima di cambiarlo. Per esempio, la procedura *ordina,* che definiamo in (18), esamina e scambia i valori dei suoi due parametri, se necessario.

```
procedure ordina (var a, b : integer);
var t : integer;
begin
    if a > b then                                          (18)
            begin
                t:=a ; a:=b ; b:=t
            end
end
```

I parametri formali *a* e *b* debbono essere dichiarati come parametri per variabile – di qui la presenza del prefisso **var** nella loro dichiarazione. In un'istruzione procedura corrispondente

```
ordina (x, y)
```

i parametri attuali *x* e *y* debbono essere variabili intere. Per di più debbono sempre avere assegnati dei valori, poiché la procedura per prima cosa esamina tali valori, per determinare la sua azione successiva.

In altri casi, i valori esistenti dei parametri per variabile possono essere irrilevanti. Per esempio, la procedura (19) legge un numero reale, che rappresenta una distanza in metri, e lo converte in piedi e pollici (arrotondando al pollice). In una corrispondente istruzione di chiamata, ad esempio

```
leggiinmetri (piedi, pollici)
```

i parametri attuali *piedi* e *pollici* debbono essere variabili intere, ma i loro valori sono irrilevanti, poiché la procedura non fa alcun tentativo di usarli prima di assegnarne loro due nuovi.

```
procedure leggiinmetri (var piedi, pollici: integer);
var metri: real; lunghezzainpollici : integer;
begin
    read (metri);                                          (19)
    lunghezzainpollici := round (metri * 39.39);
    pollici := lunghezzainpollici mod 12;
    piedi := lunghezzainpollici div 12
end
```

I parametri, come quelli di *leggiinmetri,* che sono usati per portare fuori della procedura dei valori sono talvolta chiamati parametri *d'uscita.*

I parametri, come quelli di *ordina,* che sono usati per portar valori dentro e fuori la procedura sono noti come parametri di *ingresso-uscita.* In ogni caso debbono essere dichiarati in PASCAL come parametri per variabile.

7.3.2. Parametri per valore

Si usa un parametro per valore quando occorre soltanto introdurre in una procedura un valore – un cosiddetto parametro *d'ingresso.* Il parametro formale denota semplicemente un valore, fornito dall'istruzione di chiamata, e perciò il corrispondente parametro attuale può essere qualsiasi espressione che produca un valore compatibile per assegnazione col tipo del parametro formale.

```
procedure scriviinmetri (piedi, pollici: integer);
var metri: real; lunghezzainpollici : integer;
begin
    lunghezzainpollici := 12 * piedi + pollici;                    (20)
    metri := lunghezzainpollici / 39.39;
    write (metri : 6:2)
end
```

Per esempio, la procedura (20) prende due interi, rappresentanti una distanza in piedi e pollici, e la riscrive in metri. In questo caso *piedi* e *pollici* sono parametri per valore, il che è indicato dall'assenza del prefisso **var** nella loro dichiarazione. I corrispondenti parametri attuali, nelle istruzioni chiamanti la procedura, possono essere qualsiasi espressione che produca un valore intero. Per esempio,

```
scriviinmetri (6,4)
scriviinmetri (pd, pl)
scriviinmetri (pd + 1, 0)
```

sono tutte istruzioni di chiamata permesse per questa procedura, purché *pd* e *pl* siano variabili intere.

In PASCAL, l'espressione che costituisce un parametro attuale è valutata al momento dell'esecuzione dell'istruzione di chiamata ed il suo valore è assegnato al parametro formale corrispondente, come se quest'ultimo fosse una variabile locale della procedura. Durante la sua esecuzione, la procedura può esaminare questo parametro formale e assegnargli altri valori, sempre come se fosse una variabile locale. Tali assegnazioni non hanno però effetto sul parametro attuale dell'istruzione procedura.

```
procedure scriviinmetri (piedi, pollici : integer);
var metri : real;
begin
    pollici := 12 * piedi + pollici;                               (21)
    metri := pollici / 39.39;
    write (metri : 6:2)
end
```

Per esempio, avremmo potuto scrivere la procedura *scriviinmetri* senza la variabile locale *lunghezzainpollici,* come in (21). L'effetto complessivo di questa versione è esattamente lo stesso di prima. In particolare, una chiamata della forma

scriviinmetri (pd, pl)

non produrrà effetti sulla variabile *pl,* data come valore di ingresso di *pollici.* { Naturalmente, in pratica, il corpo della procedura potrebbe essere scritto senza alcuna variabile locale, semplicemente come istruzione singola

*write ((12 * piedi + pollici) / 39.39 : 6:2) }.*

La possibilità di riutilizzare i parametri per valore come variabili locali è utile in certi casi. D'altra parte, proprio per questo, occorre prestare molta attenzione a garantire che tutti i parametri per variabile occorrenti siano dichiarati tali. L'omissione del simbolo **var** nella lista dei parametri formali significa semplicemente che i parametri formali sono presi per valore. Eventuali assegnazioni ad essi entro il corpo della procedura rimangono valide, ma non producono effetto sulle corrispondenti variabili attuali fornite dalle istruzioni di chiamata. Il risultato è un programma apparentemente corretto, ma che non produce l'effetto desiderato. Il vero errore – l'omissione di un simbolo **var** – è spesso difficile da scoprire.

Molte procedure prevedono sia parametri per variabile che per valore. Le loro dichiarazioni richiedono liste di parametri formali divise in due o più sezioni, per distinguere le classi di parametri formali coinvolte. Per esempio, potremmo definire delle procedure di conversione interna tra metri e piedi e pollici come in (22).

```
procedure convertiinmetri (piedi, pollici : integer; var metri : real);
begin
    metri := (12 * piedi + pollici) / 39.39
end                                                                    (22)

procedure convertidametri (metri : real; var piedi, pollici : integer);
var lunghezzainpollici := integer;
begin
    lunghezzainpollici := round (metri * 39.39);
    piedi := lunghezzainpollici div 12;
    pollici := lunghezzainpollici mod 12
end
```

La procedura (23) riceve i coefficienti di un'equazione quadratica

$$ax^2 + bx + c = 0$$

e calcola le sue radici reali, se esistono, usando una variabile booleana per segnalare questa condizione.

```
procedure risolvi (a, b, c : real;
                   var radice1, radice2 : real;
                   var esistonoradici : boolean);
var d : real;
```

```
begin
    d := b * b - 4 * a - c;
    if d < 0
        then esistonoradici := false                                    (23)
        else begin
                esistonoradici := true;
                d := sqrt (d);
                radice1 := -(b-d) / (2 * a);
                radice2 := -(b+d) / (2 * a)
             end
end
```

Notare che

(a) sebbene i primi cinque parametri siano tutti di tipo *real,* occorrono due sezioni parametri formali per distinguere i parametri per variabile da quelli per valore;
(b) il simbolo **var** deve essere ripetuto per ciascuna sezione di parametri per variabile di tipo distinto.

PROGRAMMA 5 (Output alfabetico di una somma di denaro)

Un programma deve leggere una serie di somme di denaro in dollari, minori di $1000, rappresentate con due interi separati da uno spazio, e deve scrivere in output, per ogni somma, sia la forma numerica, sia il suo equivalente alfabetico in inglese. Per esempio, per l'input

```
542 99
```

dovrebbe produrre l'output

```
542 99 FIVE HUNDRED AND FORTY TWO DOLLARS AND NINETY NINE
CENTS
```

La sequenza di input termina con un numero negativo. Una possibile applicazione di questo programma è la stampa di assegni guidata dal calcolatore. La struttura complessiva del programma potrebbe essere, dopo uno o più passi di raffinamento, quella in (24).

```
begin
    read (dollari);
    while dollari > = 0 do
    begin
        read (centesimi);
        write (dollari, centesimi);
        if sia dollari che centesimi sono zero
          then write ('nil')
          else begin                                                    (24)
                  if dollari > 0
```

```
                then begin
                        converti dollari in parole;
                        write ('dollars');
                        if centesimi > 0
                          then write ('and')
                     end;
              if centesimi > 0
                then begin
                        converti centesimi in parole;
                        write ('cents')
                     end
          end;
      writeln;
      read (dollari)
  end
end
```

I passi *converti dollari in parole* e *converti centesimi in parole* comprenderanno azioni simili, e possono essere riscritti come chiamate ad una stessa procedura *convertiinparole* che prende parametri adeguati; per esempio

```
convertiinparole (dollari)
convertiinparole (centesimi)
```

La procedura *convertiinparole* ha l'intestazione

```
procedure convertiinparole (x : intervallo);
```

dove il tipo *intervallo* è definito nel programma principale come il sottointervallo 0..999. L'effetto della procedura è la scrittura della forma alfabetica del suo parametro di ingresso *x*. Lo schema di questa procedura è in (25). Si osservi che, nel trattamento delle ultime due cifre di *x*, i valori tra 11 e 19 hanno forma alfabetica diversa dagli altri, e debbono quindi essere considerati separatamente.

```
begin
  separa le centinaia, le decine e le unità di x;
  if la cifra delle centinaia > 0
    then begin
           converti la cifra delle centinaia;
           write ('hundred');                                    (25)
           if la somma di decine e unità > 0
             then write ('and')
         end;
  if la cifra delle decine = 1
    then converti un numero tra 10 e 19
    else begin
```

converti la cifra delle decine;
converti la cifra delle unità
end
end

Le azioni *converti la cifra delle centinaia* e *converti la cifra delle unità* si possono esprimere come chiamate della procedura

procedure *unita (i : cifra)*

dove il tipo *cifra* del parametro per valore *i* è dichiarato non localmente come sottointervallo 0..9. Questa sarà una procedura locale di *convertiinparole,* e dunque il tipo *cifra* sarà dichiarato localmente a *convertiinparole.*

Scritta la procedura *unita* e raffinate le azioni astratte in *convertiinparole* e nel programma principale, otteniamo il programma mostrato nel Listato 5 insieme ad un output campione.

LISTATO 5

```
PROGRAM NUMERIINPAROLE(INPUT,OUTPUT);

(* QUESTO PROGRAMMA ACCETTA COME INPUT UNA SERIE DI NUMERI
   CHE RAPPRENTANO SOMME DI DENARO MINORI DI $1000 E STAMPA
   IL LORO EQUIVALENTE ALFABETICO. LA SERIE
   DI NUMERI IN INPUT E` CONCLUSA DA UN NUMERO NEGATIVO *)

TYPE INTERVALLO = 0..999;

VAR  DOLLARI   : INTEGER;
     CENTESIMI : 0..99;

     PROCEDURE CONVERTIINPAROLE(X : INTERVALLO);
     TYPE CIFRA = 0..9;
     VAR  H,T,U : CIFRA;

          PROCEDURE UNITA(I : CIFRA);
          BEGIN
             CASE I OF
                0 : ;
                1 : WRITE(' ONE');
                2 : WRITE(' TWO');
                3 : WRITE(' THREE');
                4 : WRITE(' FOUR');
                5 : WRITE(' FIVE');
                6 : WRITE(' SIX');
                7 : WRITE(' SEVEN');
                8 : WRITE(' EIGHT');
                9 : WRITE(' NINE');
             END
          END; (* UNITA *)
```

```
BEGIN
   H:= X DIV 100; T:= X MOD 100 DIV 10; U:= X MOD 10;
   IF H > 0
      THEN BEGIN
              UNITA(H);
              WRITE(' HUNDRED');
              IF T + U > 0 THEN WRITE(' AND')
           END;
   IF T = 1
      THEN CASE U OF
              0 : WRITE(' TEN');
              1 : WRITE(' ELEVEN');
              2 : WRITE(' TWELVE');
              3 : WRITE(' THIRTEEN');
              4 : WRITE(' FOURTEEN');
              5 : WRITE(' FIFTEEN');
              6 : WRITE(' SIXTEEN');
              7 : WRITE(' SEVENTEEN');
              8 : WRITE(' EIGHTEEN');
              9 : WRITE(' NINETEEN');
           END

           ELSE BEGIN
                   CASE T OF
                      0 : ;
                      2 : WRITE(' TWENTY');
                      3 : WRITE(' THIRTY');
                      4 : WRITE(' FORTY');
                      5 : WRITE(' FIFTY');
                      6 : WRITE(' SIXTY');
                      7 : WRITE(' SEVENTY');
                      8 : WRITE(' EIGHTY');
                      9 : WRITE(' NINETY');
                   END;
                   UNITA(U)
                END
     END; (* CONVERTIINPAROLE *)

BEGIN
   READ(DOLLARI);
   WHILE DOLLARI >= 0 DO
   BEGIN
      READ(CENTESIMI);
      WRITE(DOLLARI:4, CENTESIMI:3, ' ':5);
      IF (DOLLARI = 0) AND (CENTESIMI = 0)
         THEN WRITE (' NIL')
         ELSE BEGIN
                 IF DOLLARI > 0
                    THEN BEGIN
                            CONVERTIINPAROLE(DOLLARI);
                 WRITE(' DOLLARS');
                 IF CENTESIMI > 0 THEN WRITE(' AND')
              END;
```

```
      IF CENTESIMI > 0
         THEN BEGIN
                 CONVERTIINPAROLE(CENTESIMI);
                 WRITE(' CENTS')
              END
   END;
   WRITELN;
   READ(DOLLARI);
   END
END.
```

```
653 37 SIX HUNDRED AND FIFTY THREE DOLLARS AND THIRTY SEVEN CENTS
  0  0 NIL
250  0 TWO HUNDRED AND FIFTY DOLLARS
116 27 ONE HUNDRED AND SIXTEEN DOLLARS AND TWENTY SEVEN CENTS
  0 83 EIGHTY THREE CENTS
100 16 ONE HUNDRED DOLLARS AND SIXTEEN CENTS
 17 90 SEVENTEEN DOLLARS AND NINETY CENTS
```

7.4. FUNZIONI

Nei capitoli precedenti abbiamo descritto le funzioni standard del PASCAL (ad esempio *sqrt, succ, trunc, eof*), e discusso il loro uso nella costruzione di espressioni. Per esempio, nell'espressione

y / sqrt (x) + 3

sqrt (x) è una *chiamata di funzione* che ha l'effetto di iniziare una computazione che produce un risultato il cui valore è la radice quadrata di *x,* dove *x* è il parametro attuale della chiamata di funzione. Il valore prodotto è sostituito nell'espressione di cui sopra e questa viene poi valutata.

Il PASCAL fornisce non soltanto queste funzioni standard (che non richiedono dichiarazione nel programma), ma anche un mezzo per permettere al programmatore di dichiarare e far valutare, come componenti di espressioni, le proprie *funzioni* con parametri opportuni. Una funzione è una particolare forma di procedura, che descrive una computazione che produce in output un singolo valore. Però, mentre una procedura è attivata da un'istruzione di procedura, una funzione è attivata all'interno di un'espressione, al cui valore contribuisce il risultato della funzione.

La dichiarazione di una funzione è simile a quella di una procedura, e prende la seguente forma:

dichiarazione-funzione = intestazione-funzione ";" corpo-funzione.
intestazione-funzione = **"function"** *identificatore*
[lista-parametri-formali] ":" tipo-risultato.
tipo-risultato = identificatore-tipo.
corpo-funzione = blocco.

Una *dichiarazione-funzione* compare, insieme con le altre dichiarazioni di procedura o funzione, nella *parte-dichiarazione-procedure-e-funzioni* di un blocco, cioè una *dichiarazione-funzione* può apparire ovunque lo possa una *dichiarazione-procedura.*

Una *intestazione-funzione* è simile ad un'*intestazione-procedura,* tranne per il fatto che comincia col simbolo **function,** e contiene un'indicazione del tipo del risultato calcolato dalla funzione – *il tipo-risultato.* Questo tipo deve essere un tipo non strutturato, oppure puntatore (vedi Cap. 13), dichiarato precedentemente.

L'*identificatore* e *la lista-parametri-formali,* se c'è, giocano nell'*intestazione-funzione* un ruolo analogo a quello che hanno nell'*intestazione-procedura* – determinano il modo in cui si può chiamare la funzione. *L'identificatore-funzione,* però, svolge un ruolo ulteriore entro il *corpo-funzione,* perché denota il risultato che deve produrre l'esecuzione della procedura.

La categoria sintattica

identificatore-funzione = identificatore.

è usata d'ora innanzi per denotare la classe degli identificatore introdotti da una *dichiarazione-funzione.*

Il *corpo-funzione* è un blocco, e può perciò includere dichiarazioni locali di altre costanti, tipi, variabili, procedure e funzioni. Ai blocchi delle dichiarazioni di funzione si applicano le stesse regole di campo d'azione degli identificatori viste per le dichiarazioni di procedura. Entro un *corpo-funzione,* però, deve comparire almeno un'istruzione di assegnazione che attribuisca un valore di *tipo-risultato* all'*identificatore-funzione.* In ogni esecuzione della funzione, l'ultimo valore così assegnato determina il risultato della valutazione della funzione. Se nell'esecuzione non ha luogo alcuna assegnazione all'*identificatore-funzione,* il risultato di tale valutazione è un errore.

Quello che segue è un esempio di *dichiarazione-funzione,* che definisce una funzione *max* che restituisce il maggiore tra i due valori dei suoi due parametri formali *x* e *y.*

```
function max (x, y : real) : real;
begin
    if x > y
      then max := x
      else max := y
end
```

L'intestazione della funzione specifica l'identificatore *max* mediante il quale è nota la funzione, i suoi parametri formali, ed il tipo *real* per il risultato della computazione della funzione. Il corpo-funzione definisce tale computazione in modo che il risultato sia il maggiore tra i due parametri per valore.

Una funzione è chiamata (invocata, o *designata*) semplicemente mediante l'occorrenza entro un'espressione del suo identificatore e dell'eventuale *lista-parametri-attuali,* come abbiamo già visto nel caso di chiamate di funzioni standard. Richiamiamo la definizione sintattica di espressione data nel Cap. 4:

*fattore = variabile | costante-senza-segno | designatore-funzione | limite | insieme | "(" espressione ")" | "***not***" fattore.*

e definiamo il *designatore-funzione*

designatore-funzione = identificatore-funzione [*lista-parametri-attuali*].

La lista-parametri-attuali dell'invocazione di una funzione obbedisce alle stesse re-

gole di corrispondenza con la *lista-parametri-formali* della dichiarazione-funzione viste per le chiamate di procedura.

L'effetto di una chiamata di funzione consiste nell'esecuzione della parte-istruzioni della dichiarazione-funzione corrispondente, dopo la sostituzione (secondo la classe di appartenenza) dei parametri attuali ai formali. Purché durante la sua esecuzione sia stato assegnato un valore all'identificatore-funzione, la funzione ritorna un valore singolo di tipo-risultato; tale valore è poi usato nella valutazione dell'espressione contenente la chiamata di funzione. Se *x, y* e *z* sono variabili di tipo *real,* allora

```
x := max (y, z)
```

assegnerà ad *x* il maggiore dei valori delle variabili *y* e *z*;

```
x := 2 * max (y, 1.5)
```

assegnerà ad *x* il doppio del maggiore tra il valore di *y* ed 1.5;

```
x := max (x, max (y, z))
```

assegnerà ad *x* il maggiore dei valori di *x, y* e *z*. Poiché le espressioni contenute nella lista parametri attuali della chiamata debbono essere valutate prima che sia valutata l'espressione stessa, quest'ultimo esempio è equivalente a

```
t := max (y, z);
x := max (x, t)
```

dove t è una variabile ausiliaria, non usata per altri scopi.

(26)-(28) sono ulteriori esempi di dichiarazione-funzione in cui si assume l'esistenza delle seguenti definizioni di tipo

```
interononnegativo = 0..maxint;
interopositivo = 1..maxint;
```

```
function potenza (x : real; n : interononnegativo) : real;
{ questa funzione calcola x elevato n, con n > = 0 }
var i : interononnegativo; risposta : real;
begin                                                    (26)
    risposta := 1.0;
    for i := 1 to n do
        risposta := risposta * x;
    potenza := risposta
end
```

```
function sommadiquadrati (n : interononnegativo) : interononnegativo;
{ questa funzione calcola la somma dei quadrati
  dei primi n numeri interi }
var i, somma : interononnegativo;
begin                                                    (27)
    somma := 0;
    for i := 1 to n do
      somma := somma + sqr (i);
    sommadiquadrati := somma
end
```

```
function multiplo (i, j : interopositivo) : boolean;
{ questa funzione determina se i e j
  sono multipli l'uno dell'altro }
begin                                                    (28)
    multiplo := (i mod j = 0) or (j mod i = 0)
end
```

Notare che, sebbene all'*identificatore-funzione* si possano fare più assegnazioni durante l'esecuzione del *corpo-funzione,* ad un valore precedentemente assegnato non si può avere accesso per mezzo dell'*identificatore-funzione,* il quale quindi non si può riutilizzare come se fosse una variabile locale. Quando il risultato della funzione è ottenuto per accumulazione, va usata una variabile ausiliaria locale, come *risposta* nella funzione *potenza,* o *somma* in *sommadiquadrati.* Come vedremo, ogni occorrenza di un identificatore di funzione in una espressione entro il corpo della funzione stessa viene considerata un'ulteriore chiamata (ricorsiva) della funzione.

I seguenti sono esempi di istruzioni che invocano le funzioni di cui sopra:

```
y := potenza (6.3,5)
x := sommadiquadrati (3) + sommadiquadrati (n)
while not multiplo (y, x) do
begin
    y := y + 1;
    x := x + 1;
end
```

Il programma (29) legge due interi *n* e *m (n* è per ipotesi minore di *m)* e produce una lista di tutti i numeri perfetti compresi tra *n* e*m.* Un numero perfetto è tale che la somma dei suoi fattori è pari al numero stesso: ad esempio 6 = 1+2+3. La funzione *perfetto* determina se il suo parametro intero per valore è un numero perfetto.

```
program numeriperfetti (input, output);
type interononnegativo = 0..maxint;
var i, n, m : interononnegativo;
    function perfetto (j : interononnegativo) : boolean;
    var i, somma : interononnegativo;
    begin
        somma := 1;                                      (29)
        for i := 2 to j div 2 do
          if j mod i = 0
            then somma := somma + 1;
        perfetto := (somma = j)
    end;
begin
    read (n, m);
    writeln ('lista di tutti i numeri perfetti tra', n, ' e', m);
    for i := n to m do
      if perfetto (i)
        then writeln (i)
end.
```

7.4.1. Effetti collaterali delle funzioni

Quando un'istruzione, eseguita nel corpo di una funzione, altera il valore di una variabile nota all'esterno della funzione, si dice che la funzione ha un *effetto collaterale.* Gli effetti collaterali rappresentano una interazione potenziale tra i dati della funzione e quelli del programma che la invoca che è sempre pericolosa, perché rende il programma difficile da leggere (il valore di una variabile può cambiare durante la valutazione di un'espressione contenente una chiamata di funzione, e può rendere ambiguo il significato del programma).

```
function f (x : integer) : real;
begin
    v := v * x;                                    (30)
    f := sqrt (x) + 1
end
```

Supponiamo di avere una funzione *f* che, durante la sua esecuzione, altera il valore di una variabile non-locale *v,* come per esempio in (30). Consideriamo ora la valutazione dell'espressione

$$f(x) + v$$

Il valore di questa espressione dipende dall'ordine di valutazione dei suoi operandi, cioè dal fatto che il valore di *v* sia ricavato prima o dopo la determinazione del valore di *f (x).* Se si valuta prima *f (x),* allora il valore di *v* prodotto dall'assegnazione in *f* è sommato al risultato di *f (x),* altrimenti il valore originale di *v* è sommato al risultato. In PASCAL, l'ordine di valutazione degli operandi di un operatore binario, come +, risulta dipendente dall'implementazione, cosicché pure il valore dell'espressione è dipendente dall'implementazione. Persino quando sia noto l'ordine di valutazione, l'espressione $f(x) + v$ può produrre un risultato diverso da $v + f(x)$!

Volendo evitare tali situazioni ambigue, occorre fare a meno, quando possibile, di funzioni con effetti collaterali. Ne segue che le liste di parametri formali delle funzioni non dovrebbero contenere parametri per variabile, e che nelle funzioni si dovrebbe evitare di fare assegnazioni a variabili non-locali, nonché chiamate a procedure che effettuino assegnazioni di questo genere.

La valutazione di una funzione può richiedere l'esecuzione di un grosso pezzo di programma, e quindi occorre evitare chiamate inutili. Per esempio, se *f* è una funzione con parametro reale per valore, l'istruzione

```
x := f (y / 2) + f (y / 2) * sqrt (f (y / 2))
```

invoca tre volte con lo stesso parametro la funzione *f,* la quale (nell'ipotesi che *f* non produca effetti collaterali) produrrà tre volte lo stesso risultato. È dunque più efficiente valutare la funzione una volta sola, e memorizzare il suo risultato in una variabile;

```
t := f (y / 2);
x := t + t * sqrt (t)
```

Se però *f* ha effetti collaterali, allora queste due sequenze di istruzioni produrranno

effetti diversi, poiché *f* è chiamata tre volte nella prima, ma solo una nella seconda.

PROGRAMMA 6 (Ricerca di numeri primi)

Vogliamo scrivere un programma che legga un intero positivo *n,* e poi altri *n* interi, maggiori di 1, scrivendo in output il numero primo più vicino a ciascuno di essi. Nel caso di due primi equidistanti da un intero in input, verranno scritti entrambi.

Il programma richiesto svolge le azioni indicate in (31). L'azione *stampa il numero primo più vicino a j* può essere ridefinita come in (32). Il passo *localizza e stampa il numero primo più vicino* ispeziona i numeri dispari, a cominciare da quello più vicino al valore d'ingresso, allontanandosi via via da tale valore, fino a trovare un numero primo. Se l'input è un numero pari, allora la sua distanza dai due dispari più vicini è 1, se è dispari, tale distanza è 2.

```
begin
    read (n)
    for i := 1 to n do
    begin
        read (j);                                                    (31)
        write ('il numero primo piu" vicino a ', j, ' e " ' );
        stampa il numero primo più vicino a j
    end
end
```

```
if j è un numero primo
  then stampa j                                                      (32)
  else localizza e stampa il numero primo più vicino
```

Si può dunque scrivere un ciclo che testa i dispari, via via più lontani dall'input, e termina quando trova un numero primo. Ricordando che se si trovano due primi equidistanti dal valore d'ingresso vanno stampati entrambi, si può scrivere la (33).

```
if odd (j)
  then k := 2
  else k := 1;
repeat
    if j + k è primo
      then stampa j + k;                                             (33)
    if j − k è primo
      then stampa j − k;
    k := k + 2
until trovato primo
```

Poiché il ciclo deve terminare quando viene eseguita un'azione di stampa, la condizione di terminazione è ben descritta da una variabile booleana *trovato,* inizialmente falsa, che diventa vera se è avvenuta la stampa di un risultato.

Abbiamo così tre punti in cui controllare se un particolare valore è un numero primo, e cioè *j, j+k e j−k* rispettivamente. Una scelta adeguata consiste nell'introdurre una funzione primo, che prende come parametro un intero positivo, e ritorna un risul-

tato booleano vero o falso, secondo che l'intero sia primo o no. Così

```
function primo (p : interopositivo) : boolean;
. . .
```

dove il tipo *interopositivo* è definito globalmente come

```
interopositivo = 1..maxint
```

Usando questa funzione, l'azione *localizza e stampa il numero primo più vicino* si può esprimere come in (34).

```
trovato := false;
if odd (j)
  then k := 2
  else k := 1;
repeat
  if primo (j + k)
    then begin
           trovato := true;
           write (j + k)
         end ;
  if primo (j - k)
    then begin                                          (34)
           trovato := true;
           write (j - k)
         end;
  k := k + 2
until trovato;
writeln
```

Dobbiamo ora scrivere la funzione *primo*. Dato che 2 e 3 sono numeri primi, e valendoci del fatto che un numero pari maggiore di 2 non è mai primo, poiché è sempre divisibile per 2, controlliamo se sono primi solo i dispari maggiori di 3. Così lo schema della funzione sarà

```
if p < 4
  then primo := p > 1
  else if not odd (p)
         then primo := false
         else controlla se p è primo
```

Poiché i fattori di un numero si possono raggruppare in coppie tali che un fattore è maggiore o uguale della radice quadrata del numero stesso, mentre l'altro è minore o uguale di tale radice, ne consegue che nessun numero ha fattori maggiori della sua radice quadrata, se non ne ha di minori. Durante il passo *controlla se p è primo* siamo certi che *p* è dispari, e quindi tale proprietà può essere verificata semplicemente controllando se esistono numeri dispari, fattori di *p*, compresi tra 3 e la radice quadrata di *p*. Dunque *p* è numero primo se e solo se tale processo non riesce a trovare divisori di

p. Scriviamo allora *controlla se p è primo* come ciclo che divide *p* per i dispari a cominciare da 3, finché o si trova un divisore di *p*, oppure sono stati verificati tutti i dispari fino alla radice quadrata di *p* inclusa. Se tale radice non è un intero, basterà arrivare fino al dispari più vicino alla sua parte intera. Per calcolare tale limite potremmo, per esempio, scrivere

trunc (sqrt (p))

Però se *p* è un quadrato della forma $(q+1)^2$, allora approssimazioni dovute alla funzione *sqrt* potrebbero dare una radice della forma *q*.999..., ed il troncamento darebbe allora un valore limite *q*, invece che *q*+1. Per premunirsi contro tali approssimazioni scriviamo invece

round (sqrt (p))

sebbene questo talvolta dia un limite maggiore di uno del valore strettamente occorrente.

L'azione *controlla se p è primo* è descritta in (35). Il programma completo ed un output campione sono mostrati nel Listato 6.

```
begin
    radice := round (sqrt (p));
    divisore := 3;
    while (divisore < = radice) and (p mod divisore < > 0) do
        divisore := divisore + 2;                                    (35)
    primo := divisore > radice
end
```

LISTATO 6

```
PROGRAM TROVAPRIMI (INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE UN INTERO POSITIVO N SEGUITO
   DA ALTRI N INTERI POSITIVI (TUTTI > 1) E STAMPA IL
   NUMERO PRIMO PIU' VICINO A CIASCUNO.
   SE IL NUMERO E' EQUIDISTANTE DA DUE NUMERI PRIMI
   SONO STAMPATI ENTRAMBI                                        *)

TYPE INTEROPOSITIVO = 1..MAXINT;

VAR  N, I, K : INTEROPOSITIVO;
     J       : 2..MAXINT;
     TROVATO : BOOLEAN;

     FUNCTION PRIMO (P : INTEROPOSITIVO) : BOOLEAN;
     (* DETERMINA SE P E' UN NUMERO PRIMO *)
     VAR  RADICE, DIVISORE : INTEROPOSITIVO;
     BEGIN
        IF P < 4
           THEN PRIMO:= P > 1
           ELSE IF NOT ODD(P)
                   THEN PRIMO:= FALSE
```

```
                    ELSE BEGIN
                            RADICE:= ROUND(SQRT(P));
                            DIVISORE:= 3;
                            WHILE (DIVISORE <= RADICE) AND
                                  (P MOD DIVISORE <> 0) DO
                               DIVISORE:= DIVISORE + 2;
                            PRIMO:= DIVISORE > RADICE
                         END
     END; (* PRIMO *)

BEGIN
   READ(N);
   FOR I:= 1 TO N DO
   BEGIN
      READ(J);
      WRITE('IL NUMERO PRIMO PIU'' VICINO A',J,' E''');
      IF PRIMO(J)
         THEN WRITELN(J)
         ELSE BEGIN
                 TROVATO:= FALSE;
                 IF ODD(J)
                    THEN K:= 2
                    ELSE K:= 1;
                 REPEAT
                    IF PRIMO (J + K)
                       THEN BEGIN
                               TROVATO:= TRUE;
                               WRITE(J + K)
                            END;
                    IF PRIMO(J - K)
                       THEN BEGIN
                               TROVATO:= TRUE;
                               WRITE(J - K)
                            END;
                    K:= K + 2
                 UNTIL TROVATO;
                 WRITELN
              END
   END
END.
```

```
IL NUMERO PRIMO PIU' VICINO A       100 E'       101
IL NUMERO PRIMO PIU' VICINO A       246 E'       251       241
IL NUMERO PRIMO PIU' VICINO A         2 E'         2
IL NUMERO PRIMO PIU' VICINO A       333 E'       331
IL NUMERO PRIMO PIU' VICINO A      2007 E'      2011      2003
IL NUMERO PRIMO PIU' VICINO A       761 E'       761
```

7.5. PARAMETRI PROCEDURA E PARAMETRI FUNZIONE

All'inizio di questo capitolo abbiamo introdotto nei dettagli due classi di parametri, quelli per variabile e quelli per valore. Il PASCAL permette la specifica di altre classi di parametri, come i parametri procedura e funzione, che studiamo adesso.

Un parametro formale procedura è specificato da una sezione parametri formali che ha essa stessa la forma di un'intestazione di procedura. Tale sezione parametri formali specifica così il nome del parametro procedura e, se esistono, i suoi parametri. Per esempio, l'intestazione

procedure *P* (**procedure** *A*)

introduce una procedura *P* con un parametro formale procedura *A*, che non ha parametri. Entro il corpo di *P* l'identificatore *A* è usato come ogni altro identificatore di procedura senza parametri, ad esempio in un'istruzione procedura.

Un parametro formale funzione è definito, analogamente, da una sezione parametri formali della forma di una intestazione di funzione. Per esempio, l'intestazione

procedure *Q* (**function** *F* (*x* : *integer*) : *integer*)

introduce una procedura *Q* con un parametro formale *F* che prende un parametro intero per valore, e produce un risultato intero. Entro il corpo di *Q* l'identificatore *F* è usato come ogni altro identificatore di funzione simile, ad esempio in una designazione di funzione.

Quando si chiama una procedura (o funzione) che ha un parametro di classe procedura o funzione, il corrispondente parametro attuale deve essere l'identificatore di una procedura o funzione con stessa descrizione dei parametri e stesso tipo risultato. Quando viene eseguito il corpo della procedura (o funzione) chiamata, ogni occorrenza del parametro formale implica un uso corrispondente della procedura o funzione attuale fornita come parametro.

```
procedure tabulazione (function f (x : real): real; inferiore, superiore, passo: real);
var x : real;
    j : integer;
begin
    x := inferiore;                                                    (36)
    for j := 0 to trunc ((superiore−inferiore) / passo) do
    begin
        writeln (x : 13, f (x) : 20);
        x :=x + passo
    end
end
```

Si consideri la procedura (36). *tabulazione* prende un parametro funzione *f*, che restituisce un reale, e tre parametri per valore, di tipo *real*. Il suo effetto è di generare una successione di valori di *x*, determinata dai parametri *inferiore, superiore e passo*, e per ciascun *x* di valutare e stampare *f (x)*. Così un'istruzione di procedura

*tabulazione (f*1, 0.0, 1.0, 0.01*)*

causerrebbe la tabulazione di x e di $f1\ (x)$, per x che assume valori nell'intervallo da 0 a 1 in passi da 0.01, mentre l'istruzione di procedura

*tabulazione (f*2, 0.0, 1.0, 0.01*)*

produce un tabulato analogo per la funzione $f2$, dove $f1$ e $f2$ sono funzioni che trasformano reali in reali, e che saranno dichiarate altrove nel programma.

Come esempio di funzione che prende per parametro una funzione, dichiariamo una funzione *sigma* (37) che calcola la sommatoria

$$\sum_{i=\text{inferiore}}^{\text{superiore}} f(i)$$

dove f è una funzione che trasforma interi in interi.

```
function sigma (function f (i : integer) : integer;
                inferiore, superiore : integer) : integer;
begin                                                        (37)
    somma := 0;
    for i: inferiore to superiore do somma := somma + f (i);
    sigma := somma
end
```

Se il programma contiene una funzione come la (38), allora l'invocazione di funzione

s := sigma (quartapotenza, 1,10)

calcolerebbe la sommatoria

$$\sum_{i=1}^{10} i^4$$

e assegnerebbe il risultato a *s*.

```
function quartapotenza (x : integer) : integer;
begin                                                        (38)
    quartapotenza := sqr (sqr (x))
end
```

La funzione formale f in (37) e la funzione attuale *quartapotenza* sono *congrue,* in quanto prendono lo stesso numero e tipo di parametri, e producono risultati dello stesso tipo. Si noti però che per denotare i rispettivi parametri formali, sono usati identificatori diversi: *i* in un caso, *x* nell'altro. In realtà, l'identificatore *i* nell'intestazione della funzione che definisce f non serve a niente, poiché non esiste alcun corrispondente corpo della funzione che usi *i*. Il PASCAL mantiene gli identificatori

nell'intestazione di procedure e funzioni formali solo per evitare notazioni differenti tra specifiche formali ed attuali. In generale, le liste-parametri-formali di procedure e funzioni attuali e formali corrispondenti possono differire solo per gli identificatori che denotano i loro parametri formali, o per gli identificatori di limite, quando ci sono parametri array conformi. Altrimenti, le liste dei parametri debbono essere identiche nel numero delle sezioni parametri formali che contengono, nel numero e classe dei parametri definiti da sezioni corrispondenti, e nei tipi usati nella loro definizione. Se le liste di parametri di procedure o funzioni attuali e formali corrispondenti contengono altre procedure o funzioni formali, queste pure debbono essere congrue allo stesso modo.

La possibilità di passare procedure e funzioni ad altre procedure e funzioni è assai utile in alcune situazioni, e può essere usata per costruire procedure o funzioni multiuso, come *tabulazione* o *sigma* viste sopra. Tale possibilità va però usata con cautela, poiché l'effetto dell'esecuzione di una procedura che riceve un'altra procedura come parametro non è sempre ovvio. L'interazione attraverso i dati non-locali che ciascuna procedura manipola può produrre effetti complessi, e talora inaspettati. I programmi che usano procedure come parametri sono spesso difficili sia da capire che da correggere, proprio per questa ragione.

Un altro problema è costituito del fatto che, in PASCAL, non è permesso il passaggio di procedure o funzioni standard. Così chiamate come le seguenti

tabulazione (sin, 0.0, 1.0, 0.01*)*

e

s := *sigma (sqr,* 1, 100*)*

non sono valide. Tale restrizione si può facilmente superare, dichiarando una funzione equivalente che ha semplicemente il ruolo di invocare la funzione standard, e di chiamare poi la procedura (o funzione) con questa funzione equivalente come parametro attuale. Ciò è mostrato in (39).

```
function seno (x : real) : real;
begin                                                    (39)
    seno := sin (x)
end
```

Con tale dichiarazione è consentito chiamare *tabulazione* come segue:

tabulazione (seno, 0.0, 1.0, 0.01*)*

7.6. RICORSIONE

Nella definizione della struttura delle istruzioni in PASCAL, abbiamo visto vari esempi di descrizione *ricorsiva,* cioè definita in termini della struttura da descrivere. Per esempio, la definizione sintattica di un'*istruzione* PASCAL può essere espressa come

istruzione = **"if"** *espressione* **"then"** *istruzione*| ...

dove l'entità sintattica *istruzione* è definita in termini di se stessa. Di tali definizioni ri-

corsive è piena tutta la sintassi del PASCAL, perché permettono la costruzione di istruzioni, espressioni, procedure, ecc., annidate l'una nell'altra.

Questa stessa tecnica si può usare per descrivere un processo. Si consideri, per esempio, il processo che prende un valore intero non-negativo, e stampa in ordine *inverso* la successione di cifre decimali che rappresentano il numero. Tale processo si può descrivere, in astratto, come in (40).

```
begin
    stampa l'ultima cifra di N;
    if rimangono cifre                                    (40)
      then inverti le cifre rimanenti
end
```

In PASCAL, una procedura o funzione può chiamare non solo un'altra procedura o funzione, ma anche se stessa. Tale chiamata è detta *ricorsiva.* Così il processo di cui sopra si può scrivere in PASCAL in forma di procedura ricorsiva, come in (41).

```
procedure inverti (N : interononnegativo);
begin
    write (N mod 10);
    if N div 10 <> 0                                      (41)
      then inverti (N div 10)
end
```

dove il numero da invertire ad ogni passo è passato come parametro per valore in un'invocazione alla procedura ricorsiva *inverti.* L'istruzione di procedura

```
inverti (i)
```

stamperà l'intero *i* con cifre invertite. Si consideri ciò che accade quando *inverti* è chiamata con un valore particolare, per esempio 327. L'effetto della chiamata *inverti* (327) è

```
begin
    write (7);
    if true then inverti (32)
end
```

poiché 327 **mod** 10 = 7 e 327 **div** 10 = 32. L'effetto della nuova chiamata ricorsiva *inverti* (32) è

```
begin
    write (2);
    if true then inverti (3)
end
```

L'effetto della nuova chiamata ricorsiva *inverti* (3) è

```
begin
    write (3);
    if false then inverti (0)
end
```

e poiché 3 **div** 10 = 0, non scattano altre chiamate ricorsive. La chiamata *inverti* (3) è completata, quindi la chiamata *inverti* (32) continua. Anche questa viene completata, così continua la chiamata *inverti* (327), ed a sua volta anche questa termina. L'effetto complessivo della chiamata originale è dunque di stampare le cifre 7,2,3 in quest'ordine, con una chiamata per ogni cifra stampata. La successione di chiamate ricorsive termina a causa dell'istruzione condizionale che controlla la chiamata ricorsiva entro il corpo di *inverti*; un'istruzione del genere è necessaria in tutte le procedure ricorsive ben costruite.

La durata, o *tempo di esistenza,* di ciascuna chiamata ricorsiva della procedura *inverti* è strettamente compresa nella durata della sua chiamata progenitrice. Tale annidamento presenta vantaggi ulteriori quando i dati sono dichiarati locali alla procedura ricorsiva. Come abbiamo visto, la chiamata di una procedura crea un insieme di variabili locali che non ha nessuna relazione con l'insieme usato nella precedente esecuzione della stessa procedura. Ciò è vero anche quando una procedura chiama se stessa, ma poiché l'esecuzione precedente della procedura non è completa quando essa viene di nuovo chiamata ricorsivamente, le variabili locali dell'esecuzione precedente esistono ancora. (Non si può però accedere ad esse finché non è stata completata la chiamata ricorsiva). Così, per ogni variabile *v* dichiarata locale in una procedura ricorsiva *R,* una chiamata che generi *k* chiamate ricorsive di *R* produce *k*+1 istanze distinte della variabile *v,* il tempo di esistenza di ciascuna delle quali è strettamente compreso in quello della precedente.

Questa proprietà delle variabili locali di una procedura ricorsiva può essere usata vantaggiosamente in molte applicazioni della ricorsione. Si consideri una variante del nostro primo processo ricorsivo – quello che legge una sequenza di caratteri di lunghezza arbitraria conclusa da un carattere speciale come per esempio '.', e stampa la sequenza invertita. Questo processo può essere programmato ricorsivamente come indicato in (42).

```
procedure invertiinput;
var c : char;
begin
    read (c)
    if c <> '.'                                          (42)
      then invertiinput;
    write (c)
end
```

Quando è chiamata ad invertire una sequenza di input di lunghezza *n, invertiinput* genera in totale *n* chiamate ricorsive di procedura con *n* istanze della variabile *c.* Al livello *i*-esimo di ricorsione, l'istanza della variabile *c* contiene l'*i*-esimo carattere di input, e quando si disinnesca la ricorsione (cioè le chiamate ricorsive sono completate) tali istanze di *c* hanno prodotto i loro valori nell'ordine c_n, c_{n-1} ,..., c_1. *Invertiinput* dunque genera automaticamente il numero esatto di istanze di variabile richieste per invertire quella particolare sequenza di input, e ciascuna istanza esiste solo il tempo strettamente necessario.

Molti semplici processi si possono esprimere ricorsivamente, ma d'altra parte possono essere descritti ugualmente bene da semplici processi non ricorsivi che sfruttino

l'iterazione. Per esempio, la procedura *inverti* può essere scritta non-ricorsivamente come riportato in (43).

```
procedure inverti (N : interononnegativo);
begin
    repeat
        write (N mod 10);
        N := N div 10
    until N = 0
end
```
(43)

Molte funzioni matematiche sono definite ricorsivamente, e dunque sembra naturale calcolarne i valori scrivendo funzioni ricorsive. Un esempio familiare è il fattoriale, che può essere definito

$$fatt\ (0) = 1$$
$$fatt\ (n) = n * fatt\ (n-1), \qquad \text{per } n > 0$$

e nello stesso modo si può dichiarare una funzione PASCAL per valutare il fattoriale di un valore dato *n*, come in (44).

```
function fatt (N : interononnegativo) : interopositivo;
begin
    if n = 0
      then fatt := 1
      else fatt := n * fatt (n-1)
end
```
(44)

Anche la forma non ricorsiva di *fatt* è semplice

```
function fatt (n : interononnegativo) : interopositivo;
var x, i : integer;
begin
    x := 1;
    for i := 1 to n do
      x := x * i;
    fatt := x
end
```
(45)

In molte implementazioni queste forme non ricorsive di *inverti* e *fatt* sono assai più efficienti in termini di memoria e di tempo di esecuzione delle loro equivalenti ricorsive. Il rimpiazzamento di formulazioni ricorsive con forme iterative equivalenti può perciò rendersi opportuno per ragioni di semplicità ed efficienza.

Se, in generale, è sempre possibile riesprimere una procedura ricorsiva in forma non-ricorsiva, usando un'equivalente forma di iterazione, tale riespressione non è però affatto banale. Ci sono, perciò, in parecchie situazioni vantaggi significativi nell'uso delle procedure ricorsive:

(a) In molti casi esse sono il modo più naturale e immediato di descrivere un processo, com'è illustrato in alcuni dei programmi presentati più avanti nel libro. In particolare, sono uno strumento fondamentale nella descrizione dell'elaborazione di dati inerentemente ricorsivi (vedi Cap. 13).

(b) Esse permettono la generazione di certi processi iterativi, e dei dati necessari a controllarli, senza dover introdurre una "struttura dati" esplicita atta a contenere i dati stessi. Diversamente dalla procedura *inverti,* l'effetto della procedura ricorsiva *invertiinput* non può esser generato non-ricorsivamente senza l'introduzione di qualche strumento per conservare simultaneamente l'intera sequenza di caratteri. La procedura *invertiinput* invece genera esattamente il numero richiesto di variabili di tipo *char* per contenere i caratteri di input, mediante la sua variabile locale *c.*

PROGRAMMA 7 (Le torri di Hanoi)

Questo è un problema leggendario (ora sul mercato come gioco per bambini) che, sebbene appaia a prima vista difficile da risolvere, trova una semplice soluzione mediante l'uso di una procedura ricorsiva.

Ci sono tre pali di legno (detti sinistro, centrale e destro). Sul palo sinistro è infilata una pila di n dischi di misura decrescente bucati al centro (in modo che il disco di diametro maggiore sia sul fondo). La disposizione iniziale dei dischi è illustrata in Fig. 7.1, per il caso n=5.

Il problema è muovere tutti i dischi dal palo sinistro al destro, rispettando queste regole:

(a) si può muovere solo un disco per volta;
(b) non si può mai mettere un disco più grande su uno più piccolo;
(c) in ogni momento ogni disco deve essere su uno dei tre pali.

Noi vogliamo costruire un programma che legge un intero n e stampa la successione di mosse necessarie per risolvere il problema nel caso di n dischi.

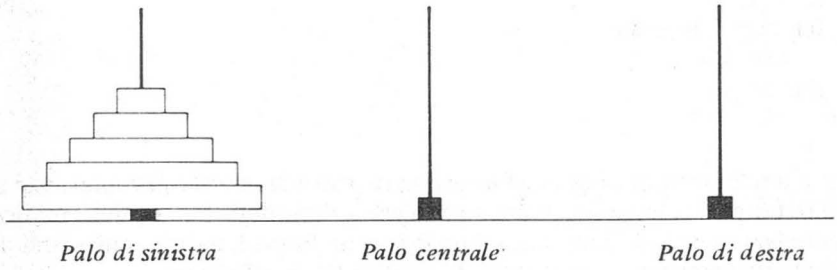


Fig. 7.1 Le torri di Hanoi

Il problema di muovere n (> 1) dischi può essere ridotto al problema di muovere $n-1$ dischi considerando la seguente soluzione in tre passi:

(a) muovi gli $n-1$ dischi più in alto dal palo sinistro al centrale, usando il palo destro come palo "ausiliario";

(b) muovi il disco rimanente dal palo sinistro al destro;

(c) muovi gli $n-1$ dischi dal palo centrale al palo destro, usando il sinistro come palo "ausiliario".

Tale metodo si uniforma alle regole del gioco.

In tal modo il problema di spostare n dischi è stato ridotto a quello di spostare $n-1$ dischi. In altre parole, il problema di muovere n dischi è stato descritto ricorsivamente in termini dello stesso problema con un disco di meno.

Quando $n=1$ il problema è risolto per semplice spostamento del disco dal palo sinistro al palo destro.

Tentiamo così una formulazione ricorsiva del problema di muovere n dischi da un palo all'altro. Questa procedura richiederà quattro parametri – il numero di dischi da muovere, il palo "sorgente" da cui partono i dischi, il palo "destinazione" che dovrà contenerli alla fine, il palo "ausiliario" che va usato per le mosse intermedie.

Per analogia con la descrizione ricorsiva precedente scriviamo la procedura *muovi* riportata in (47).

```
procedure muovi (n : interopositivo; sorgente, ausiliario, destinazione : palo);
begin
    if n = 1
      then muovi un disco da sorgente a destinazione                    (47)
      else begin
                muovi (n−1, sorgente, destinazione, ausiliario);
                muovi un disco da sorgente a destinazione;
                muovi (n−1, ausiliario, sorgente, destinazione)
           end
end
```

Il tipo *palo* è definito nel programma principale come tipo enumerato

palo = (sinistro, centrale, destro)

e la parte istruzioni del programma principale deve solo leggere il numero di dischi *numerodidischi* sul canale di input, e poi chiamare

muovi (numerodidischi, sinistro, centrale, destro)

Poiché l'output richiesto è la sequenza di mosse che risolve il problema, l'azione *muovi un disco da sorgente a destinazione* è programmata come procedura, ogni chiamata della quale stampa la mossa corrispondente. Essa usa una procedura locale per stampare le identità dei pali coinvolti in ciascuna mossa.

Il programma completo, con l'output che produce per $n=4$, è mostrato nel Listato 7.

LISTATO 7

```
PROGRAM TORRIDIHANOI (INPUT,OUTPUT);

TYPE PALO = (SINISTRO, CENTRALE, DESTRO);
     INTEROPOSITIVO = 1..MAXINT;

VAR  NUMERODIDISCHI : INTEROPOSITIVO;

     PROCEDURE MUOVI (N: INTEROPOSITIVO;
                      SORGENTE, AUSILIARIO, DESTINAZIONE: PALO);

        PROCEDURE MUOVIUNDISCODASORGENTEADESTINAZIONE;
           PROCEDURE STAMPAPALO(P:PALO);
           BEGIN
              CASE P OF
                 SINISTRO  : WRITE('SINISTRO');
                 CENTRALE  : WRITE('CENTRALE');
                 DESTRO    : WRITE('DESTRO');
              END
           END; (* STAMPAPALO *)
        BEGIN
           WRITE('MUOVI UN DISCO DAL '); STAMPAPALO(SORGENTE);
           WRITE(' AL '); STAMPAPALO(DESTINAZIONE);
           WRITELN
        END; (* MUOVIUNDISCODASORGENTEADESTINAZIONE *)

     BEGIN
        IF N = 1
           THEN MUOVIUNDISCODASORGENTEADESTINAZIONE
           ELSE BEGIN
                   MUOVI(N-1, SORGENTE, DESTINAZIONE, AUSILIARIO);
                   MUOVIUNDISCODASORGENTEADESTINAZIONE;
                   MUOVI(N-1, AUSILIARIO, SORGENTE, DESTINAZIONE)
                END
     END; (* MUOVI *)

BEGIN
   READ(NUMERODIDISCHI);
   WRITELN('PER',NUMERODIDISCHI:3);
   WRITE(' DISCHI LE MOSSE NECESSARIE SONO:');
   WRITELN;
   MUOVI(NUMERODIDISCHI,SINISTRO,CENTRALE,DESTRO)
END.
```

```
PER  4 DISCHI LE MOSSE NECESSARIE SONO:

MUOVI UN DISCO DAL SINISTRO AL CENTRALE
MUOVI UN DISCO DAL SINISTRO AL DESTRO
MUOVI UN DISCO DAL CENTRALE AL DESTRO
MUOVI UN DISCO DAL SINISTRO AL CENTRALE
MUOVI UN DISCO DAL DESTRO AL SINISTRO
MUOVI UN DISCO DAL DESTRO AL CENTRALE
```

```
MUOVI UN DISCO DAL SINISTRO AL CENTRALE
MUOVI UN DISCO DAL SINISTRO AL DESTRO
MUOVI UN DISCO DAL CENTRALE AL DESTRO
MUOVI UN DISCO DAL CENTRALE AL SINISTRO
MUOVI UN DISCO DAL DESTRO AL SINISTRO
MUOVI UN DISCO DAL CENTRALE AL DESTRO
MUOVI UN DISCO DAL SINISTRO AL CENTRALE
MUOVI UN DISCO DAL SINISTRO AL DESTRO
MUOVI UN DISCO DAL CENTRALE AL DESTRO
    :
```

7.6.1 Mutua ricorsione

Le procedure e le funzioni ricorsive assumono, di solito, una delle due seguenti forme:

(a) *auto-ricorsione,* quando la procedura o funzione ricorsiva chiama esplicitamente se stessa, come negli esempi visti;
(b) *mutua ricorsione,* quando una procedura o funzione *A* chiama qualche altra procedura o funzione *B,* che a sua volta (direttamente o indirettamente) chiama *A.*

Nel caso di ricorsione mutua sorge un problema, dal momento che le regole di campo d'azione del PASCAL richiedono che una procedura o funzione debba essere dichiarata nel testo del programma prima di qualsiasi sua chiamata. Ovviamente, se due procedure *A* e *B* dichiarate nello stesso blocco si chiamano l'un l'altra, è impossibile che ciascuna procedura sia dichiarata prima della sua chiamata. La definizione del PASCAL supera questo problema permettendo di dichiarare l'uso di una procedura o funzione prima della sua dichiarazione vera. Tale dichiarazione "fittizia" annuncia l'esistenza della procedura, cosicché possano comparire chiamate della procedura prima della sua dichiarazione. La dichiarazione "fittizia" ha la forma

intestazione-procedura ; *"forward"*;

oppure

intestazione-funzione ; *"forward"*;

e appare entro la stessa *parte-dichiarazione-procedure-e-funzioni* della dichiarazione vera. La lista di parametri formali (se esiste), ed il tipo risultato (nel caso di funzione), sono inclusi solo nella dichiarazione "fittizia", ed omessi in quella completa.

Per esempio, le funzioni *A* e *B* in (46) possono essere mutuamente ricorsive, e *B* può essere chiamata da dentro *A,* anche se la sua vera dichiarazione segue quella di *A.*

```
function B (x : integer) : char; forward;
function A (y : integer) : char;
begin
   . . . B (i) . . .
end;                                              (46)
function B ; { parametri e tipo risultato omessi }
begin
   . . . A (j) . . .
end;
```

Per permettere la ricorsione mutua col meccanismo *forward,* in effetti si divide l'intestazione della procedura, che ne definisce il nome (insieme ai parametri di chiamata), dal corpo, che ne definisce le azioni. Ciò permette il riconoscimento, e la verifica di correttezza, delle chiamate alla procedura, prima che ne sia stato esaminato il corpo. Ci sono parecchi contesti in cui tale separazione dell'intestazione dal corpo della procedura può tornare utile. Per esempio, per mantenere il testo della procedura separato dal programma che la chiama, o per compilare la procedura separatamente dal programma chiamante. In effetti, può rendersi necessario scrivere la procedura addirittura in un diverso linguaggio di programmazione.

In pratica, la definizione del PASCAL fornisce un contesto generale entro il quale le implementazioni possono fornire le adeguate funzionalità, permettendo un insieme di cosiddette *direttive,* delle quali *forward* è un esempio.

La sintassi completa di una dichiarazione di procedura è la seguente:

dichiarazione-procedura = *intestazione-procedura* "**;**" *corpo-procedura* |
intestazione-procedura "**;**" *direttiva* |
identificazione-procedura "**;**" *corpo-procedura.*
intestazione-procedura = "**procedure**" *identificatore* [*lista-parametri-formali*].
identificazione-procedura = "**procedure**" *identificatore-procedura.*
corpo-procedura = *blocco.*

Nel caso normale un'intestazione di procedura ed il corrispondente blocco sono giustapposti direttamente come dichiarazione di procedura. Qualsiasi dichiarazione del genere può, però, prendere la forma di un'intestazione di procedura e di una *direttiva,* scelta nell'insieme fornito dall'implementazione. L'effetto di tale dichiarazione dipende dalla direttiva scelta. Nel caso della direttiva *forward,* questo effetto consiste nel rimandare la definizione del corpo della procedura ad un altro punto del programma. In tale punto verrà usata una corrispondente identificazione di procedura, ad indicare che il successivo corpo è quello della procedura identificata la cui intestazione è già stata data.

Una sintassi analoga permette l'uso delle direttive nelle dichiarazioni di funzione.

dichiarazione-funzione = *intestazione-funzione* "**;**" *corpo-funzione* |
intestazione-funzione "**;**" *direttiva* |
identificazione-funzione "**;**" *corpo-funzione.*
intestazione-funzione = "**function**" *identificatore*
[*lista-parametri-formali*] "**:**" *tipo-risultato.*
identificazione-funzione = "**function**" *identificatore-funzione.*
corpo-funzione = *blocco.*
tipo-risultato = *identificatore.*

La direttiva *forward* è la sola prevista dalla definizione del PASCAL, ma le implementazioni possono fornire altre direttive per altri scopi. I dettagli delle direttive disponibili dovrebbero trovarsi nella documentazione di ciascuna implementazione. Il loro uso non sarà ulteriormente preso in considerazione in questo libro.

ESERCIZI

7.1 Scrivere una procedura *ordina3* che scambia i valori dei suoi tre parametri interi *a, b* e *c,* se occorre, in modo tale che $a \leq b \leq c$. (usare la procedura *ordina* del par. 7.3.1).
Usare questa procedura per riscrivere il Programma 3, in modo che le lunghezze dei lati del triangolo in analisi non debbano essere date in input in ordine crescente.

7.2 Estendere la procedura *convertiinparole* del Programma 5, per poter convertire qualsiasi numero da 1 a 999999. Modificare il programma perché accetti somme di denaro minori di un milione di dollari, e riverificarlo con dati opportuni.

7.3 Scrivere
(a) una procedura *leggiottale* che legge una sequenza di cifre ottali e assegna ad un parametro l'intero positivo equivalente;
(b) una procedura *scriviottale* che stampa la sequenza di cifre ottali che denota il valore del suo parametro intero positivo.

Usare queste procedure in un programma che legge e stampa una sequenza di numeri ottali, uno per linea, stampando infine la loro somma in ottale.

7.4 L'esponenziale e^x di un numero x può essere definito come

$$\exp(x) = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad \left(= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \right).$$

Scrivere una funzione con intestazione

function *esponenziale (x : real) : real*

per calcolare questa formula con una precisione di quattro cifre significative.
Suggerimento: si smette di addizionare nuovi termini quando l'ultimo è minore di un decimillesimo della somma calcolata fino a quel punto.
Includere tale funzione in un programma che tabula i valori di e^x per $x = 0.0, 0.1, 0.2, \ldots, 1.0$. Paragonare tali valori con quelli prodotti dalla funzione standard *exp*.

7.5 Scrivere una funzione che accetta come parametri un intero positivo ed una cifra, e determina se la rappresentazione decimale dell'intero contiene quella cifra.
Scrivere un programma che legge una cifra *d* e tabula tutti gli interi da 1 a 100 tali che la rappresentazione decimale del numero, il suo quadrato ed il suo cubo contengono tutti *d*. Per esempio, se $d=1$, 13 è un numero di questa specie, perché 13, 169 e 2197 contengono tutti la cifra 1.

7.6 Riscrivere il programma che stampa una piramide di cifre (Eser. 6.3), usando un ciclo ed una procedura ricorsiva.

8.

L'istruzione-goto

Tutti i programmi che abbiamo presentato nei capitoli precedenti sono stati costruiti usando le forme semplici di istruzioni PASCAL (assegnazione, input, output), combinate in strutture più potenti per mezzo delle istruzioni strutturate, introdotte nel Cap. 6, e delle dichiarazioni di procedure e funzioni, descritte nel Cap. 7. Le istruzioni dei programmi PASCAL risultanti vengono eseguite nella sequenza in cui compaiono nel testo del programma, secondo gli schemi di controllo imposti dalle varie istruzioni strutturate. Questi schemi di controllo sono quasi sempre sufficienti per esprimere le azioni richieste in un programma.

Tuttavia, durante il progetto di un programma, capitano talvolta situazioni in cui descrivere le azioni di parti del programma in termini delle istruzioni strutturate PASCAL risulta artificioso o inefficiente. In tali circostanze potrebbe essere utile, per aumentare l'efficienza e la chiarezza del programma, deviare dall'ordine implicito di esecuzione associato alle istruzioni sin qui considerate. Questo è reso possibile dall'uso di *etichette* e di *istruzioni-goto.*

Nel Cap. 4 è stata presentata la sintassi di un'*istruzione* nelle seguente forma

istruzione = [*etichetta* ":"] *(istruzione-semplice* | *istruzione-strutturata).*
istruzione-semplice = [*(istruzione-assegnazione* | *istruzione-procedura* | *istruzione-goto)*].

Un'istruzione può essere *etichettata* mettendo davanti ad essa un'*etichetta,* cioè un intero nell'intervallo da 0 a 9999, seguito dal simbolo due punti.

etichetta = *intero-senza-segno.*

I seguenti sono esempi di istruzioni etichettate:

```
 99  :  writeln ('trovato un errore')
777  :  a := b
  1  :  repeat P (i) ; i := i+1 until i = j
```

Etichettando un'istruzione si ha la possibilità di fare riferimento ad essa da altre parti del programma, per mezzo di un'*istruzione-goto,* che ha la forma

istruzione-goto = **"goto"** *etichetta.*

L'effetto di questa istruzione, al momento dell'esecuzione, è di far diventare l'istruzione preceduta dall'*etichetta* specificata la successiva istruzione da eseguire, al posto dell'istruzione immediatamente seguente l'*istruzione-goto*. Esempi di istruzione-goto:

```
goto 777
if x > massimovalore then goto 99
if i < j then goto 1
```

Tutte le etichette devono essere dichiarate nella *parte-dichiarazione-etichette* del blocco in cui vengono usate per etichettare istruzioni. La *parte-dichiarazione-etichette* è la prima componente di un blocco (vedi la definizione del *blocco* nel paragrafo 2.8.), e la sua definizione sintattica è

parte-dichiarazione-etichette = "**label**" *etichetta* { "," *etichetta* } ";".

Essa è formata cioè dal simbolo **label,** seguito da una sequenza di interi senza segno separati da virgole. Ad esempio, nello schema di programma (1) sono dichiarate ed usate tre etichette.

```
program esempio (input, output);
label 1,99,777;
const maxvalore = 1000;
var i, j, x : integer;
begin
   ⋮
   777 : a := b;
         ⋮
         if i < j then goto 1;
         ⋮
     1 : repeat P (i); i := i + 1 until i = j;                    (1)
         ⋮
         if x > maxvalore then goto 99;
         ⋮
    99 : writeln ('trovato errore');
         ⋮
         goto 777
         ⋮
end.
```

Il campo d'azione della dichiarazione di un'etichetta è l'intero testo del blocco in cui si trova, a meno di una sua possibile ridichiarazione in un blocco più interno; cioè le regole che determinano il campo d'azione di un'etichetta sono le stesse degli identificatori. Un'etichetta dichiarata in un blocco deve essere usata per etichettare un'istruzione che si trova nella parte istruzioni di quel blocco. Ad ogni modo, all'istruzione così etichettata si può far riferimento da un'istruzione-goto che si trovi nella parte istruzioni di qualsiasi blocco più interno, purché l'etichetta non sia stata ridichiarata ad un livello intermedio.

Ovviamente due istruzioni in un blocco non possono avere la stessa etichetta, altrimenti si avrebbe un'ambiguità nel decidere la destinazione di un'istruzione-goto che faccia riferimento a quella etichetta.

L'istruzione-goto viene talvolta usata per uscire da una struttura di controllo nel caso di errore o di circostanze eccezionali. Si pensi, ad esempio, ad un programma che deve leggere i nomi dei clienti nei primi 30 caratteri di una linea di input, e produrli in output nello stesso formato, fermandosi quando non ci sono più dati di input. Il programma potrebbe essere scritto come in (2).

```
program elenco (input, output);
  procedure uncliente;
  var i : 1..30; c : char;
  begin
    for i := 1 to 30 do
    begin
      read (c);
      write (c)
    end;
    readln;
    writeln
  end;
begin
  repeat
    uncliente
  until eof (input)
end.
```

(2)

Supponiamo ora che il programma debba controllare che i nomi letti contengano soltanto caratteri dell'alfabeto e spazi. Quando viene identificato un errore, si deve far seguire al nome parzialmente stampato un simbolo di segnalazione dell'errore e continuare la stampa con il nome successivo. Questo si potrebbe realizzare modificando la procedura *uncliente* come indicato in (3). In essa viene usata l'istruzione-goto per alterare il normale controllo del ciclo-for, facendo terminare prematuramente l'elaborazione relativa a tale cliente.

```
procedure uncliente;
label 99;
var i : 1..30; c : char;
begin
  for i := 1 to 30 do
  begin
    read (c);
    if ((c < 'A') or (c > 'Z')) and (c <> ' ')
      then begin
             write ('***** ERRORE');
             goto 99
           end;
```

(3)

```
        write (c)
    end;
99 : readln;
     writeln
end;
```

Si noti che, nella procedura sopraindicata, abbiamo assunto che tutti i caratteri compresi tra 'A' e 'Z' dell'insieme di caratteri disponibile siano lettere.

Quando è necessario trasferire il controllo con un'istruzione-goto alla fine di una istruzione-composta, bisogna mettere prima del simbolo **end** un'istruzione etichettata vuota, cioè

```
    begin
      ⋮
      if x < 0
        then begin
               errore := true;
               goto 1
             end;
      ⋮
      . . . ;
1 : end
```

Si noti ancora che, per poter introdurre l'istruzione vuota, un punto e virgola deve precedere l'etichetta.

Un'istruzione-goto all'interno di un'istruzione-composta può trasferire il controllo soltanto ad una istruzione etichettata che si trovi nella stessa istruzione-composta, o in un'istruzione-composta più esterna: cioè non è possibile trasferire il controllo all'interno di un'istruzione-composta. A questo riguardo la sequenza di istruzioni racchiuse dai simboli **repeat** ed **until** in un'istruzione-repeat è da considerarsi un'istruzione-composta. Il frammento di programma (4a) illustra l'uso corretto dell'istruzione-goto, mentre il frammento (4b) mostra un suo uso scorretto.

```
9 :                                    repeat
   ⋮                                     ⋮
   begin                            77 : . . .
     ⋮                                   ⋮
     goto 9;          (4a)              until a < b;          (4b)
     ⋮                                   ⋮
   end                                   goto 77
```

Le istruzioni controllate da un'istruzione-if, un'istruzione-while o un'istruzione-for, non si dovrebbero etichettare. Sebbene la definizione del PASCAL consenta il riferimento ad un'etichetta di questo tipo dall'interno della stessa istruzione etichettata, l'effetto di una tale istruzione-goto può generare confusione, ed il suo uso viene caldamente sconsigliato.

Gli interi usati come etichette di case nelle istruzioni-case, e quelli usati per etichet-

tare istruzioni alle quali si fa riferimento con goto, sono molto simili dal punto di vista del testo, ma concettualmente del tutto indipendenti. Perciò si possono usare etichette come in (5), dove l'etichetta 1 e l'etichetta-di-case 1, sono del tutto diverse, anche se in qualche modo si possono confondere.

```
1 : case p div q of
      1 : S;
      2,3 : T;                                    (5)
      4 : W;
    end;
    ⋮
    goto 1
```

In un'istruzione-goto si può usare un'etichetta non locale per trasferire il controllo fuori da una procedura o funzione ad una istruzione in un blocco più esterno, ma solo se l'istruzione etichettata si trova al livello più esterno di annidamento delle istruzioni del blocco. Nell'esempio precedente, si supponga che sia richiesto di abortire l'elaborazione dell'intera sequenza dei nomi dei clienti non appena viene identificato un errore. Per ottenere ciò possiamo introdurre un'etichetta alla fine del blocco stesso del programma, ed usare questa etichetta nella procedura *uncliente,* come riportato in (6).

```
program elenco (input, output);
label 99;
  procedure uncliente;
  var i : 1..30; c : char;
  begin
    for i := 1 to 30 do
    begin
      read (c);
      if ( (c < 'A' ) or (c > 'Z') ) and (c <> ' ' )
        then begin
               writeln ( ' **** ERRORE');
               writeln ( 'ESECUZIONE INTERROTTA');
               goto 99
             end;
      write (c)
    end;
    readln;
    writeln                                       (6)
  end;
begin
  repeat
    uncliente
  until eof (input);
99 :
end.
```

In questo caso l'istruzione-goto viene usata per sottrarre il controllo al ciclo-for all'in-

terno della procedura *uncliente,* alla chiamata di procedura stessa ed al ciclo del programma all'interno del quale si trova la chiamata.

L'effetto di un'istruzione-goto non locale è, in alcune situazioni, molto potente. Si consideri lo schema di programma (7). La procedura *Q* non è chiamata direttamente dalla parte istruzioni di *P*, ma è chiamata da *R* che a sua volta è stata chiamata da *P*. Perciò l'effetto dell'istruzione **goto** 99 in *Q* è di provocare l'uscita non solo dall'esecuzione di *Q* ma anche dall'esecuzione di *R* (che aveva chiamato *Q*), e quindi di trasferire il controllo all'istruzione di *P* etichettata 99. Se *Q* o *R* sono procedurre ricorsive che hanno già chiamato se stesse più volte, l'esecuzione di **goto** 99 all'interno di *Q* provoca l'uscita da tutte le attivazioni ricorsive di *Q* ed *R,* ed il ritorno all'istruzione 99 di *P*.

Se *P* è a sua volta ricorsiva, l'esecuzione dell'istruzione **goto** 99 in *Q* restituisce sempre il controllo all'attivazione più recente di *P*.

```
procedure P;
label 99;
    procedure Q;
    begin
        ⋮
        Q;
        ⋮
        goto 99;
        ⋮
    end;                                          (7)
    procedure R;
    begin
        ⋮
        Q;
        ⋮
    end;
begin
  ⋮
  R;
99 : ...
  ⋮
end;
```

Abbiamo illustrato l'uso dell'istruzione-goto per trattare situazioni eccezionali non altrimenti esprimibili con le istruzioni offerte dal PASCAL. In teoria l'istruzione-goto, usata con una semplice clausola condizionale come **if** ... **then goto** ..., permette di realizzare l'effetto di tutte le strutture di selezione e ripetizione descritte nel Cap. 6. Lo svantaggio di procedere in questo modo è che la struttura che si intende realizzare non risulta più evidente, e può capitare che nella sua realizzazione si commetta un errore che rimane nascosto. La consistenza strutturale di un programma espresso in termini dei costrutti **if** ... **then** ... **else** ..., **case** ... **of** ..., **while** ... **do** ..., **repeat** ... **until** ..., ecc., è garantita dalle regole stesse del linguaggio. Della consistenza del medesimo programma, espresso con istruzioni-goto, è responsabile il programmatore. È questa libertà di distruggere strutture utili che rende l'istruzione-goto uno strumento di pro-

grammazione pericoloso. È buona regola di programmazione considerare l'istruzione-goto l'ultima risorsa, alla quale bisogna ricorrere solo quando la struttura di controllo richiesta non si può ragionevolmente esprimere in termini delle altre strutture di controllo del PASCAL.

9.

Array

9.1. LA NOZIONE DI ARRAY

Nei Capp. 6 e 7 abbiamo visto come si può costruire un programma che realizza un'azione complessa a partire da azioni elementari come assegnazioni, input ed output, per mezzo di istruzioni composte, selettive ed iterative, e dell'astrazione procedurale. I dati manipolati da tali azioni complesse spesso sono a loro volta complessi, o strutturati; in tal caso i loro componenti elementari sono valori dei tipi semplici introdotti nel Cap. 3. La descrizione dei dati in modo strutturato è utile sia a chiarire la natura dei dati, sia a semplificarne la manipolazione da parte del programma.

Si consideri l'esempio seguente. Una ditta impiega un gruppo di venti venditori (numerati da 1 a 20), cui è pagata una provvigione sulla parte delle loro vendite che supera i due terzi delle vendite medie del gruppo. Si richiede un programma che legge le vendite dei venti venditori, e stampa i numeri di riferimento dei venditori che hanno diritto alla provvigione, insieme con le loro vendite.

Questo problema presenta due aspetti difficili da programmare in PASCAL, usando le funzionalità introdotte sinora. È chiaro che il programma effettuerà elaborazioni simili sui dati di ciascun venditore – lettura delle venti cifre di vendita; calcolo delle vendite medie; confronto delle vendite di ciascun venditore coi due terzi della media per decidere se abbia diritto o no alla provvigione. Tale analogia di elaborazione suggerisce una qualche forma di iterazione. In secondo luogo, nel programma sarà necessario memorizzare le quantità relative ai venditori – lette all'inizio ed usate per determinare la soglia oltre la quale si assegna la provvigione, ma necessarie anche nella parte finale del programma, quando si determina quali venditori riceveranno la provvigione. Ci occorrono dunque venti variabili per conservare le venti cifre di vendita. Potremmo dichiarare venti variabili del tipo appropriato, cioè *vendite*1, *vendite*2, ..., *vendite*20. La stesura del programma – vedi (1) – però risulta poco agevole.

```
begin
  read (vendite1, vendite2, ..., vendite20);
  calcola la media di provvigione;
  if vendite1 > media then writeln (1, vendite1);                    (1)
  if vendite2 > media then writeln (2, vendite 2);
```

```
    ⋮
  if vendite20 > media then writeln (20, vendite20);
end
```

Poichè le variabili *vendite*1, ..., *vendite*20 sono distinte, siamo costretti a scrivere il programma come una successione di istruzioni-if. Analogamente, la ripetitività potenziale della lettura dei valori di vendita, e del calcolo della loro somma, è ostacolata dall'indipendenza di tali variabili. Ovviamente questo è un programma piuttosto noioso da scrivere. E quanto sarebbe più noioso se la ditta occupasse non 20 venditori, ma 200!

Una soluzione preferibile consiste nel considerare queste variabili come componenti di un solo dato *vendite,* denotando il suo *i*-esimo componente mediante la convenzione matematica dell'indice sottoscritto, cioè $vendite_i$. Allora la soluzione del problema, per un numero arbitrario N di venditori, può essere espressa come in (2).

```
begin
  somma := 0.0;
  for i := 1 to N do
    begin
        read (vendite_i);                                    (2)
        somma := somma + vendite
    end;
  livelloprovvigione := 2 / 3 * (somma / N);
  for i := 1 to N do
  if vendite_i > livelloprovvigione
    then writeln (i, vendite_i)
end
```

Questa soluzione comporta l'uso di due nuovi costrutti linguistici, sinora non disponibili:

(a) la possibilità di denotare un gruppo di variabili, od una variabile composta, con un solo identificatore *vendite,* e inoltre
(b) la possibilità di distinguere una particolare variabile appartenente al gruppo, o componente, scrivendo accanto all'identificatore del gruppo un opportuno indice sottoscritto *i.*

In PASCAL un *tipo-array* è definito come segue:

tipo-array = "**array**" "[" *tipo-indice* { "," *tipo-indice* } "]" "**of**" *tipo.*
tipo-indice = *tipo.*

Ovvero un *tipo-array* consiste del simbolo **array** seguito da uno o più *tipi-indice,* separati da virgole e racchiusi tra parentesi quadre, seguiti dal simbolo **of** e dalla definizione del tipo delle singole variabili dell'array (il *tipo-elementi).* Un tipo-indice può essere qualsiasi tipo ordinale, mentre gli elementi dell'array possono essere di qualsiasi tipo. Una variabile di tipo array consiste di un elemento per ciascun valore del tipo-in-

dice, o di un elemento per ciascuna combinazione di valori dei tipi-indice ove siano specificati due o più tipi-indice.

Per esempio,

vendite : **array** [1..20] **of** *real*

è una dichiarazione di variabile array che crea un array di 20 elementi. In questo caso il tipo indice è l'intervallo degli interi 1..20 ed il tipo degli elementi è *real.*

Il tipo indice di un tipo array è spesso, ma non sempre, un sottointervallo degli interi. Si potrebbe per esempio usare un tipo enumerato

colore = (rosso, blu, giallo)

in un tipo array

contafiori : **array** [*colore*] **of** *integer*

per creare un array di tre elementi interi, uno per ciascuno dei valori dell'indice *rosso, blu, giallo.*

Un singolo elemento di una variabile array è denotato come *variabile-indiciata,* cioè scrivendo il nome dell'array seguito dai valori corrispondenti dei tipi indice racchiusi tra parentesi quadre:

variabile-indiciata = variabile-array "[" espressione { "," espressione} "]".
variabile-array = variabile.

Le espressioni racchiuse tra parentesi quadre debbono produrre valori compatibili per assegnazione coi corrispondenti tipi indice.

Per esempio

vendite [14]

denota l'elemento dell'array vendite corrispondente al valore dell'indice 14, mentre

contafiori [*rosso*]

denota l'elemento dell'array *contafiori* corrispondente al valore dell'indice *rosso.*

L'operazione di denotazione con indice prende il nome di *sottoscrittura* della variabile, ed il valore dell'indice viene detto *indice sottoscritto,* perché corrisponde alla notazione matematica convenzionale $vendite_{14}$. In PASCAL, però, l'indice sottoscritto si scrive tra parentesi quadre subito dopo il nome dell'array.

Come si deduce dalla definizione di cui sopra, il valore dell'indice di una variabile indiciata può non essere una costante – è ammessa qualsiasi espressione che produca un valore appartenente al tipo indice dichiarato per l'array. Sono dunque ammesse tutte le seguenti forme:

vendite [*i*]
vendite [*i + j*]
vendite [*trunc (x * 4.2) + j*]

purché le espressioni *i, i+j* e *trunc (x * 4.2) + j* producano tutte valori interi apparte-

nenti all'intervallo 1..20. In generale

vendite [*n*]

è una variabile indiciata ammissibile, purché *n* denoti un valore intero in 1..20. Se il valore di *n* è fuori intervallo allora ci si riferisce ad un componente dell'array che non esiste. In tal caso si ha un errore nell'indice dell'array *(indice fuori limite)*. La maggior parte delle implementazioni segnala il verificarsi di tale condizione di errore durante l'esecuzione di un programma.

Un tipo array permette l'uso di un numero anche grande di variabili di tipo identico, dichiarate e manipolate come (componenti di) una sola variabile array. In combinazione con l'istruzione-for descritta nel Cap. 6, gli array forniscono un compatto e potente strumento di elaborazione di grandi quantità di dati variabili. Per esempio, se si rappresenta un gruppo di 500 variabili reali come array

x : **array** [1..500] **of** *real*

basta un'istruzione per azzerare tutte le variabili, cioè

for *j* := 1 **to** 500 **do** *x* [*j*] := 0

Possiamo adesso presentare un programma completo (3) per risolvere il problema delle commissioni di vendita visto in precedenza. Se il numero dei venditori del gruppo dovesse in seguito cambiare, l'unica modifica necessaria riguarderà il valore della costante *numerovenditori*.

```
program provvigionevendite (input, output);
const numerovenditori = 20;
var vendite : array [1.. numerovenditori] of real;
    i : 1..numerovenditori;
    somma, livelloprovvigione : real;
begin
  somma := 0;
  for i := 0 to numerovenditori do                                    (3)
  begin
    read (vendite [i]);
    somma := somma + vendite [i]
  end;
  livelloprovvigione := somma * 2 / 3 / numerovenditori;
  for i := 1 to numerovenditori do
    if vendite [i] > livelloprovvigione
      then writeln (i : 6, vendite [i] : 11 : 2)
end.
```

Si dice palindroma una frase che (considerando solo le lettere ed ignorando tutti gli spazi e i simboli di punteggiatura) si legge nello stesso modo, sia da sinistra a destra, che da destra a sinistra. Per esempio

AI LATI D'ITALIA

Il programma (4) legge una frase lunga fino a cento lettere, conclusa da un punto, e determina se è una frase palindroma.

```
program palindromo (input, output);
var lettera : array [1..100] of char;
    i       : 0..100;
    j       : 1..100;
    ch      : char;
begin
    { leggi e scrivi la frase memorizzandone tutte le lettere nell'array lettera }
    i := 0; read (ch);
    repeat
      if (ch > = 'A') and (ch < = 'Z')                                    (4)
        then begin
                 i := i + 1;
                 lettera [i] := ch
             end;
      write (ch);
      read (ch)
    until ch = '.';
    { controlla se l'array contiene una frase palindroma }
    j := 1;
    while (j < i) and (lettera [j] = lettera [i]) do
    begin
      j := j+1;
      i := i-1
    end;
    if j > = i
      then write ('e' ' ')
      else write ('non e' ' ');
    write ('una frase palindroma')
end.
```

9.2. ARRAY BIDIMENSIONALI

Finora nei nostri esempi abbiamo usato array dichiarati con un solo tipo indice, per i quali la selezione di un elemento corrisponde alla valutazione di un solo valore di indice. Tali array sono detti *monodimensionali*. Il PASCAL permette d'altronde tipi array con più di un tipo indice. Ad esempio, una pagina di stampa di 66 righe, ciascuna comprendente 120 caratteri, può essere dichiarata come variabile array

pagina : **array** [1..66, 1..120] **of** *char*

Il *j*-esimo carattere della linea *i*-esima di *pagina* sarebbe allora l'elemento denotato da

pagina [*i, j*]

dove *i* assume valori in 1..66, e *j* in 1..120.

La seguente istruzione assegna a ciascuno dei 7920 caratteri componenti la *pagina* lo spazio ' '.

```
for i := 1 to 66 do
  for j := 1 to 120 do pagina [i, j] := ' '
```

Si dice che l'array *pagina* è *bidimensionale*. Si possono dichiarare array con un numero qualsiasi di indici – se un array ha *n* indici è un array *n-dimensionale* ed i suoi elementi si denotano con l'identificatore dell'array seguito da *n* espressioni di indice.

Un array bidimensionale si usa solitamente per rappresentare la nozione matematica di *matrice* (allo stesso modo in cui un array monodimensionale può essere usato per rappresentare un *vettore*). Il programma (5) legge una matrice 6 x8 dal canale di input in un array bidimensionale *A*, calcola le somme di riga in un vettore *B* e quelle di colonna in un vettore *C*, e stampa i tre array nella forma mostrata in Fig. 9.1.

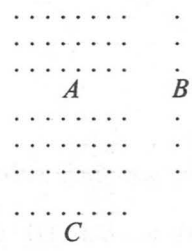


Fig. 9.1

```
program  matrici (input, output);
const    maxriga = 6; maxcolonna = 8;
type     riga = 1.. maxriga;
         colonna = 1.. maxcolonna;
var      A : array [ riga, colonna] of integer;
         B : array [riga] of integer;
         C : array [colonna] of integer;
         i : riga;
         j : colonna;
         somma : integer;
begin                                                    (5)
   { leggi in A i valori della matrice }
   for i := 1 to maxriga do
     for j := 1 to maxcolonna do read (A [i, j]);
   { totalizza in B le somme di riga }
   for i := 1 to maxriga do
   begin
        somma := 0;
        for j := 1 to maxcolonna do somma := somma + A [i, j];
        B [i] := somma
   end;
```

```
    { totalizza in C le somme di colonna }
    for j := 1 to maxcolonna do
    begin
      somma := 0;
      for i := 1 to maxriga do somma := somma + A [i, j];
      C [j] := somma
    end;
    { stampa A, B, C nella forma richiesta }
    for i := 1 to maxriga do
    begin
      for j := 1 to maxcolonna do write A [i, j];
      writeln (B [i] : 15)
    end;
    writeln;
    writeln;
    for j := 1 to maxcolonna do write (C [j])
end.
```

In realtà si potrebbe scrivere un programma equivalente molto più corto, usando solo il vettore *C*. Questo è lasciato per esercizio al lettore.

Gli elementi di un array PASCAL possono essere di qualsiasi altro tipo, ed in particolare possono essere di un altro tipo array. Questo fatto rende possibile la creazione di un array bidimensionale per dichiarazione di un array monodimensionale i cui componenti siano di un altro tipo array monodimensionale. Per esempio, l'array *pagina* visto prima avrebbe potuto essere dichiarato

pagina : **array** [1..66] **of array** [1..120] **of** *char*

Questo array ha sempre 7920 elementi, dei quali il *j*-esimo della *i*-esima riga è denotato

pagina [*i*] [*j*]

Inoltre l'identificatore *pagina* seguito da un solo indice *i*

pagina [*i*]

denota un elemento di tipo **array** [1..120] **of** *char,* e cioè l'*i*-esima riga della pagina rappresentata. In realtà, la definizione del PASCAL stabilisce che quest'ultima dichiarazione di *pagina* è completamente equivalente alla precedente, cosicché le seguenti sono tutte scritture ammissibili con ciascuna delle due forme di dichiarazione:

pagina [*i, j*]
pagina [*i*] [*j*]
pagina [*i*]

9.3. OPERAZIONI SU ARRAY COMPLETI

Un tipo array può essere definito nella dichiarazione di una variabile array, come negli esempi visti finora, oppure gli può essere assegnato un nome con una definizio-

ne di tipo; per esempio

```
type scheda = array [1..90] of char;
     linea = array [1..120] of char;
     pagina = array [1..66] of linea;
     totalimensili = array [1970..1979, mese] of real;
```

questi nomi di tipo possono essere poi usati in una dichiarazione di variabili come la seguente:

```
var scheda1, scheda2 : scheda;
    pagina1 : pagina;
    l : linea;
    pioggia, sole : totalimensili;
```

Se occorre dichiarare array dello stesso tipo in punti diversi del programma, il nome del tipo si deve sempre definire, ma anche se si usa una volta sola questo nome migliora in molti casi la leggibilità del programma.

Se due variabili array sono dello stesso tipo, è possibile allora assegnare i valori di ciascun elemento dell'uno ai corrispondenti elementi dell'altro con una sola istruzione di assegnazione. Ad esempio, l'assegnazione

```
scheda1 := scheda2
```

è equivalente al ciclo-for

```
for i := 1 to 80 do scheda1 [i] := scheda2 [i]
```

ma è più efficiente di quest'ultimo nella maggior parte delle implementazioni.

Come le altre variabili, un array completo può essere passato come parametro attuale ad una procedura o funzione. In tal caso, il tipo array deve essere stato precedentemente definito in un blocco racchiudente la procedura, poiché il tipo del parametro nella lista parametri formali si può specificare solo con un identificatore. Per esempio, se definiamo un tipo array

```
untipo = array [1..10] of integer
```

una funzione che ritorni come risultato la somma dei valori degli elementi di un array di questo tipo potrebbe essere dichiarata come in (6).

```
function somma (a : untipo) : integer;
var i : 1..10; totale : integer;
begin
  totale := 0;
  for i := 1 to 10 do totale := totale + a [i];
  somma := totale
end
```

(6)

Se *array*1 e *array*2 sono variabili dichiarate di *untipo*, allora si può chiamare la funzione usandole come parametri;

```
if somma (array1) > somma (array2) then array2 := array1
```

La procedura (7) esegue la moltiplicazione righe per colonne di due matrici quadrate *A* e *B*, scrivendo il risultato in una matrice *C*, dove *A*, *B* e *C* sono matrici di tipo

matricequadrata = **array** [1..*N*,1..*N*] **of** *real*

e *N* è una costante. Poiché l'array *C* è modificato dalla procedura, va dichiarato come parametro per variabile, come in (7).

```
procedure moltiplicamatrici (A, B: matricequadrata; var C: matricequadrata);
var i, j, k : 1..N; somma : real;
begin
  for j := 1 to N do
    for k := 1 to N do                                              (7)
    begin
      somma := 0;
      for i := 1 to n do
        somma := somma + A [j, i] * B [i, j];
      C [j, k] := somma;
    end
end
```

Le assegnazioni di array e i passaggi di array per valore implicano l'operazione di copia dell'intero array, ovvero di ciascuno dei suoi valori componenti. Se gli array hanno grandi dimensioni, questa operazione è assai dispendiosa in termini di tempo e memoria occupata dal parametro per valore. Il programmatore dovrebbe tener conto di questo, nel manipolare array completi. Può infatti rendersi necessario durante la fase di progetto di un programma minimizzare le operazioni di copia.

PROGRAMMA 8 (Calcolo di banconote e monete)

Ogni settimana l'ufficio paghe di una ditta deve stabilire il numero di banconote e monete dei tagli esistenti da richiedere alla banca per poter confezionare le buste paga. Questa operazione viene realizzata da un programma, il cui input è una serie di linee contenenti i dati relativi al personale. Ogni linea contiene il nome di un impiegato (nelle posizioni 1-20), seguito dalla sua paga della settimana.

Il programma deve calcolare il modo in cui ciascuna busta paga va confezionata, usando il minimo numero di banconote e monete. I tagli disponibili sono in banconote da $10, $5, $1, e in monete da 25, 10, 5, 2 e 1 centesimi ($1 = 100 centesimi). Per assolvere allo scopo il programma dovrà stampare, per ciascun impiegato, il suo nome ed il contenuto dettagliato della sua busta paga. A questo dovrà seguire il prelievo totale richiesto alla banca, in termini del numero di pezzi di ciascuna banconota o moneta. Si assume che nessun impiegato riceva più di $100 alla settimana.

```
for ogni impiegato do
  begin
    leggi e scrivi nome e paga;
    calcola banconote e monete necessarie;                          (8)
    stampa banconote e monete necessarie;
```

```
        aggiorna i totali di paga, banconote e monete
    end
stampa totali
```

La struttura base del programma, indicata in (8), è quella di un ciclo che elabora i dati di ciscun impiegato ed accumula i totali richiesti. In questa fase il solo problema significativo consiste nell'esprimere *calcola banconote e monete necessarie.* Noi lo risolviamo con l'istruzione di procedura

trasforma (pagaincentesimi, numeropezzi)

dove il primo parametro riporta, per valore, la paga dell'impiegato in centesimi (nell'intervallo 1..10000), ed il secondo è un parametro per variabile di tipo

arraydenaro = **array** [*tagli*] **of** *integer*

L'array *numeropezzi* sarà calcolato dalla procedura *trasforma,* e indicherà il numero di banconote e monete occorrenti nei vari tagli per poter confezionare *pagaincentesimi* nel modo più efficiente. Il tipo *tagli* è definito per enumerazione dei tipi correnti di banconote e monete; per esempio

tagli = (diecidollari, cinquedollari, undollaro, venticinquecentesimi, diecicentesimi, cinquecentesimi, duecentesimi, uncentesimo);

Per ora non ci preoccupiamo di raffinare ulteriormente la procedura *trasforma.*

Il programma principale deve conservare il totale delle paghe e del numero di banconote e monete necessarie per ciascun taglio. Introduciamo allora la variabile

pagatotale : *real*;

(poiché vediamo la paga di ciascun impiegato come numero reale) ed un array

totale : *arraydenaro*

inizializzato a zero e poi usato per conservare i totali parziali delle banconote e delle monete.

L'elaborazione dei dati di ciascun impiegato comprende la lettura di una linea di input, e tutto il processo termina quando sono stati tutti letti i dati di input. Il programma precedente si potrà così estendere come riportato in (9).

```
begin
  pagatotale := 0;
  inizializza il totale azzerandolo;
  scrivi l'intestazione;
  while not eof (input) do
  begin                                                    (9)
    leggi la linea contenente nome e paga;
    scrivi nome e paga;
    pagatotale := pagatotale + paga;
    trasforma (pagaincentesimi, numeropezzi);
    aggiorna i totali con numeropezzi;
    scrivi il numero necessario di banconote e monete
```

```
    end;
    stampa i totali
end
```

L'istruzione *inizializza il totale azzerandolo* è un ciclo-for

```
for t := diecidollari to uncentesimo do totale [t] := 0;
```

mentre *aggiorna totali con numeropezzi* e *scrivi il numero necessario di banconote e monete* possono essere raccolte nell'unico ciclo (10). Il resto del programma principale trova ora un'immediata codifica in PASCAL. Occorrebbe stampare inoltre le appropriate intestazioni, in modo da rendere più leggibile l'output.

```
for t := diecidollari to uncentesimo do
begin
    write (numeropezzi [t]);
    totale [t] := totale [t] + numeropezzi [t]                    (10)
end
```

Passiamo ora ad esaminare in dettaglio la procedura *trasforma.*

```
procedure trasforma (somma : paga; var numeropezzi : arraydenaro)
```

La strategia per esprimere una somma in termini del minimo numero di banconote e monete consiste, ovviamente, nell'uso di quante più banconote da $10 sia possibile, e poi di confezionare il resto con quante più banconote da $5 possibile, poi con pezzi da $1, da 25 centesimi, ecc.. Il corpo di *trasforma* va allora espresso come in (11).

```
for ciascun taglio in ordine decrescente di valore do
begin
    calcola il numero massimo di pezzi usabili di quel taglio;    (11)
    ricalcola la somma da considerare
end
```

Per ora, in realtà, non possiamo usare un ciclo-for, perché il valore associato a ciascun taglio deve essere specificato separatamente; ad esempio potremmo scrivere le istruzioni riportate in (12)

```
numeropezzi [diecidollari] := somma div 1000;
somma := somma mod 1000;
numeropezzi [cinquedollari] := somma div 500;
somma := somma mod 500;                                           (12)
:
numeropezzi [cinquecentesimi] := somma div 5;
somma := somma mod 5;
numeropezzi [uncentesimo] := somma div 1;
```

Se però introduciamo un array

```
valore : array [tagli] of 1..1000
```

che memorizzi il valore associato a ciascun taglio, possiamo scrivere la serie di istru-

zioni (12) con un ciclo-for. L'array *valore* va inizializzato in modo opportuno:

```
valore [diecidollari] := 1000;
valore [cinquedollari] := 500;
        .
valore [uncentesimo] := 1;
```

dopodiché il ciclo-for è quello mostrato in (13).

```
for t := diecidollari to uncentesimo do
begin                                                    (13)
  numeropezzi [t] := somma div valore [t];
  somma := somma mod valore [t]
end
```

Il Listato 8 mostra il programma completo insieme con un output campione.

LISTATO 8

```
PROGRAM BUSTEPAGA (INPUT,OUTPUT);

TYPE PAGA          = 0..10000;
     TAGLI         = (DIECIDOLLARI, CINQUEDOLLARI, UNDOLLARO,
                      VENTICINQUECENTESIMI,DIECICENTESIMI,
                      CINQUECENTESIMI, UNCENTESIMO);
     ARRAYDENARO = ARRAY [TAGLI] OF INTEGER;

VAR  VALORE : ARRAY [TAGLI] OF 1..1000;
     TOTALE, NUMEROPEZZI : ARRAYDENARO;
     T : TAGLI;
     PAGAINCENTESIMI : PAGA;
     X, PAGATOTALE : REAL;
     CH : CHAR;
     I : 1..20;

     PROCEDURE TRASFORMA(SOMMA : PAGA;
                         VAR NUMEROPEZZI : ARRAYDENARO);
     VAR  T: TAGLI;
     BEGIN
        FOR T:= DIECIDOLLARI TO UNCENTESIMO DO
           BEGIN
              NUMEROPEZZI[T]:= SOMMA DIV VALORE[T];
              SOMMA:= SOMMA MOD VALORE[T]
           END
     END; (* TRASFORMA *)

BEGIN
   FOR T:= DIECIDOLLARI TO UNCENTESIMO DO
      TOTALE[T]:= 0;
   VALORE[DIECIDOLLARI]:= 1000;
   VALORE[CINQUEDOLLARI]:= 500;
   VALORE[UNDOLLARO]:= 100;
   VALORE[VENTICINQUECENTESIMI]:= 25;
```

```
VALORE[DIECICENTESIMI]:= 10;
VALORE[CINQUECENTESIMI]:= 5;
VALORE[UNCENTESIMO]:= 1;
WRITE('   IMPIEGATO            ');
WRITELN('PAGA   $10   $5   $1  25C  10C   5C   1C');
WRITELN;
WHILE NOT EOF (INPUT) DO
BEGIN
   FOR I:= 1 TO 20 DO BEGIN READ(CH); WRITE(CH) END;
   READLN(X); WRITE(X:7:2);
   PAGAINCENTESIMI:= ROUND(X * 100);
   PAGATOTALE:= PAGATOTALE + X;
   TRASFORMA(PAGAINCENTESIMI,NUMEROPEZZI);
   FOR T:= DIECIDOLLARI TO UNCENTESIMO DO
   BEGIN
      WRITE(NUMEROPEZZI[T]:5);
      TOTALE[T]:= TOTALE[T] + NUMEROPEZZI[T]
   END;
   WRITELN
END;


   WRITELN;
   WRITELN('*** PRELEVAMENTO TOTALE ***');
   WRITELN;
   WRITE(PAGATOTALE:27:2);
   FOR T:= DIECIDOLLARI TO UNCENTESIMO DO WRITE(TOTALE[T]:5)
END.
```

```
   IMPIEGATO            PAGA  $10   $5   $1  25C  10C   5C   1C

ALEC SLOAN              9.00    0    1    4    0    0    0    0
GORDON QUIRK           58.03    5    1    3    0    0    0    3
MARTIN MACDONALD        9.74    0    1    4    2    2    0    4
DENIS DAKES            16.17    1    1    1    0    1    1    2
SAMUEL JENKINS         27.42    2    1    2    1    1    1    2

*** PRELEVAMENTO TOTALE ***

                      120.36    8    5   14    3    4    2   11
```

9.4. ARRAY IMPACCATI

Come abbiamo appena visto gli array possono essere di notevoli dimensioni in termini di memoria occorrente per conservare i loro componenti; per esempio l'array

A : **array** [−1000..1000] **of** *boolean*

comprende 2001 dati. Un programma contenente parecchi array del genere può richiedere tanta memoria da non poter essere eseguito su un dato calcolatore. In alcuni casi è possibile ridurre la quantità di memoria necessaria ad un array – specie se il ti-

po degli elementi è *boolean* o *char,* oppure è un'enumerazione o un sottointervallo.

Se alla definizione di un tipo-array si premette il simbolo **packed,** come in

B : **packed array** [−1000..1000] **of** *boolean*

si richiede all'implementazione PASCAL di tentare di minimizzare la memoria usata dall'array (compattando strettamente, mediante "impaccamento", gli elementi dell'array nella memoria del calcolatore).

L'impaccamento può aumentare il tempo di accesso ai singoli componenti dell'array, cosicché la decisione di definire un array impaccato rappresenta un compromesso tra quantità di memoria occupata da un programma e velocità di esecuzione. Se però l'operazione più frequentemente eseguita sull'array è la copia, attraverso operazioni di assegnazione o di passaggio per valore, allora va tenuto presente che l'impaccamento può far diminuire *sia* la memoria occorrente al programma *sia* il suo tempo di esecuzione.

L'effetto dell'impaccamento di un array dipende dalla natura dei suoi elementi, e dalla particolare implementazione PASCAL. In certi casi può non avere alcun effetto, nè sulla memoria occupata, nè sul tempo di esecuzione, rispetto al caso in cui l'array non sia impaccato.

In generale il significato di un programma non è influenzato dal fatto che un array sia impaccato. L'impaccamento è, d'altronde, considerato un aspetto intrinseco del tipo-array, cosicché risulta modificato il contesto in cui può essere usato un array oppure i suoi componenti. In particolare, un elemento di un array impaccato non può essere passato come parametro attuale per variabile ad una procedura o funzione. Se abbiamo dunque una procedura intestata:

procedure *P* (**var** *q* : *boolean*)

e gli array *A* e *B* dichiarati come sopra, allora

P (*A* [*i*])

è una chiamata ammessa, mentre non lo è

P (*B* [*i*])

Se una parte di un programma fa spesso riferimento agli elementi di un array impaccato, ciò può avere un pessimo effetto sulla velocità di esecuzione di quella parte del programma. Se l'impaccamento è desiderabile per altre ragioni, un possibile rimedio potrebbe consistere nel disimpaccare i componenti dell'array in un array locale non impaccato, prima di entrare nella parte del programma che fa frequenti riferimenti ai componenti; successivamente gli elementi dell'array potranno essere reimpaccati. Il PASCAL fornisce due procedure standard che permettono il trasferimento tra array impaccati o meno con una sola istruzione.

Siano date le seguenti dichiarazioni:

U : **array** [*a..b*] **of** *untipo*;
P : **packed array** [*c..d*] **of** *untipo*;

La chiamata di procedura standard

pack (U, i, P)

dove *i* è un indice ammissibile dell'array *U*, ha l'effetto di copiare elementi dall'array non impaccato *U*, a cominciare dall'elemento *U*[*i*], nell'array impaccato *P*, a cominciare dal suo primo elemento *P*[*c*], finchè *P* non è pieno. L'effetto è equivalente a

```
k := i;
for j := c to d do
begin
  P[j] := U[k];
  k := succ (k)
end
```

Si noti che l'array *U* deve contenere un numero di elementi, da *U*[*i*] in avanti, sufficiente a riempire *P*, altrimenti si verifica un errore di indice di array.

La chiamata di procedura standard

```
unpack (P, U, i)
```

ha l'effetto inverso, cioè copia elementi dall'array impaccato *P*, a partire dal primo, *P*[*c*], nell'array non impaccato *U*, a partire dall'elemento *U*[*i*], finchè l'array impaccato *P* non è pieno. L'effetto è così equivalente a

```
k := i;
for j := c to d do
begin
  U[k] := P[j];
  k := succ (k)
end
```

Di nuovo, l'array *U* deve avere, da *U*[*i*] in avanti, elementi sufficienti a contenere gli elementi di *P*.

9.5. STRINGHE

Sebbene un array impaccato ammetta, in generale, lo stesso insieme di operazioni di un array non impaccato, in PASCAL c'è una classe di array impaccati con caratteristiche ed operazioni speciali. Nel Cap. 2 sono state introdotte le stringhe, definite come sequenze di caratteri racchiusi da apici. Una stringa di lunghezza *n* caratteri (esclusi gli apici) è considerata una costante di tipo

packed array [1..*n*] **of** *char*

Per esempio,

'OGGI PIOVE'

è una costante di tipo

packed array [1..10] **of** *char*

La variabili di tipo **packed array** [1..*n*] **of** *char* sono dette *variabili stringa di lunghezza n*, e si possono usare per memorizzare stringhe di tale lunghezza.

Abbiamo già visto che il valore di un array può essere assegnato ad un'altra variabi-

le array dello stesso tipo. Se definiamo un tipo

nome = **packed array** [1..12] **of** *char*

e dichiariamo

*nome*1, *nome*2 : *nome*

variabili di tipo *nome*, allora è permessa l'assegnazione

*nome*1 := *nome*2

come per qualsiasi altra coppia di array dello stesso tipo. Per di più, come valore da assegnare può essere usata qualsiasi stringa di 12 caratteri; ad esempio

*nome*1 := *'JOHN F JONES'*

Un valore stringa, variabile o costante, è compatibile per assegnazione con qualsiasi tipo stringa della stessa lunghezza.

Come abbiamo visto nel Cap. 5, le procedure standard *write* e *writeln* permettono l'output di stringhe di lunghezza qualsiasi, come in

writeln ('la risposta e '' ', x + y)

In questo modo è possibile stampare anche il valore di una variabile stringa. Per esempio l'istruzione

*writeln ('NOME:', nome*1*)*

stampa due stringhe, una costante e una variabile, per produrre una linea di output della forma

NOME: JOHN F JONES

NB Si noti però che tale possibilità vale solo per le variabili stringa, e non per gli altri array impaccati e no. Notare pure che non è possibile leggere in questo modo un valore di una variabile stringa – va letto carattere per carattere, per esempio così:

for *i* := 1 **to** 12 **do** *read (nome*1 [*i*]*)*

Gli operatori relazionali =, <>, <, <=, >, >= possono essere usati avendo come operandi due stringhe della stessa lunghezza. Per esempio si può scrivere

if *nome*1 = *'JOHN F JONES'* **then**...

Il significato degli operatori di eguaglianza e disuguaglianza, = e <>, è ovvio, tenendo presente che i due operandi debbono avere la stessa lunghezza. Il significato degli operatori di ordinamento < <= > >= è definito dalla normale convenzione lessicografica usata per ordinare le parole in un dizionario. L'ordine di due operandi stringhe è determinato dall'ordine della prima coppia di caratteri corrispondenti che differiscono. Così

'AZZZZ' < 'BCCCC' dà *true* perché *'A' < 'B'*
'AXCYE' > 'AXCDZ' dà *true* perché *'Y' > 'D'*

NB L'ordinamento è comunque definito solo per stringhe della stessa lunghezza, diver-

samente che in un dizionario, dove sono ordinate parole di lunghezza diversa.

Notare, infine, che gli operatori relazionali sono definiti solo per valori stringhe, e non per altri array impaccati o meno.

PROGRAMMA 9 (Costruzione di una lista di concordanze)

Si richiede un programma che esamini un testo e produca una lista, in ordine alfabetico, di tutte le parole distinte che appaiono nel testo. Per esempio all'esame dell'input

tanto gentile e tanto onesta pare

dovrebbe corrispondere l'output

e
gentile
onesta
pare
tanto

Si può assumere che non esistano parole più lunghe di 16 lettere, e che nessuna parola contenga apostrofi o trattini.

Il programma richiede la costruzione di una lista di tutte le parole diverse che appaiono nel testo. Definiamo il tipo *parola* come stringa lunga 16

parola = **packed array** [1..16] **of** *char*

La lista sarà rappresentata come array

indice: **array** [1.. *ampiezzaindice*] **of** *parola*

mentre la variabile

numeroelementi: 0..*ampiezzaindice*

servirà a definire la lunghezza della lista, cioè il numero di elementi significativi contenuti nell'*indice*.

Il contenuto della lista, una volta esaminato tutto l'input, verrà stampato in ordine alfabetico. Dobbiamo perciò decidere se costruire la lista in modo tale che i suoi elementi siano già in ordine alfabetico (inserendo nuovi elementi senza alterare l'ordinamento), oppure se costruire una lista disordinata ed ordinarla immediatamente prima dell'output della lista finale. Scegliamo il primo metodo, che è un pò più efficiente. Costruiamo allora la lista in modo che i contenuti dell'array *indice* soddisfino sempre la relazione

$\forall$ *i, j tali che* $i < j \leq$ *numeroelementi, indice* $[i] <$ *indice* $[j]$

La struttura base del programma è un ciclo che determina la parola successiva sul canale di input e, se necessario, la inserisce in lista. Il ciclo (14) termina quando il canale di input è stato letto completamente.

```
finetesto := false;
prendiparolasuccessiva;
while not finetesto do
begin                                                    (14)
  aggiungiallalista;
  prendiparolasuccessiva
end;
stampatabella
```

La procedura *prendiparolasuccessiva* scandisce il testo alla ricerca della parola successiva. Se l'input è esaurito, allora alla variabile globale booleana *finetesto* viene assegnato il valore *true,* altrimenti si memorizza la variabile successiva in una variabile globale

```
lettereparola : parola
```

cosicché la procedura *aggiungiallalista* può, se necessario, aggiungere questa parola alla lista delle parole trovate nel testo. La struttura della procedura *prendiparolasuccessiva* è quindi

```
scandisci il testo fino a trovare una parola o la fine del testo;
if trovata lettera
  then scandisci e memorizza la parola
  else finetesto := true
```

Ogni esecuzione di questa procedura legge tutti i caratteri non alfabetici precedenti la parola successiva, le lettere che formano la parola, ed il carattere seguente. Nell'ipotesi che il canale di input sia strutturato in linee, il carattere letto quando *eof (input)* diventa *true* deve essere uno spazio corrispondente al fine linea conclusivo. Così *eof (input)* può diventare vero durante la scansione dei caratteri non-alfabetici, nel qual caso a *finetesto* sarà assegnato il valore *true,* oppure leggendo il carattere immediatamente successivo ad una parola, nel qual caso *finetesto* diventa *true* con la chiamata seguente di *prendiparolasuccessiva.*

Introducendo una variabile *ch* di tipo *char* possiamo riscrivere le istruzioni precedenti come riportato in (15).

```
if not eof (input)
  then repeat read (ch)
       until ch è una lettera or eof (input);              (15)
if not eof (input)
  then scandisci e memorizza la parola
  else finetesto := true
```

Nell'ipotesi che le lettere formino un sottointervallo continuo dei valori di tipo *char,* come succede in molte implementazioni, la condizione *ch è una lettera* può essere scritta

```
(ch >= 'A') and (ch <= 'Z')
```

Se non è così si renderà necessario un test più complicato. Nel Cap. 11 è spiegato un metodo alternativo per scrivere condizioni come *ch è una lettera,* usando una costante di tipo insieme.

Il passo *scandisci e memorizza la parola* deve scandire il testo fino al primo carattere non-lettera, memorizzare le lettere *lettereparola* lette nell'array e riempire di spazi tale array se sono state lette meno di 16 lettere. Introducendo una variabile *lunghezza* per contare le lettere memorizzate, possiamo scrivere il blocco (16). In esso abbiamo assunto che l'ultimo carattere del canale di input non può essere una lettera. Quindi *eof(input)* può diventare true solo se si trova la fine di una parola, e viene considerato nella successiva chiamata di *prendiprossimaparola.*

```
begin
  lunghezza := 0;
  repeat
    lunghezza := lunghezza + 1;
    lettereparola [lunghezza] := ch;                    (16)
    read (ch)
  until (ch < 'A') or (ch > 'Z');
  for lunghezza := lunghezza + 1 to 16 do lettereparola [lunghezza] := ' '
end
```

In certe implementazioni può risultare più efficiente raccogliere le lettere della parola scandita in un array non impaccato di caratteri, e poi impaccarle nella variabile stringa *lettereparola* usando la procedura standard *pack.*

La procedura *aggiungiallalista* prende la parola in *lettereparola* e cerca nell'*indice* per determinare se tale parola è già stata trovata nel testo. Se no, la nuova parola va inserita in modo da mantenere l'ordinamento alfabetico degli elementi dell'*indice.* Lo schema di *aggiungiallalista* è così

```
begin
  cerca nella lista;
  crea un nuovo elemento nella lista se occorre
end
```

Noi faremo una ricerca lineare nella lista: dal momento che gli elementi di *indice* sono in ordine alfabetico ascendente, cominciamo la ricerca dal primo elemento di *indice* proseguendola finché troviamo la parola o la fine della lista o una parola di ordine alfabetico maggiore.

Introduciamo le variabili locali

```
i : integer;
trovataposizione : boolean;
```

per scrivere l'azione *cerca nella lista.*

```
i := 1; trovataposizione := false;
while (i <= numeroelementi) and not trovataposizione do
  if indice [i] < lettereparola
    then i := i + 1
    else trovataposizione := true
```

La variabile booleana *trovataposizione* è necessaria perché il PASCAL non specifica che il secondo operando *b* di un'espressione booleana *a* **and** *b* non va valutato se il pri-

mo, *a*, è falso (vedi par. 4.2). Se dunque scriviamo

while *(i <= numeroelementi)* **and** *(indice [i] <= parolaletta)* **do** *i := i + 1*

quando *i* raggiunge *numeroelementi* + 1, si tenta di valutare *indice [i]*, con accesso ad un elemento non assegnato dell'array, o persino con un errore di indice fuori limite (se *numeroelementi = ampiezzaindice*).

Ci sono metodi più veloci della ricerca lineare per scandire una lista ordinata, come quella contenuta in *indice*. Nella ricerca binaria, per esempio, si esamina prima il punto di mezzo della lista per determinare in quale metà della lista può trovarsi la parola richiesta. Si esamina poi il punto di mezzo della metà individuata, e così via, fino a trovare la parola oppure a stabilire che non è compresa nella lista. Se la lista contiene 2^n elementi, allora tale tecnica richiede al più *n* confronti. Il programma presentato in questo esempio usa una ricerca lineare – si lascia come esercizio al lettore la modifica del programma per renderlo più efficiente mediante la ricerca binaria.

L'istruzione *crea un nuovo elemento nella lista se occorre* è esprimibile come riportato in (17).

```
if trovataposizione
  then begin                                                    (17)
         if indice [i] <> parolaletta
           then aggiungi un nuovo elemento (in posizione i)
       end
  else aggiungi un nuovo elemento (alla fine della lista)
```

In ogni caso la posizione del nuovo elemento è indicata dal valore di *i* all'uscita dal ciclo di ricerca nella lista, cosicché ogni processo di inserimento si può scrivere sotto forma di chiamata ad una procedura locale *inserisciparola*.

Per aggiungere una nuova parola nell'indice si incrementa *numeroelementi* di 1 e si spostano di una posizione verso il basso tutte le parole alfabeticamente maggiori della nuova, facendole spazio. Per evitare di sovrapporre elementi, il riposizionamento comincerà dall'ultimo elemento della lista corrente. Ricordando che la posizione della nuova parola è data dal valore di *i* all'uscita dal ciclo di ricerca, la procedura *inserisciparola* diviene come in (18).

```
procedure inserisciparola;
var j: integer;
begin
  if numeroelementi < ampiezzaindice
    then begin
           numeroelementi := numeroelementi + 1;
           for j := numeroelementi downto i + 1 do
             indice [j] := indice [j - 1];                      (18)
           indice [i] := lettereparola
         end
end
```

Infine, la stampa dell'indice si ottiene con un semplice ciclo che stampa gli elementi significativi di *indice*. Essendo questi rappresentati come stringhe, si possono stam-

pare direttamente:

for *i* := 1 **to** *numeroelementi* **do** *writeln (indice [i])*

Il Listato 9 mostra il programma completo ed un output campione prodotto dall'esecuzione del programma sull'input seguente:

```
QUOT SINT GENERA PRINCIPATUM ET QUIBUS MODIS ACQUIRANTUR

TUTTI GLI STATI , TUTTI E DOMINI CHE HANNO AVUTO ED HANNO
IMPERIO SOPRA GLI UOMINI, SONO STATI E SONO O REPUBBLICHE O
PRINCIPATI .

     I PRINCIPATI SONO, O EREDITARII, DE QUALI EL SANGUE DEL LORO
SIGNORE NE SIA SUTO LUNGO TEMPO PRINCIPE , O E SONO NUOVI .
```

LISTATO 9

```
PROGRAM CONCORDANZE (INPUT,OUTPUT);

CONST MAXPAROLA       = 16;
      AMPIEZZAINDICE = 200;

TYPE  LUNGHEZZAPAROLA = 1..MAXPAROLA;
      PAROLA          = PACKED ARRAY [LUNGHEZZAPAROLA] OF CHAR;

VAR   INDICE          : ARRAY [1..AMPIEZZAINDICE] OF PAROLA;
      NUMEROELEMENTI  : 0..AMPIEZZAINDICE;
      PAROLALETTA     : PAROLA;
      FINETESTO       : BOOLEAN;

      PROCEDURE PRENDIPAROLASUCCESSIVA;
      VAR  CH : CHAR;
           LUNGHEZZA: 0..MAXPAROLA;
      BEGIN
         IF NOT EOF(INPUT)
            THEN REPEAT
                    READ(CH)
                 UNTIL ((CH >= 'A') AND (CH <= 'Z'))
                                        OR EOF(INPUT);
         IF NOT EOF(INPUT)
            THEN BEGIN
                    LUNGHEZZA:= 0;
                    REPEAT
                       LUNGHEZZA:= LUNGHEZZA + 1;
                       PAROLALETTA[LUNGHEZZA]:= CH;
                       READ(CH)
                    UNTIL (CH < 'A') OR (CH > 'Z');
                    FOR LUNGHEZZA:= LUNGHEZZA+1 TO MAXPAROLA DO
                       PAROLALETTA[LUNGHEZZA]:= ' '
                 END
            ELSE FINETESTO:= TRUE
      END; (* PRENDIPAROLASUCCESSIVA *)
```

```
      PROCEDURE AGGIUNGIALLALISTA;
      VAR  TROVATAPOSIZIONE: BOOLEAN;
           I: 1..AMPIEZZAINDICE;

           PROCEDURE INSERISCIPAROLA;
           VAR J: 1..AMPIEZZAINDICE;
           BEGIN
              IF NUMEROELEMENTI < AMPIEZZAINDICE
                 THEN BEGIN
                         NUMEROELEMENTI:= NUMEROELEMENTI + 1;
                         FOR J:= NUMEROELEMENTI DOWNTO I + 1 DO
                            INDICE[J]:= INDICE[J - 1];
                         INDICE[I]:= PAROLALETTA;
                      END
           END; (* INSERISCIPAROLA *)

      BEGIN (* AGGIUNGIALLALISTA *)
         I:= 1;
         TROVATAPOSIZIONE:= FALSE;
         WHILE (I <= NUMEROELEMENTI) AND
               NOT TROVATAPOSIZIONE DO
            IF INDICE[I] < PAROLALETTA
               THEN I:= I + 1
               ELSE TROVATAPOSIZIONE:= TRUE;
            IF TROVATAPOSIZIONE
               THEN BEGIN
                       IF INDICE[I] <> PAROLALETTA
                         THEN INSERISCIPAROLA
                    END
               ELSE INSERISCIPAROLA
      END; (* AGGIUNGI ALLA LISTA *)

      PROCEDURE STAMPATABELLA;
      VAR  I : 1..AMPIEZZAINDICE;
      BEGIN
        WRITELN('*** INDICE ***');
        WRITELN;
        FOR I:= 1 TO NUMEROELEMENTI DO
           WRITELN(INDICE[I])
      END; (* STAMPATABELLA *)

BEGIN (* PROGRAMMA PRINCIPALE *)
   FINETESTO:= FALSE;
   NUMEROELEMENTI:= 0;
   PRENDIPAROLASUCCESSIVA;
   WHILE NOT FINETESTO DO
   BEGIN
      AGGIUNGIALLALISTA;
      PRENDIPAROLASUCCESSIVA
   END;
   STAMPATABELLA
END.
```

```
*** INDICE ***

ACQUIRANTUR
AVUTO
CHE
DE
DEL
DOMINI
E
ED
EL
EREDITARII
ET
GENERA
GLI
HANNO
IMPERIO
LORO
LUNGO
MODIS
NE
NUOVI
O
PRINCIPATI
PRINCIPATUM
PRINCIPE
QUALI
QUIBUS
QUOT
REPUBBLICHE
SANGUE
SIA
SIGNORE
SINT
SONO
SOPRA
STATI
SUTO
TEMPO
TUTTI
UOMINI
```

9.6. PARAMETRI ARRAY CONFORMI

La possibilità di passare array come parametri, come descritto nel paragrafo 9.3, esiste se i parametri attuali corrispondenti ad un dato parametro formale in chiamate differenti della procedura o funzione interessata sono di tipo identico.

Può essere utile in certi casi definire una procedura o funzione i cui parametri attuali variano tra una chiamata ed un'altra per il numero di elementi che contengono. Per esempio, possiamo pensare ad una funzione *somma* analoga a quella definita nel par. 9.3, che potrebbe essere usata in un punto per calcolare la somma dei dieci elementi

di un array dichiarato come segue:

```
X : array [1..10] of integer;
```

ed in un'altro punto per calcolare la somma dei 100 elementi di un array dichiarato

```
Y : array [0..99] of integer;
```

Lo schema della funzione richiesta è analogo a quello dato nel par. 9.3, e si potrebbe scrivere così:

```
function somma (...a:....?): integer;
var i, totale: integer;
begin
  totale := 0;
  for i := primo to ultimo do totale := totale + a [i];
  somma := totale
end
```

dove *primo* e *ultimo* rappresentano gli indici iniziale e finale dell'array parametro attuale. Il problema sta nel trasmettere tali valori alla funzione, e nel descrivere l'array stesso nella lista parametri formali della funzione. Per risolvere questo problema, il PASCAL permette di definire *parametri array conformi,* per quanto questa sia una opzione che le implementazioni *non sono* obbligate a fornire. Se si usa il meccanismo del parametro array conforme, il problema di cui sopra può essere risolto con una funzione definita nella forma seguente:

```
function somma (var a : array [primo..ultimo : integer] of integer): integer;
var i, totale : integer;
begin
  totale := 0;
  for i := primo to ultimo do totale := totale + a [i];
  somma := totale
end
```

In questa forma la lista parametri formali specifica che *a* è un array monodimensionale di interi, ma non indica l'intervallo esatto di valori usabili come indici; si dice solo che sono interi, e che i valori iniziale e finale saranno chiamati *primo* e *ultimo*.

Il parametro *a* si chiama *parametro array conforme,* ed il corrispondente parametro attuale deve adeguarsi allo schema di *a*: deve essere un array monodimensionale di interi con tipo indice sottointervallo degli interi. Per ogni parametro attuale passato, gli identificatori *primo* ed *ultimo* denotano i limiti inferiore e superiore dell'intervallo dell'indice, e possono essere usati nel corpo della funzione per ottenere i valori di tali limiti.

Un parametro array conforme, in una lista di parametri formali, si scrive nella sezione parametri per variabile o per valore in forma di *schema-array-conforme,* definito come segue:

```
schema-array-conforme = schema-array-impaccato-conforme |
                        schema-array-non-impaccato-conforme.
```

schema-array-impaccato-conforme = "**packed**" "**array**" "[" *specifica-limiti* "]" "**of**" *identificatore-tipo*.
schema-array-non-impaccato-conforme = "**array**" "[" *specifica-limiti* { "," *specifica-limiti* } "]" "**of**" (*identificatore-tipo* | *schema-array-conforme*).
specifica-limiti = *identificatore* ".." *identificatore* ":" *identificatore-tipo-ordinale*.

Uno schema di array conforme definisce il numero di dimensioni di ciascun parametro array conforme dichiarato nella sezione, il tipo dei suoi elementi ed il tipo ospite cui deve appartenere l'intervallo dell'indice di ciascuna dimensione. Non definisce però i limiti inferiore e superiore di tale intervallo: sono introdotti invece degli identificatori ausiliari che denotano tali limiti. Questi identificatori di limite possono essere usati, dentro la procedura o funzione che manipola il parametro array conforme, in espressioni che usano i limiti inferiore e superiore dell'indice corrispondente del parametro attuale, come abbiamo visto nel caso della funzione *somma*.

La categoria sintattica

identificatore-limite = *identificatore*.

si introduce per denotare la classe degli identificatori così definiti. Il loro uso nelle espressioni è permesso dalle seguenti definizioni sintattiche

fattore = *variabile* | *costante-senza-segno* | *designatore-funzione* | *limite* | *insieme* | "(" *espressione* ")" | "**not**" *fattore*.
limite = *identificatore-limite*.

Come nel caso dei tipi array, uno schema di array conforme della forma

array [*a*1..*b*1 : *T*1; *a*2..*b*2 : *T*2;...; *an*..*bn* : *Tn*] **of** *T*

è strettamente equivalente a

```
array [a1..b1 : T1] of
    array [a2..b2 : T2] of
        ...
        array [an.. bn : Tn] of T
```

Notare che in un array conforme multidimensionale può essere impaccata solo la dimensione finale, e lo schema di array conforme va scritto nella forma

```
array [a1..b1 : T1...
                ...] of
  packed array [an..bn : Tn] of T
```

La restrizione che limita l'impaccamento ad una sola dimensione ostacola l'uso del meccanismo di array conforme per quel che riguarda alcuni array impaccati, ma permette di scrivere procedure e funzioni che manipolano stringhe di lunghezza variabile. Per esempio, la procedura seguente può essere usata per scrivere in forma codificata una stringa di lunghezza qualsiasi, usando la funzione di codifica di caratteri fornita come secondo parametro:

```
procedure Codifica (s : packed array [a..b : integer] of char;
                    function codice (c : char) : char);
var i : integer;
begin
   for i := a to b do
   write (codice (s [i]))
end
```

Sebbene i parametri attuali passati alla procedura *Codifica* possano essere tutti di tipo stringa, cioè

```
packed array [1..n] of char
```

il fatto che il limite inferiore del loro tipo indice sia sempre 1 non può essere rappresentato nella lista parametri formali. Se i limiti del tipo indice di un parametro array devono variare, allora vanno descritti entrambi in forma di variabile nella specifica di parametro array conforme. Nel caso di manipolazione di stringhe, ciò significa che parametri come *s* (vedi sopra) non vanno considerati variabili stringa entro la procedura, di modo che non si possono usare le operazioni speciali applicabili alle stringhe, come l'uso diretto nelle istruzioni *write,* l'assegnazione di stringhe letterali, od il loro confronto mediante gli operatori relazionali.

Quando si definiscono due o più parametri array conformi nella stessa sezione parametri, i parametri attuali corrispondenti, in qualsiasi chiamata della procedura o funzione, debbono essere dello stesso tipo; ciò assicura che i valori di limite denotati dagli identificatori di limite valgono per tutti i parametri attuali usati: quindi nella procedura o funzione gli array formali corrispondenti potranno essere considerati dello stesso tipo. In tal modo l'operazione di assegnazione del valore di uno di tali parametri ad un altro è valida. La procedura che segue massimizza i contenuti di due parametri array assegnando all'array la cui somma degli elementi è minore il valore dell'altro.

```
procedura Massimizza (var x,y: array [m..n: integer] of integer);
begin
   if somma (x) > somma (y)
     then y := x
     else x := y
end
```

Le assegnazioni

```
y := x e x := y
```

sono valide solo se i parametri array conformi x e *y* sono dichiarati nella stessa sezione parametri. I parametri attuali corrispondenti debbono essere dello stesso tipo.

Questi esempi illustrano l'insieme delle operazioni che possono essere applicate ai parametri array conformi:

(a) Si può far riferimento ai singoli componenti di un parametro array conforme usando la normale notazione delle variabili indiciate, come visto in *Codifica.*
(b) Un parametro array conforme può essere passato come parametro ad un'altra procedura o funzione che si aspetta un parametro array conforme analogo, così

come *x* e *y* sono passati a *somma* all'interno di *Massimizza.*

(c) Se due parametri array conformi sono dichiarati nella stessa sezione, debbono necessariamente essere dello stesso tipo, ed i loro valori possono, perciò, essere assegnati dall'uno all'altro, come in *Massimizza.*

Uno schema di array conforme si può usare per definire parametri sia *per valore* che *per variabile,* con le solite conseguenze sulle loro modalità d'uso. Il parametro attuale corrispondente ad un parametro formale array conforme per variabile deve essere una variabile array (o un altro parametro array conforme), e le operazioni sul parametro formale compiute nella procedura o funzione sono applicate direttamente a questo parametro attuale. La procedura *Massimizza,* che deve modificare l'uno o l'altro dei suoi parametri attuali, deve aver dichiarati *per variabile* i suoi parametri formali array conformi *x* e *y.*

Il parametro attuale corrispondente ad un parametro formale array conforme per valore può essere un'espressione che produce un valore di tipo array. (In pratica, le sole espressioni PASCAL che producono valori di array sono le variabili di tipo array, o le stringhe). Questo valore è assegnato al parametro formale quando viene chiamata la procedura o funzione; le operazioni successive sul parametro formale non alterano il parametro attuale. La procedura *Codifica* non deve alterare il suo parametro attuale array, e perciò il parametro formale *s* è dichiarato parametro array conforme *per valore.* Questo significa che *Codifica* può essere chiamata avendo come parametro una variabile stringa *v:*

Codifica (v,...)

o un parametro costante stringa

Codifica ('Mata Hari', ...)

Però, rispetto al primo esempio visto di parametro array conforme, nella funzione *somma,* questo schema è diverso. La funzione non deve alterare il suo parametro attuale, eppure il parametro formale *a* è dichiarato *per variabile.* La ragione di quest'apparente anomalia è la seguente. I parametri per valore sono implementati creando una copia dei parametri attuali, di solito all'interno della procedura o funzione chiamata. Certe implementazioni possono trovarsi in difficoltà nell'assegnare spazio all'interno di una procedura o funzione ad un parametro array conforme la cui ampiezza non sia predeterminata. Per questa ragione, la definizione del PASCAL permette esplicitamente che le copie di parametri array conformi siano fatte nel punto di chiamata, dove sono note le dimensioni del parametro formale. Per evitare tutti i problemi di copiatura si impone la restrizione addizionale che un parametro formale array conforme, o qualsiasi suo componente che sia conforme (cioè non di ampiezza prefissata), non può essere passato ad un'altra procedura o funzione come parametro attuale array conforme *per valore.* Se dunque la funzione *somma* fosse stata dichiarata con un parametro array conforme per valore, non avrebbe potuto essere utilizzata nella procedura *Massimizza* come visto sopra.

Sfortunatamente tale interdipendenza tra le liste dei parametri di procedure o funzioni che si chiamano l'un l'altra non è sempre evidente al momento della loro stesura, specie se debbono servire come funzioni di utilità generale (general-purpose routines). In tali casi la restrizione sopra descritta scoraggia l'uso di parametri array confor-

mi per valore, persino quando la procedura o funzione non deve modificare i parametri attuali che usa. D'altra parte, è pure inaccettabile l'uso dei soli parametri array conformi per variabile, nella stesura di procedure o funzioni che manipolano stringhe di caratteri, perché questo impedisce di invocarle con parametri costanti stringhe. Come esempio finale di uso del meccanismo del parametro array conforme, si consideri la generalizzazione della procedura di manipolazione di matrici vista nel par. 9.3. Possiamo ora estenderla per moltiplicare qualsiasi coppia di matrici rettangolari compatibili, ottenendo come risultato una corrispondente matrice rettangolare prodotto:

```
procedure moltiplicamatrici
        (A: array [m1A..n1A: integer; m2A..n2A: integer] of real;
         B: array [m1B..n1B : integer; m2B..n2B : integer] of real;
         var C: array [m1C..n1C : integer; m2C..n2C : integer] of real);
    { asserzione:
    m1A = m1C, m2B = m2C, m2A = m1B
    n1A = n1C, n2B = n2C, n2A = n1B }
var i,j,k : integer; somma : real;
begin
    for j := m1C to n1C do
      for k := m2C to n2C do
      begin
          somma := 0;
          for i := m2A to n2A do
            somma := somma + A [j, i] * B [i, k];
          C [j, k] := somma
      end
end
```

Come mostra il commento iniziale, la procedura assume che le matrici iniziali attuali abbiano per ciascuna dimensione uguali limiti superiore ed inferiore, e così il programmatore deve assicurarsi che i parametri attuali usati siano conformi a tale dichiarazione. In alternativa, si potrebbero scrivere, all'entrata della procedura, alcune istruzioni che controllino la validità di tali relazioni.

Tutti gli esempi mostrano che il meccanismo del parametro array conforme permette di definire procedure e funzioni assai generali, ma fanno anche vedere con quanta cura si debbano scrivere ed usare tali procedure, rispetto al caso in cui si usano parametri array 'fissi'. Quando abbiamo discusso i parametri array fissi nel paragrafo precedente, abbiamo detto che occorre tener conto dello spreco di memoria introdotto dalle operazioni di copia di grossi array che siano passati come parametri per valore. Tali considerazioni vanno applicate anche nel caso di parametri array conformi, ma in questo caso lo spreco può essere dovuto anche allo stesso meccanismo di parametro conforme. In molte implementazioni il tempo occorrente per fare accesso ad un elemento di un parametro array conforme può essere maggiore di quello occorrente per l'accesso ad un parametro array di tipo fisso, ed il tempo necessario per controllare che gli indici appartengano ad un intervallo corretto può essere anche maggiore. Occorre dunque per queste ragioni usare con parsimonia il meccanismo dell'array conforme. In generale va usato solo nelle situazioni in cui una procedura o funzione

deve gestire array di dimensioni variabili nel corso della stessa esecuzione del programma cui appartiene. In tutti gli altri casi l'uso del meccanismo normale di passaggio degli array è più semplice da capire, meno soggetto ad errori ed in parecchi casi più efficiente.

9.7. ALTRI TIPI STRUTTURATI

Il tipo array rappresenta solo uno dei modi in cui si possono definire dati composti, o strutturati. Nel Cap. 3 abbiamo definito *tipo* così:

tipo = *tipo-semplice* | *tipo strutturato* | *tipo-puntatore* | *identificatore-tipo.*

I *tipi-semplici* sono stati studiati nel Cap. 3, mentre i *tipi-puntatore* sono definiti nel Cap. 13. Ci sono poi in realtà quattro possibili forme di *tipo-strutturato.*

tipo-strutturato = [**"packed"**] *tipo-strutturato-non-impaccato*
tipo-strutturato-non-impaccato = *tipo-array* | *tipo-record* |
tipo-insieme | *tipo-file.*

I tipi array sono stati argomento di questo capitolo; i tipi record, insieme e file sono discussi nei Capp. 10, 11 e 12 rispettivamente.

Ogni tipo strutturato può avere il prefisso **packed** con effetto analogo, cioè l'implementazione dovrà cercare di minimizzare la quantità di memoria usata dalle variabili di quel tipo, anche a costo di un aumento del tempo di esecuzione del programma.

Le variabili di tipo strutturato possono essere manipolate come entità, oppure componente per componente, mediante gli opportuni costrutti di selezione.

Nel caso degli array la notazione delle variabili con indice permette di far riferimento e di manipolare i singoli componenti come se fossero variabili individuali. Nel Cap. 3 abbiamo definito *variabile* nel modo seguente:

variabile = *variabile-semplice* | *variabile-componente* | *variabile-puntata.*
variabile-semplice = *identificatore-variabile.*

Una *variabile-semplice* è soltanto un identificatore introdotto per denotare una variabile con una dichiarazione di variabile – può perciò denotare una variabile di tipo semplice (o puntatore), o una variabile di tipo strutturato, vista come entità. Una *variabile-componente* si usa per denotare un componente particolare di una variabile di tipo strutturato, e presenta una delle tre forme seguenti:

variabile-componente = *variabile-indiciata* | *designatore-campo* | *buffer-file.*

Come abbiamo visto, le *variabili-indiciate* sono usate per denotare i componenti di variabili di tipo array.

I *designatori-campo* ed i *buffer-file* denotano i componenti accessibili di una variabile di tipo record e tipo file rispettivamente. I componenti di una variabile di tipo insieme non sono accessibili individualmente come variabili, sebbene possano essere manipolati in altri modi, come si vedrà nel Cap. 11.

ESERCIZI

9.1 Estendere il programma di analisi di un testo (Es. 6.6), per determinare il numero di occorrenze di tutte le lettere da *A* a *Z* nel testo di input.

9.2 Scrivere un programma che legge una serie di 10 interi positivi, ne trova il massimo, stampa tale valore, il numero di volte che viene incontrato, e le posizioni in cui appare. Il processo va ripetuto col valore successivo in ordine di grandezza, e così via.

Esempio:
input : 7 10 143 10 52 143 72 10 143 7
output : 143 *C' E'* 3 *VOLTE, NELLE POSIZIONI* 3 6 9
...
7 *C' E'* 2 *VOLTE, NELLE POSIZIONI* 1 10

9.3 Scrivere un programma che legge un testo in input e stampa il numero delle occorrenze in cui appare nel testo ciascuna coppia di lettere adiacenti.

9.4 Scrivere un programma che legge una sequenza di coppie di parole e stampa ciascuna coppia in ordine alfabetico. Si possono fare le ipotesi che ciascuna coppia di parole sia da sola su una linea di input, che le parole di ciascuna coppia siano separate da almeno uno spazio, e che ogni parola non sia più lunga di 16 lettere. Usare un formato analogo per l'output.

9.5 Scrivere una procedura

procedure *grafico* (**function** *f (x : real) : real*; *xinferiore, xsuperiore : real)*

che "disegna" un grafico della funzione *f (x)* nell'intervallo *xinferiore-xsuperiore,* con l'asse *x* in ascissa. Il grafico dovrebbe essere in scala, e usare 100 posizioni in ascissa e 50 in ordinata; alla fine vanno stampati entrambi gli assi.

Usare questa procedura per disegnare il grafico di *sin (x)* nell'intervallo da 0 a 2π.

10.
Record

10.1. LA NOZIONE DI RECORD

Un array è formato da componenti omogenei, che risultano dello stesso tipo. Capita spesso, invece, che un dato sia costituito da un certo numero di componenti non omogenei, che possono essere di tipo diverso. Nel Programma 4 abbiamo introdotto tre variabili per indicare la data:

```
giorno : 1..31;
mese : 1..12;
anno : 1900..2000;
```

assumendo implicitamente che le tre variabili rappresentavano le diverse componenti della stessa data.

Una data si può perciò considerare formata proprio da tre valori dei tre diversi tipi specificati. Il PASCAL consente di definire un tipo di dato i cui valori sono una composizione di altri e diversi tipi di dato – il *tipo-record.* Per le date possiamo definire il tipo record (1). Questa definizione specifica che un valore di tipo *data* è formato da tre valori componenti, di tipo 1..12, 1..31 e 1900..2000, rispettivamente. Gli identificatori *g, m* ed *a,* introdotti nella definizione del tipo record, sono i nomi attribuiti ai componenti individuali, o campi, e permettono di fare riferimento ai valori che compongono una variabile di tipo *data.*

```
data = record
         g : 1..31;
         m : 1..12;                                    (1)
         a : 1900..2000
       end;
```

La forma generale di un tipo record, in PASCAL, è definita sintatticamente così

tipo-record = " **record** " *lista-campi* " **end** ".
lista-campi = [*(parte-fissa* [";" *parte-variante*] | *parte-variante)* [";"]]

Per il momento ignoriamo i tipi record contenenti una *parte-variante,* che verranno discussi in un secondo momento nel corso di questo capitolo.

parte-fissa = *elemento-record* { " ; " *elemento-record* }
elemento-record = *lista-identificatori* " : " *tipo.*

Per ora, dunque, un *tipo-record* è costituito dai simboli **record** ed **end** che racchiudono una *parte-fissa,* che è una lista di uno o più *elementi-record* (separati da punto e virgola). Ogni *elemento-record* presenta una lista di uno o più identificatori (separati da virgole) e la definizione del loro tipo, oppure può essere vuoto. Questi identificatori sono detti *identificatori-campo,* e di qui in avanti la categoria sintattica

identificatore-campo = *identificatore.*

viene usata per denotare la classe degli identificatori così introdotti. Gli esempi riportati in (2) illustrano ulteriormente la definizione dei tipi-record.

```
cartesiano = record
                x,y : real
             end;
seme = (fiori, quadri, cuori, picche);
valore = (due, tre, quattro, cinque, sei, sette, otto, nove, dieci,
          fante, regina, re, asso);                                  (2)
carta = record
            s : seme;
            v : valore
        end;
ora = record
          ore : 0..24;
          minuti, secondi : 0..59
      end;
```

Il campo d'azione degli identificatori di campo introdotti nella definizione di un tipo-record è il tipo record stesso, e così gli identificatori di campo devono essere distinti nell'ambito dello stesso tipo-record, ma non necessariamente distinti dagli altri identificatori dichiarati al di fuori del tipo-record.

Dopo aver definito un tipo-record, ed avergli attribuito un nome per mezzo di una definizione come quelle sopraindicate, nella parte-dichiarazione-variabili si possono dichiarare nel solito modo variabili dei tipi cui è stato attribuito un nome; ad esempio:

decollo, atterraggio : *ora*; *fuoco*1, *fuoco*2 : *cartesiano*; *giocata* : *carta*;

Naturalmente un tipo record si può definire nella stessa dichiarazione di variabili, come in (3), ma in tal caso non si possono dichiarare in altri punti del programma variabili dello stesso tipo. L'introduzione degli identificatori di tipo per denotare tipi condivisi è spesso necessaria, ed in ogni caso è buona regola di programmazione.

```
giorno1, giorno2 : record
                       g : 1..31;
                       m : 1..12;                                    (3)
                       a : 1900..2000
                   end
```

I dati mantenuti in un programma sotto forma di variabili di tipo record si devono poter in qualche modo manipolare. L'operazione più semplice che si può applicare ad

una variabile di tipo record è l'assegnazione ad essa del valore di una variabile dello stesso tipo record; ad esempio

decollo := *atterraggio*

che ha l'effetto di assegnare il valore di ciascun campo della variabile *atterraggio* al corrispondente campo della variabile *decollo.*

Per far riferimento ad uno dei campi componenti il record si usa un *designatore-campo,* che permette di specificare gli elementi di una variabile di tipo record mediante i rispettivi identificatori di campo.

designatore-campo = *variabile-record* " . " *identificatore-campo.*
variabile-record = *variabile.*

Così, per la variabile *giorno*1 di tipo *data* già vista

*giorno*1.*g* denota la componente giorno di tipo 1..31;
*giorno*1.*m* denota la componente mese di tipo 1..12;
*giorno*1.*a* denota la componente anno di tipo 1900..2000.

Seguono ulteriori esempi di designatori di campo, che presuppongono le dichiarazioni di variabili fornite in precedenza:

decollo.ore
giocata.s
*fuoco*1.*x*

Il componente di una variabile di tipo record si può usare ovunque è consentito l'uso di una variabile del tipo di quel componente, per esempio

```
xx := (fuoco1.x + fuoco2.x) / 2;
if giocata.s = cuori then...
write (giorno1.m, giorno1.g, giorno1.a)
```

Si può così cambiare il valore di un solo componente del record, lasciando invariati i valori di tutti gli altri componenti, ponendo il nome del componente richiesto in una istruzione di assegnazione; per esempio

```
fuoco1.x := 0;
giorno1.a := 1949;
decollo.ore := decollo.ore + 1
```

Questa operazione prende il nome di *aggiornamento selettivo*. L'assegnazione di variabile di tipo record

*giorno*1 := *giorno*2

in cui *giorno*1 e *giorno*2 sono variabili di tipo *data,* equivale perciò alle tre assegnazioni di aggiornamento selettivo

```
giorno1.g := giorno2.g;
giorno1.m := giorno2.m;
giorno1.a := giorno2.a
```

In tale caso l'assegnazione alla variabile-record è da preferirsi, perché più chiara, più sintetica, ed in molte implementazioni più efficiente.

L'uso dei designatori di campo è illustrato dalla procedura (4), che determina il punto medio *c* del segmento congiungente due punti *p*1 e *p*2; tutti i punti sono rappresentati come record di tipo *cartesiano.*

```
procedure puntomedio (p1, p2 : cartesiano; var c : cartesiano);
begin
  c.x := (p1.x + p2.x) /2;                                    (4)
  c.y := (p1.y + p2.y) /2
end
```

10.2. L'ISTRUZIONE-WITH

Quando si opera su una variabile di tipo record, è piuttosto usuale fare parecchie assegnazioni alle sue componenti in una piccola zona del programma. Ad esempio, per inizializzare la variabile *giorno*1 si potrebbe scrivere

*giorno*1.*g* := 1; *giorno*1.*m* := 1; *giorno*1.*a* := 1980

In queste occasioni i ripetuti riferimenti ai campi di un record che richiedono la scrittura del nome della variabile di tipo record, con la specifica dell'identificatore di campo, diventano presto noiosi, specialmente se si usano identificatori lunghi. Il PASCAL fornisce allora un'istruzione, da usare con variabili di tipo record, che permette di far riferimento alle componenti del record senza dover ripetere per ogni riferimento l'identificatore della variabile di tipo record. Questa è l'*istruzione-with,* definita come segue:

istruzione-with = "**with**" *variabile-record* { "," *variabile-record* } "**do**" *istruzione.*

All'interno di una istruzione controllata da un'istruzione-with uno qualsiasi dei campi della variabile di tipo record indicata può essere denotato usando soltanto l'identificatore di campo. L'effetto di un'istruzione-with è di aprire un nuovo campo d'azione, che contiene i corrispondenti identificatori di campo per ciascuna delle variabili di tipo record in essa nominate, consentendo così l'uso degli identificatori di campo come variabili. L'inizializzazione della variabile *giorno*1 si può perciò scrivere:

```
with giorno1 do
begin
  g := 1;
  m := 1;
  a := 1980
end
```

La forma generale dell'*istruzione-with*

with *v*1, *v*2, ..., *vn* **do** *S*

è equivalente a (5), cioè i campi d'azione vengono aperti, e perciò annidati, nell'ordine in cui sono elencate le variabili nell'istruzione-with.

```
with v1 do
  with v2 do                                    (5)
    :
    :
    with vn do S
```

In tal modo se le variabili di tipo record *v1* e *v2* hanno un campo identificato da *F,* una semplice occorrenza di *F* all'interno dell'istruzione *S* denota il corrispondente campo di *v2,* non quello di *v1*, per via delle regole dei campi d'azione annidati. Il campo *F* di *v1* si può denotare all'interno di *S* scrivendo esplicitamente *v1.F.*

L'uso dell'istruzione-with non solo permette di ridurre la lunghezza del testo di un programma, ma aumenta anche la sua leggibilità, ed in alcuni casi produce anche un programma più efficiente.

L'uso dei record e dell'istruzione-with è illustrato nella riformulazione del Programma 4 presentata in (6). Le date sono ora rappresentate come variabili del tipo record *data,* e viene usata una procedura di nome *aggiorna* che aggiorna il suo parametro variabile di tipo *data* col valore del giorno seguente la data originale. Il programma principale legge una sequenza di date e, per ciascuna di esse, produce in output la data del giorno successivo.

```
program giornosuccessivo (input, output);
type data = record
              giorno : 1..31;
              mese : 1..12;
              anno : 1900..2000
            end;
var giorno : data;
    procedure scrividata (d : data);
    begin
        with d do
            write (giorno : 2, '/', mese : 3, '/', anno : 4)
    end;
    procedure aggiorna (var d : data);
    var giornidelmese : 28..31;
    begin
      with d do
      begin
        case mese of
          1,3,5,7,8,10,12 : giornidelmese := 31;
          4,6,9,11 : giornidelmese := 30;
          2 : if (anno mod 4 = 0) and (anno <> 1900)          (6)
                then giornidelmese := 29
                else giornidelmese := 28
        end;
        if giorno = giornidelmese
          then begin
                 giorno := 1;
```

```
                    if mese = 12
                      then begin
                             mese := 1;
                             anno := anno + 1
                           end
                      else mese := mese + 1
                end
           else giorno := giorno + 1
      end
end;
begin
  while not eof (input) do
  begin
    with giorno do readln (giorno, mese, anno);
    write ('il giorno seguente');
    scrividata (giorno);
    aggiorna (giorno);
    write (' e''');
    scrividata (giorno);
    writeln
  end
end.
```

10.3. STRUTTURE MISTE

La definizione di un *elemento-record* indica che i campi di un record possono essere di tipo qualsiasi – non solo di tipo semplice, come negli esempi considerati finora, ma anche di un tipo strutturato, come un array, o un altro tipo record. Ad esempio, potremmo definire un tipo record per descrivere i dati anagrafici di una persona come segue:

```
persona = record
            nome : packed array [1..20] of char;
            datadinascita: data
          end
```

Un valore di tipo *persona* è formato da due valori componenti – un campo *nome,* che è una stringa di 20 caratteri, ed un campo *datadinascita,* che è a sua volta un record composto da tre valori, *g, m* ed *a.*

Analogamente, un record può essere un componente di un altro tipo strutturato. Perciò si può definire un tipo array i cui elementi sono record. Per esempio, possiamo definire il tipo

```
manodibridge = array [1..13] of carta
```

nel quale il tipo record *carta* è già stato definito in precedenza.

In questa maniera si possono descrivere strutture di dati di arbitraria complessità in termini di strutture più semplici precedentemente definite, ed i loro componenti si

possono manipolare a vari livelli, usando la notazione già introdotta.

Ad esempio, nel caso di una dichiarazione

p : persona

si ha

p	denota una variabile di tipo *persona* che potrebbe essere assegnata ad altre variabili dello stesso tipo;
p.nome	denota una stringa di 20 caratteri che potrebbe essere assegnata, confrontata o messa in output come tutte le altre variabili di tipo stringa;
p.nome [*i*]	denota l'iesimo carattere di *p.nome,* che potrebbe essere usato in qualsiasi modo ammissibile per una variabile di tipo carattere;
p.datadinascita	denota una variabile del tipo record *data* che potrebbe essere assegnato a o da altre variabili di tipo *data*;
p.datadinascita.d	denota una variabile di tipo 1..31

e così via.

Perciò l'istruzione

writeln (p.nome, p.datadinascita.a)

stampa il nome e l'anno di nascita delle persona descritta dalla variabile *p*. L'istruzione

with *p* **do** *writeln (nome, datadinascita.a)*

ha esattamente lo stesso effetto.

10.4. RECORD IMPACCATI

Nei capitoli precedenti è stato descritto l'uso di array impaccati. Anche le variabili di tipo record si possono dichiarare impaccate facendo precedere, nella definizione di tipo-record, il simbolo **record** dal simbolo **packed.** I vantaggi che offre l'impaccamento dei record sono gli stessi che per gli array, cioè la riduzione dello spazio di memoria occupato, al prezzo di un aumento del tempo di accesso nella selezione dei campi. Per questo val la pena di usare l'impaccamento solo se la principale operazione associata ad un tipo record è la copia del valore complessivo del record e non la selezione dei singoli campi.

Non è consentito passare un componente di un record impaccato come parametro variabile nella chiamata di una procedura o funzione.

PROGRAMMA 10 (Aggiornamento della classifica del campionato)

Sei squadre partecipano ad un campionato di calcio, giocando ognuna due volte con tutte le altre. Si richiede un programma il cui input sia una classifica del campionato, formata da una linea per ogni squadra. Una linea contiene i seguenti dati:

il nome della squadra, di 12 caratteri;
il numero delle partite giocate, vinte, perse e pareggiate finora, rappresentate da quattro numeri interi, ciascuno seguito da almeno uno spazio;
il numero dei punti conquistati dalla squadra, sapendo che una vittoria vale due punti, un pareggio uno, e una sconfitta zero.

L'input della classifica è seguito dai risultati dell'ultimo turno di partite, presentati in modo che ciascun risultato sia in una linea separata contenente

il nome della squadra di casa,
il punteggio della squadra di casa,
il nome della squadra in trasferta,
il punteggio della squadra in trasferta,

ed i cui elementi sono separati da almeno uno spazio. Il programma deve leggere la classifica ed i risultati, e fornire in output una classifica aggiornata, in cui le squadre con i dati relativi siano elencate nell'ordine corrispondente alla loro nuova posizione in classifica.

```
begin
  leggiclassifica;
  aggiornaclassifica;                                    (7)
  stampaclassifica
end
```

La struttura base del programma si può esprimere come indicato in (7). Tutti e tre questi processi manipolano la classifica, perciò dobbiamo innanzitutto definire una adeguata struttura per rappresentarla all'interno del programma. Poiché ad ogni squadra in classifica è associato un insieme di dati, una struttura adatta potrebbe essere un array

classifica : **array** [*posizionesquadra*] **of** *datisquadra*

in cui *posizionesquadra* è così definito

posizionesquadra = 1..*numerosquadre*

I dati associati ad ogni squadra sono il nome della squadra, il numero delle partite giocate, vinte, perse e pareggiate, ed i punti finora ottenuti. Per cui il tipo *datisquadra* si può definire come il tipo record con sei campi riportato in (8).

```
datisquadra = record
                nome : nomesquadra;
                giocate, vinte, perse, pari : 0..maxnumpartite;    (8)
                punti : 0..maxnumpunti
              end
```

in cui

nomesquadra = **packed array** [1..12] **of** *char*

Con questa struttura dati siamo in grado di passare al progetto dei tre processi principali del programma.

Il processo *leggiclassifica* deve semplicemente realizzare la lettura delle prime linee di input, contenenti la classifica corrente, ed immagazzinare i dati relativi a ciascuna squadra nel corrispondente elemento dell'array *classifica,* come mostrato in (9).

```
procedure leggiclassifica;
begin
  for ogni squadra do                                          (9)
  begin
    leggi il nome e i dati ponendoli nel successivo elemento della classifica
  end
end
```

Questo processo si può già facilmente programmare nei dettagli, ma siccome i nomi delle squadre devono esser letti anche nelle linee contenenti i risultati, è utile introdurre una procedura condivisa per questa operazione, avente la forma

```
procedura legginome (var N : nomesquadra);
```

Il processo *aggiornaclassifica* realizza la lettura degli ultimi risultati e, per ogni partita, aggiorna la classifica delle squadre coinvolte a seconda del risultato. Ricordando che una vittoria vale 2 punti, un pareggio 1 punto ed una sconfitta 0 punti, la procedura *aggiornaclassifica* si può scrivere informalmente come in (10).

```
procedure (aggiornaclassifica);
begin
  for ogni partita do
  begin
    leggi le squadre ed i punteggi;
    somma 1 alle partite giocate di entrambe le squadre;
    case risultato of
      vittoriaincasa : begin
                       somma 1 alle partite vinte e 2 ai punti della squadra di casa;
                       somma 1 alle partite perse della squadra in trasferta
                       end;                                    (10)
      vittoriaintrasferta : begin
                            somma 1 alle partite perse dalla squadra di casa;
                            somma 1 alle partite vinte e 2 ai punti della squadra
                              in trasferta
                            end;
      pareggio : begin
                 somma 1 alle partite pari ed ai punti di entrambe le squadre
                 end
    end
  end
end
```

Per ogni squadra che compare in una linea di risultati è necessario individuare il corrispondente record nella classifica. Si tratta di una semplice operazione di ricerca, che si può isolare in una funzione della forma

```
function posizione (nome : nomesquadra) : posizionesquadra;
```

Utilizzando questa funzione il processo di aggiornamento si può programmare con facilità.

Il processo *stampaclassifica* si divide naturalmente in due parti, riportate in (11).

```
begin
  ordina la classifica in base ai punti;                    (11)
  stampa la classifica ordinata
end
```

L'ordinamento della classifica in base ai punti delle squadre è descritto in (12).

```
for i := 1 to numerosquadre - 1 do
begin
  trova la squadra con il massimo numero di punti
    dalla posizione i in avanti;
  scambia i dati relativi alla squadra                      (12)
    trovata con quella in posizione i se necessario
end
```

Per trovare la squadra con il massimo numero di punti introduciamo le variabili di lavoro *maxpunti* e *maxposizione,* per registrare il massimo numero di punti individuato finora e la sua posizione (13).

```
maxposizione := i;
maxpunti := classifica [i].punti;
for j := i + 1 to numerosquadre do
  if classifica [j].punti > maxpunti
    then begin                                              (13)
           maxposizione := j;
           maxpunti := classifica [j].punti
         end
```

Lo scambio dei dati relativi alle squadre è riportato nella sequenza (14).

```
if maxposizione <> i
  then begin
         salvadati := classifica [i];
         classifica [i] := classifica [maxposizione];
         classifica [maxposizione] := salvadati             (14)
       end
```

L'output della classifica ordinata ed aggiornata richiede la stampa dei dati relativi ad ogni record dell'array *classifica* ordinato.

Il Listato 10 mostra il programma finale e l'output prodotto dall'input corrispondente alla classifica iniziale e ai risultati riportati nella Tabella 10.1.

	P	V	N	P	Pts
Red Rovers	3	2	1	0	5
Silver City	3	1	2	0	4
Blue United	3	1	1	1	3
Black Forest	3	1	1	1	3
Green Villa	3	1	0	2	2
Brown Town	3	0	1	2	1

Black Forest	3	Brown Town	4
Green Villa	2	Red Rovers	2
Blue United	1	Silver City	0

Tabella 10.1 Classifica e risultati

LISTATO 10

```
PROGRAM AGGIORNACLASSIFICA(INPUT,OUTPUT);


CONST NUMEROSQUADRE    =  6;
      MAXNUMPARTITE    = 10;
      MAXNUMPUNTI      = 20;


TYPE  POSIZIONESQUADRA = 1..NUMEROSQUADRE;
      NOMESQUADRA      = PACKED ARRAY [1..12] OF CHAR;
      DATISQUADRA      = RECORD
                           NOME : NOMESQUADRA;
                           GIOCATE,VINTE,
                           PARI,PERSE    : 0..MAXNUMPARTITE;
                           PUNTI : 0..MAXNUMPUNTI
                         END;
VAR   CLASSIFICA : ARRAY [POSIZIONESQUADRA] OF DATISQUADRA;


      PROCEDURE LEGGINOME(VAR N : NOMESQUADRA);
      VAR  CH : CHAR;
           I  : 2..12;
      BEGIN
         REPEAT
            READ(CH)
         UNTIL CH <> ' ';
         N[1]:= CH;
         FOR I:= 1 TO 12 DO
           READ (N[I])
      END; (* LEGGINOME *)

      PROCEDURE LEGGICLASSIFICA;
      VAR  I : POSIZIONESQUADRA;
      BEGIN
         FOR I:= 1 TO NUMEROSQUADRE DO
            WITH CLASSIFICA[I] DO
```

```
      BEGIN
         LEGGINOME(NOME);
         READLN(GIOCATE,VINTE,PARI,PERSE,PUNTI)
      END
END; (* LEGGICLASSIFICA *)


PROCEDURE AGGIORNACLASSIFICA;
VAR  NOME : NOMESQUADRA;
     SQUADRA1,SQUADRA2 : POSIZIONESQUADRA;
     RETI1,RETI2 : 0..MAXINT;

     FUNCTION POSIZIONE(NOMECERCATO : NOMESQUADRA) :
                                        POSIZIONESQUADRA;
     VAR  I: POSIZIONESQUADRA;
     BEGIN
        I:= 1;
        WHILE CLASSIFICA[I].NOME <> NOMECERCATO DO
        I:= I + 1;
        POSIZIONE:= 1
     END; (* POSIZIONE *)


BEGIN (* AGGIORNA CLASSIFICA *)
   REPEAT
   LEGGINOME(NOME);
   SQUADRA1:= POSIZIONE(NOME);
   READ(RETI1);
   LEGGINOME(NOME);
   SQUADRA2:= POSIZIONE(NOME);
   READLN(RETI2);
   WITH CLASSIFICA[SQUADRA1] DO
      GIOCATE:= GIOCATE + 1;
   WITH CLASSIFICA[SQUADRA2] DO
      GIOCATE:= GIOCATE + 2;
   IF RETI1 > RETI2
      THEN BEGIN (* VITTORIA IN CASA *)
              WITH CLASSIFICA[SQUADRA1] DO
                 BEGIN
                    VINTE:= VINTE + 1;
                    PUNTI:= PUNTI + 2
                 END;
              WITH CLASSIFICA[SQUADRA2] DO
                 PERSE:= PERSE + 1
           END
      ELSE IF RETI2 > RETI1
              THEN BEGIN (* VITTORIA IN TRASFERTA *)
                      WITH CLASSIFICA[SQUADRA2] DO
                         BEGIN
                            VINTE:= VINTE + 1;
                            PUNTI:= PUNTI + 2
                         END;
                      WITH CLASSIFICA[SQUADRA1] DO
                         PERSE:= PERSE + 1
                   END
```

```
            ELSE BEGIN (* PAREGGIO *)
                    WITH CLASSIFICA[SQUADRA1] DO
                       BEGIN
                          PARI:= PARI + 1;
                          PUNTI:= PUNTI + 1
                       END;
                    WITH CLASSIFICA[SQUADRA2] DO
                       BEGIN
                          PARI:= PARI + 1;
                          PUNTI:= PUNTI + 1
                       END
                 END
   UNTIL EOF(INPUT)
END; (* AGGIORNACLASSIFICA *)


PROCEDURE STAMPACLASSIFICA;
VAR  I, J, MAXPOSIZIONE : POSIZIONESQUADRA;
     MAXPUNTI : 0..MAXNUMPUNTI;
     SALVADATI : DATISQUADRA;
BEGIN
   (* ORDINA CLASSIFICA IN ORDINE DI PUNTI *)
   FOR I:= 1 TO NUMEROSQUADRE - 1 DO
   BEGIN
      (* TROVA SQUADRA COL MAX NUMERO DI PUNTI
         DALLA POSIZIONE I IN AVANTI          *)
      MAXPUNTI:= CLASSIFICA[I].PUNTI;
      MAXPOSIZIONE:= I;
      FOR J:= I + 1 TO NUMEROSQUADRE DO
         IF CLASSIFICA[J].PUNTI > MAXPUNTI
            THEN BEGIN
                    MAXPUNTI:= CLASSIFICA[J].PUNTI;
                    MAXPOSIZIONE:= J
                 END;
      (* SE NECESSARIO SCAMBIA CON LA POSIZIONE I *)
      IF MAXPOSIZIONE <> I
         THEN BEGIN
                 SALVADATI:= CLASSIFICA[I];
                 CLASSIFICA[I]:= CLASSIFICA[MAXPOSIZIONE];
                 CLASSIFICA[MAXPOSIZIONE]:= SALVADATI
              END
    END;
    (* STAMPA CLASSIFICA ORDINATA *)
    WRITELN ('P':16,'V':4,'N':4,'P':4,'   PTI');
    FOR I:= 1 TO NUMEROSQUADRE DO
       WITH CLASSIFICA[I] DO
          WRITELN(NOME, GIOCATE:4, VINTE:4, PARI:4,
                  PERSE:4, PUNTI:6)
 END; (* STAMPACLASSIFICA *)


BEGIN (* PROGRAMMA PRINCIPALE *)
   LEGGICLASSIFICA;
   AGGIORNACLASSIFICA;
```

```
   STAMPACLASSIFICA
END.
```

```
                P   V   N   P   PTI
RED ROVERS      4   2   2   0     6
BLUE UNITED     4   2   1   1     5
SILVER CITY     4   1   2   1     4
BLACK FOREST    4   1   1   2     3
GREEN VILLA     4   1   1   2     3
BROWN TOWN      4   1   1   2     3
```

10.5. RECORD VARIANTI

La natura ed il tipo di alcune componenti di un dato talvolta dipendono dai valori di altre componenti. Ad esempio, si consideri la definizione di tipo record (15), che descrive l'informazione contenuta in un registro di tutte le persone di uno stato in un dato momento. In essa si assume che i tipi *nomepersona* e *data* siano definiti altrove. Il campo *origine* distingue tra i cittadini e gli stranieri temporaneamente in visita nello stato.

```
type persona = record
                   nome : nomepersona;
                   datadinascita : data;                    (15)
                   origine : (nazionale, straniero)
               end
```

Supponiamo di voler estendere questa definizione di tipo per fornire altre informazioni sulle persone. Nel caso di cittadini dobbiamo registrare il luogo di nascita, mentre per gli stranieri si richiede lo stato d'origine, la data ed il porto d'ingresso nello stato.

Il tipo record *persona* ha pertanto due strutture alternative, o *varianti,* a seconda che l'origine di una persona sia nazionale o straniera. In entrambi i casi il nome e la data di nascita sono comunque richiesti, ma abbiamo un diverso numero e tipo degli altri componenti del record, a seconda del valore del campo *origine.*

In PASCAL si possono usare record con strutture alternative grazie all'introduzione nella definizione del tipo record della cosiddetta *parte-variante.* Come annunciato in precedenza, la *parte-variante* di un record segue la *parte-fissa* (se questa esiste), cioè la dichiarazione dei campi in comune a tutte le strutture alternative precede la dichiarazione di queste alternative.

La *parte-variante* è formata da un *campo-etichetta,* di un tipo precedentemente definito i cui valori permettono di distinguere le possibili strutture alternative o varianti, ed è seguita dalle ulteriori *liste-campi* corrispondenti a ciascuna di queste strutture alternative. Ogni *lista-campi* è etichettata con il valore del *campo-etichetta* corrispondente.

parte-variante = "**case**" *campo-etichetta tipo-identificatore* "**of**" *variante*
{ ";" *variante* }.
campo-etichetta = [*identificatore* ":"].
variante = *lista-etichette-case* ":" "(" *lista-campi* ")".

Il *campo-etichetta* deve essere di un tipo ordinale, ogni valore del quale deve apparire una sola volta come etichetta case nelle varianti che seguono. Così, il tipo *persona* potrebbe essere esteso come mostrato in (16). Notare che il tipo *generepersona* va introdotto prima, poiché il campo-etichetta può essere specificato soltanto come identificatore di tipo.

```
type generepersona = (nazionale, straniero);
     persona = record
                 nome : nomepersona;
                 datadinascita : data;
                 case origine : generepersona of
                   nazionale : (luogonascita : nomeluogo);
                   straniero : (statoorigine : nomeluogo;
                                datadiingresso : data;
                                portodiingresso : nomeluogo)
                 end
```
(16)

Nella definizione di un record variante, come la precedente, il campo-etichetta è un campo esplicito del record, che può essere selezionato ed aggiornato come un qualunque altro campo. Quindi, se *questapersona* è una variabile di tipo *persona,* si può scrivere l'istruzione

```
questapersona.origine := nazionale
```

oppure

```
if questapersona.origine <> straniero
  then writeln (questapersona.nome)
```

Ad ogni modo, la selezione di questi campi è valida soltanto quando il *campo-etichetta origine* ha davvero il valore presunto *(straniero* nel primo caso, *nazionale* nel secondo). Per ragioni di efficienza, sono poche le implementazioni del PASCAL che realmente controllano il valore del campo-etichetta quando viene selezionato il campo di una variante. Diventa quindi responsabilità del programmatore assicurare che ai campi delle varianti vengano fatti accessi corretti: sbagliare nel far questo porta generalmente a programmi scorretti.

Sfortunatamente il PASCAL non offre nessuno strumento per aiutare il programmatore a questo riguardo. Tuttavia, l'uso sistematico dell'istruzione-case in corrispondenza con un'istruzione-with permette di diminuire la probabilità di errore. Ad esempio, se scriviamo la sequenza mostrata in (17), appare evidente che al campo *luogonascita* si può far riferimento solo dall'interno del corpo case etichettato *nazionale,* mentre ai campi *statoorigine, portodiingresso* e *datadiingresso* si può accedere soltanto dal corpo case etichettato *straniero.*

```
with questapersona do
begin
  nome := ...;
  datadinascita := ...;
  case origine of
    nazionale : luogonascita := ...;
```
(17)

```
        straniero : begin
                        statoorigine := ...;
                        datadiingresso := ...;
                        portodiingresso := ...;
                    end
    end;
    :
end
```

In alcuni casi può non esser richiesto alcun campo per una variante *v* di un tipo-record, nel qual caso la variante si definisce come una lista vuota, cioè

```
v : ( )
```

Per esempio, se non avessimo desiderato nessuna informazione supplementare sugli stranieri, avremmo potuto definire il tipo record *persona* come indicato in (18).

```
persona = record
              nome : nomepersona;
              datadinascita : data;
              case origine : generepersona of
                nazionale : (luogonascita : nomeluogo);
                straniero : ( )
          end
```
(18)

Una struttura record può non avere la parte-fissa (cioè nessun campo comune a tutte le sue strutture alternative) e consistere semplicemente di una parte-variante. Per esempio, giocando a poker con i jolly, qualsiasi carta presa in considerazione potrebbe essere una delle 52 carte normali oppure un jolly. Per descrivere una carta per il poker potremmo usare i tipi specificati in (19).

```
seme = (fiori, quadri, cuori, picche);
valore = (due, tre, quattro, cinque, sei, sette, otto, nove,
          dieci, fante, regina, re, asso);
jolly = (jollyrosso, jollynero);
tipocarta = (normale, matta);
cartadapoker = record
                   case quale : tipocarta of
                     normale : (s : seme; v : valore);
                     matta : (j : jolly)
               end
```
(19)

Negli esempi sin qui visti il *campo-etichetta* è una componente del tipo record variante cui si può assegnare un valore, e l'elaborazione di un record prevede il controllo di questa etichetta per determinare la variante che interessa. La sintassi del *campo-etichetta* consente, però, l'omissione dell'identificatore del *campo-etichetta,* nel qual caso il programmatore non ha alcun modo per assegnare ad essa un valore o verificarlo – la variante corrente per un record è implicata dai campi cui si è fatto accesso. Un tipo-record siffatto potrebbe essere utile nei programmi in cui la parte variante di un re-

cord è sempre definita dal contesto. Potremmo considerare, ad esempio, le pagine di un libro definite come in (20).

```
tipopagina = (prima,successiva);
pagina = record
           case tipopagina of
             prima : (titolo : nomelibro;                    (20)
                      autore : nomepersona;
                      ISBN : packed array [1..10] of char);
             successiva : (testo : array [1..dimensionepagina]
                             of linee)
         end
```

Il contesto impone che solo i record che rappresentano la prima pagina di un libro assumano la prima variante; tutti gli altri la seconda. Se il programma è sempre al corrente di quali record rappresentano la prima pagina (per esempio, mediante l'ordine in cui vengono elaborati i record) non sono necessarie assegnazioni al campo-etichetta.

La definizione sintattica della *lista-campi* specifica che la *parte-variante* deve seguire la *parte-fissa* (se esiste). Poiché la sintassi della *parte-variante* contiene una *lista-campi*, risulta possibile che le parti varianti di un record contengano a loro volta *parti-varianti* annidate. Questo è illustrato dall'estensione della definizione del tipo record *persona*, presentata in (21), per distinguere tra i cittadini di uno stato nati nello stato stesso e quelli nati altrove, e poi naturalizzati.

```
type generepersona = (nazionale, straniero);
     condizione = (pernascita, naturalizzato);
     persona = record
                 nome : nomepersona;
                 datadinascita : data;
                 case origine : generepersona of
                   nazionale : (luogonascita : nomeluogo;        (21)
                                case qualifica : condizione of
                                  pernascita : ( );
                                  naturalizzato : (numero : integer;
                                                   datanaturalizzazione : data));
                   straniero : (statoorigine : nomeluogo;
                                datadiingresso : data;
                                portodiingresso : nomeluogo)
               end
```

I campi compresi nella parte-variante più interna vengono selezionati come qualsiasi altro campo del record; per esempio, si può scrivere

```
if questapersona.qualifica = naturalizzato
  then writeln (questapersona.numero)
```

oppure

```
questapersona.datanaturalizzazione.a := 1980
```

In quest'ultimo caso si assume che i campi etichetta *origine* e *qualifica* abbiano i valori *nazionale* e *naturalizzato* rispettivamente.

I record varianti sono uno strumento adeguato per esprimere i dati in molte applicazioni pratiche. Tuttavia, come già osservato, manipolarli nei programmi richiede molta attenzione, specialmente quando vengono modificate le parti varianti di un record. In un record variante che ha l'identificatore del campo etichetta, la parte variante cambia quando viene assegnato un nuovo valore al campo etichetta. Nel caso invece di record variante senza identificatore del campo etichetta, un cambiamento della parte variante è provocato dal riferimento ad un campo relativo ad un'altra variante. In entrambi i casi, i campi varianti della vecchia variante cessano di esistere, ed i campi della nuova variante hanno valori indefiniti. La manipolazione dei campi deve essere consistente con queste operazioni di creazione e distruzione che avvengono ad ogni cambiamento di una variante. Per impedire situazioni inconsistenti, che potrebbero essere causate da cambiamenti delle parti varianti, il PASCAL impone due ulteriori regole:

(a) Un cambiamento di variante non può avvenire quando un campo variante è usato come parametro attuale per variabile o come variabile di tipo record in un'istruzione-with.
(b) Un campo etichetta non può essere passato ad una procedura o funzione come parametro attuale per variabile.

Purtroppo, come nel caso delle regole base per l'accesso dei campi varianti e per i valori indefiniti, queste regole risultano difficili o inefficienti da implementare, ed in molti casi non sono implementate affatto. Molto spesso quindi, capita che non viene individuato un errore il cui effetto è sconcertante e la cui causa è difficile da scoprire. Per minimizzare la probabilità di errori di questo genere, si raccomanda molta attenzione nella stesura di programmi che manipolano record varianti.

PROGRAMMA 11 (Formattazione di un testo)

Nella stampa di un testo l'informazione in esso contenuta deve essere formattata per adattarla alla lunghezza di linea consentita dal dispositivo utilizzato.

Si richiede un programma che legga una serie di linee di testo, lunghe al più 80 caratteri, e presenti in output il testo come una serie di linee lunghe al più 60 caratteri. Il testo in input è formato da parole (di non più di 16 lettere), virgole, punti e virgola, punti e "strumenti di impaginazione" come la linea bianca ed il paragrafo nuovo. Qualsiasi linea non vuota che inizia con tre o più spazi è da considerarsi l'inizio di un nuovo paragrafo.

L'output deve conservare le linee vuote e la paragrafazione del testo in input. Il posizionamento dei simboli di punteggiatura è sottoposto alle regole usuali – un simbolo di punteggiatura deve seguire immediatamente la parola precedente, ed entrambi si devono trovare sulla stessa linea, con il simbolo di punteggiatura seguito da uno spazio, a meno che non sia l'ultimo carattere della linea.

```
begin
  leggi un elemento;
  repeat
    scrivi un elemento;                                    (22)
    leggi un elemento
  until finetesto
end
```

Inizialmente il programma si potrebbe concepire come un ciclo (22), che identifica l'elemento successivo del testo, e lo trasferisce nel testo in output. È conveniente stabilire che la fine del testo sia indicata da un "elemento" speciale che viene letto, ma non scritto, dal programma. Lo schema risulta tuttavia inadeguato se si considera che il modo in cui una parola viene scritta dipende dalla condizione che il simbolo successivo sia o no un simbolo di punteggiatura. L'input del programma deve perciò essere avanti di un elemento rispetto all'output, e quindi va riscritto come mostrato in (23).

```
begin
  leggi il prossimo elemento;
  repeat
    questoelemento := prossimoelemento;                    (23)
    leggi prossimoelemento;
    stampa questoelemento
  until il prossimoelemento è finetesto
end.
```

Questo processo si può esprimere per mezzo di due procedure *leggiprossimoelemento* e *scriviprossimoelemento,* di cui l'ultima ha accesso all'informazione relativa all'elemento successivo per realizzare l'operazione richiesta.

Come si potrebbero rappresentare gli elementi che compaiono nel testo? Essi possono essere parole lunghe fino a 16 lettere, simboli di punteggiatura (virgola, punto e virgola, punto) e "strumenti di impaginazione" (una linea bianca o un paragrafo nuovo) oppure l'elemento che indica la fine del testo. Dobbiamo perciò definire un tipo-record variante, come riportato in (24), per descrivere queste possibilità.

```
type tipoelemento = (parola, punteggiatura, impaginazione, finetesto);
     elementotesto = record
                       case tipo : tipoelemento of
                         parola : (lunghezza : 1..16;
                                   lettere : array [1..16] of char);      (24)
                         punteggiatura : (simbolo : char);
                         impaginazione : (forma : (lineabianca, paragrafo));
                         finetesto : ( )
                     end
```

Si possono ora usare due variabili di tipo *elementotesto,* per mantenere le informazioni relative agli elementi elaborati da *leggiprossimoelemento* e *scriviprossimoelemento*:

```
var questoelemento, prossimoelemento : elementotesto;
```

La procedura *leggiprossimoelemento* deve leggere i caratteri da cui è costituito l'ele-

mento successivo dell'input, e lasciare la sua descrizione nella variabile *prossimoelemento*. Per alcuni elementi come le parole e i paragrafi nuovi non si può determinare se l'elemento è completo, finché non è stato letto il carattere successivo–a questo scopo l'input carattere per carattere deve risultare un carattere avanti rispetto all'elemento finora scandito. Questo comporta, però, un ulteriore problema alla fine di una linea–se viene letto lo spazio che il sistema PASCAL inserisce in più alla fine di ogni linea letta, la condizione di fine linea va perduta. Piuttosto che definire una procedura che qualche volta è un carattere avanti e qualche volta no, si adotta la seguente strategia.

All'inizio di ogni nuova linea di input l'intera linea di caratteri viene letta in un array

```
linea : array [1.. maxinput] of char
```

All'interno dell'array *linea* viene memorizzato un carattere speciale che non può capitare nel testo, ' ↑ ' ad esempio, subito dopo l'ultimo carattere letto della linea. Il processo di scansione degli elementi si può ora programmare come un'operazione sull'array *linea,* e può sempre essere un carattere avanti dato che si incontra il carattere ' ↑ ' alla fine di ogni linea. Quando si incontra questo carattere come primo carattere diverso dal carattere spazio di un elemento, la procedura *leggiprossimoelemento* può leggere la successiva linea di input e comportarsi di conseguenza. La procedura *leggiprossimoelemento* è riportata in (25).

```
procedure leggiprossimoelemento;
begin
  salta gli spazi se necessario;
  if il carattere successivo è ' ↑ '                                    (25)
    then if eof (input)
           then l'elemento è finetesto
           else begin
                  leggi la linea successiva;
                  salta e conta gli spazi iniziali;
                  if il carattere successivo è ' ↑ '
                    then l'elemento è una linea bianca
                    else if più di due spazi
                           then l'elemento è paragrafo
                           else cerca il primo elemento non spazio della linea
                end
    else cerca il primo elemento non spazio della linea
end
```

A questo punto, *cerca il primo elemento non spazio* della linea può essere considerata una procedura (26) il cui effetto dipende dal carattere della linea sul quale sta operando.

```
procedure cercaelementononspazio;
begin
  with prossimoelemento do
```

```
case linea [puntatore] of
  'A', 'B', ..., 'Z' : begin
                         l'elemento è una parola;
                         scandisci la parola e registra la sua lunghezza
                         e le sue lettere
                       end;                                          (26)
  ',',';','.' : begin
                  l'elemento è un simbolo di punteggiatura;
                  scandisci e registra il simbolo
                end
end
```

Introducendo la variabile

puntatore : 1..*maxinput*

in cui viene registrata la posizione nell'array *linea* del successivo carattere da considerare, queste procedure si possono facilmente scrivere in PASCAL. Si osservi che *linea* e *puntatore* devono a loro volta essere variabili globali, perché il loro valore deve essere conservato tra le chiamate di *leggiprossimoelemento* e *cercaprimononspazio*.

Per garantire che la prima linea di input venga letta dalla prima chiamata di *leggiprossimoelemento* occorre inizializzare *linea* e *puntatore* nel seguente modo:

```
linea [1] := ' ↑ ';
puntatore := 1
```

La struttura adatta per la procedura *scriviquestoelemento* è un'istruzione case, che discrimina tra i vari elementi del testo, ed è indicata in (27).

```
with questoelemento do
case tipoelemento of
  parola : begin
             if non c'è spazio sulla linea
               then prendi una nuova linea
               else if non è l'inizio della linea
                      then metti uno spazio prima della parola;
             scrivi la parola
           end;
  punteggiatura : scrivi il simbolo;
  impaginazione : begin
                    if la linea è piena in parte                     (27)
                      then prendi una nuova linea;
                    if forma = lineabianca
                      then prendi una nuova linea
                      else prendi un nuovo paragrafo
                  end
end
```

Anche questa procedura si può facilmente esprimere in PASCAL con l'introduzione

della variabile globale

spazioadestra : 0..*maxoutput*

in cui viene registrato il numero delle posizioni restanti della linea di input in cui ci sono caratteri disponibili. Il controllo della variabile *prossimoelemento* quando *questoelemento* è una *parola* consente di impedire la separazione su due linee tra una parola ed il simbolo di punteggiatura immediatamente seguente.

Il Listato 11 mostra il programma PASCAL completo. Notare l'uso delle costanti per definire la lunghezza delle linee di input e di output—questo rende il programma facilmente modificabile rispetto al numero di caratteri delle linee di input e di output.

Il campione di output presentato è stato ottenuto con gli stessi dati di input usati nel Programma 9.

LISTATO 11

```
PROGRAM TEXTEDITOR(INPUT,OUTPUT);
CONST MAXINPUT  = 81;
      FINELINEA = '%';
      MAXOUTPUT = 60;
TYPE  TIPOELEMENTO  = (PAROLA,PUNTEGGIATURA,
                       IMPAGINAZIONE,FINETESTO);
      ELEMENTOTESTO =
                    RECORD
                       CASE TIPO:TIPOELEMENTO OF
                          PAROLA:
                                (LUNGHEZZA: 1..16;
                                 LETTERE: ARRAY [1..16] OF CHAR);
                          PUNTEGGIATURA:(SIMBOLO : CHAR);
                          IMPAGINAZIONE:(FORMA :
                                         (LINEABIANCA,PARAGRAFO));
                          FINETESTO     :()
                    END;
      AMPIEZZALINEA = 1..MAXINPUT;

VAR   QUESTOELEMENTO,PROSSIMOELEMENTO : ELEMENTOTESTO;
      LINEA : ARRAY [AMPIEZZALINEA] OF CHAR;
      PUNTATORE: AMPIEZZALINEA;
      SPAZIOADESTRA : 0..MAXOUTPUT;

      PROCEDURE LEGGIPROSSIMOELEMENTO;

         PROCEDURE LEGGILINEA;
         VAR  I : AMPIEZZALINEA;
         BEGIN
            I:= 1;
            WHILE NOT EOLN(INPUT) DO
            BEGIN
               READ(LINEA[I]);
               I:= I + 1
            END;
```

```
      READLN;
      LINEA[I]:= FINELINEA;
      PUNTATORE:= 1
   END; (* LEGGILINEA *)

   PROCEDURE CERCAELEMENTO;
   VAR  L : 0..16;
   BEGIN
      WITH PROSSIMOELEMENTO DO
         CASE LINEA[PUNTATORE] OF
            'A','B','C','D','E','F','G',
            'H','I','J','K','L','M','N',
            'O','P','Q','R','S','T','U',
            'V','W','X','Y','Z' :
             BEGIN
                TIPO:= PAROLA;
                L:= 0;
                REPEAT
                   L:= L + 1;
                   LETTERE[L]:= LINEA[PUNTATORE];
                   PUNTATORE:= PUNTATORE + 1
                UNTIL (LINEA[PUNTATORE] < 'A') OR
                      (LINEA[PUNTATORE] > 'Z');
                LUNGHEZZA:= L
             END;
             ',',';','.',':',';'  :
             BEGIN
                TIPO:= PUNTEGGIATURA;
                SIMBOLO:= LINEA[PUNTATORE];
                PUNTATORE:= PUNTATORE + 1
             END
          END
   END; (* CERCAELEMENTO *)


BEGIN (* LEGGIPROSSIMOELEMENTO *)
   WHILE LINEA[PUNTATORE] = ' ' DO
      PUNTATORE:= PUNTATORE + 1;
   IF LINEA[PUNTATORE] = FINELINEA
      THEN IF EOF(INPUT)
              THEN PROSSIMOELEMENTO.TIPO:= FINETESTO
              ELSE BEGIN
                      LEGGILINEA;
                      WHILE LINEA[PUNTATORE] = ' ' DO
                         PUNTATORE:= PUNTATORE + 1;
                      WITH PROSSIMOELEMENTO DO
                         IF LINEA[PUNTATORE] = FINELINEA
                            THEN BEGIN
                                    TIPO:= IMPAGINAZIONE;
                                    FORMA:= LINEABIANCA
                                 END
                            ELSE IF PUNTATORE > 2
                                    THEN BEGIN
                                            TIPO:= IMPAGINAZIONE;
```

```
                                        FORMA:= PARAGRAFO
                                    END
                                ELSE CERCAELEMENTO
                    END
            ELSE CERCAELEMENTO
    END; (* LEGGIPROSSIMOELEMENTO *)

    PROCEDURE SCRIVIQUESTOELEMENTO;
    VAR  SPAZIONECESSARIO : 1..18;
         I : 1..16;

         PROCEDURE PRENDILINEANUOVA;
         BEGIN
            WRITELN;
            SPAZIOADESTRA:= MAXOUTPUT
         END; (* PRENDILINEANUOVA *)

BEGIN
   WITH QUESTOELEMENTO DO
      CASE TIPO OF
         PAROLA :
         BEGIN
            IF PROSSIMOELEMENTO.TIPO = PUNTEGGIATURA
               THEN SPAZIONECESSARIO:= LUNGHEZZA + 2
               ELSE SPAZIONECESSARIO:= LUNGHEZZA + 1;
            IF SPAZIONECESSARIO > SPAZIOADESTRA
               THEN PRENDILINEANUOVA
               ELSE IF SPAZIOADESTRA <> MAXOUTPUT
                       THEN BEGIN
                               WRITE(' ');
                               SPAZIOADESTRA:= SPAZIOADESTRA - 1
                            END;
            FOR I:= 1 TO LUNGHEZZA DO
               WRITE(LETTERE[I]);
            SPAZIOADESTRA:= SPAZIOADESTRA - LUNGHEZZA
         END;
         PUNTEGGIATURA :
         BEGIN
            WRITE(SIMBOLO);
            SPAZIOADESTRA:= SPAZIOADESTRA - 1
         END;
         IMPAGINAZIONE :
         BEGIN
            IF SPAZIOADESTRA < MAXOUTPUT
               THEN PRENDILINEANUOVA;
            IF FORMA = LINEABIANCA
               THEN PRENDILINEANUOVA
               ELSE BEGIN
                       WRITE('  ');
                       SPAZIOADESTRA:= SPAZIOADESTRA - 2
                    END
         END
      END
END; (* SCRIVIQUESTOELEMENTO *)
```

```
BEGIN (* PROGRAMMA PRINCIPALE *)
   LINEA[1]:= FINELINEA;
   PUNTATORE:= 1;
   SPAZIOADESTRA:= MAXOUTPUT;
   LEGGIPROSSIMOELEMENTO;
   REPEAT
      QUESTOELEMENTO:= PROSSIMOELEMENTO;
      LEGGIPROSSIMOELEMENTO;
      SCRIVIQUESTOELEMENTO
   UNTIL PROSSIMOELEMENTO.TIPO = FINETESTO
END.
```

```
QUOT SINT GENERA PRINCIPATUM ET QUIBUS MODIS ACQUIRANTUR

TUTTI GLI STATI, TUTTI E DOMINI CHE HANNO AVUTO ED HANNO
IMPERIO SOPRA GLI UOMINI, SONO STATI E SONO O REPUBBLICHE O
PRINCIPATI.

   I PRINCIPATI SONO, O EREDITARII, DE QUALI EL SANGUE DEL
LORO SIGNORE NE SIA SUTO LUNGO TEMPO PRINCIPE, O E SONO
NUOVI.
```

ESERCIZI

10.1 Estendere il Programma 10 in modo che siano registrate anche le reti fatte e subite da ciascuna squadra, e che la posizione in classifica delle squadre a pari punti sia determinata in base alla differenza reti (reti fatte − reti subite).

10.2 Definire un tipo-record, ogni valore del quale denoti un punto su una griglia *x-y* in cui *x* ed *y* possono assumere valori interi nell'intervallo 1, 100. Scrivere un programma che legge quattro coppie di valori che rappresentano i vertici *A, B, C, D* di un quadrilatero in ordine ciclico e determini se *ABCD* è un quadrato, un rettangolo o altro.

10.3 Definire un tipo-record contenente il nome, i voti ed il loro totale per memorizzare in una variabile di questo tipo record le informazioni relative agli studenti considerati nel Programma 2. Scrivere una procedura che legge la valutazione di uno studente da una linea di input ed immagazzina l'informazione in un record del tipo definito. Usare questa procedura in una nuova versione del Programma 2 che tabuli, come la precedente, i risultati della classe ma in ordine decrescente di voto totale.

10.4 Definire un tipo record variante per descrivere la forma e le dimensioni di figure geometriche quali cerchi, quadrati, rettangoli e triangoli. Scrivere una procedura che legge la descrizione di una figura geometrica e la registra in un record del tipo definito. Ogni descrizione in input è su una linea diversa ed inizia con una lettera *C, Q, R* o *T* che indica la forma. La lettera è seguita da uno, due o tre numeri reali che specificano la lunghezza del raggio, del lato o dei lati appropriati; per esempio

```
C   12.1
R   30.4  17.5
Q   16.7
T   40.6  27.9  21.9
```

Scrivere una funzione che determini l'area del figura descritta nel record. Usare la procedura e la funzione in un programma che legge una sequenza di descrizioni di figure e stampa la descrizione della figura avente l'area massima.

11. Insiemi

11.1. LA NOZIONE DI INSIEME

Nel Cap. 3 abbiamo definito un tipo come l'insieme dei valori che un dato di quel tipo può assumere. Un dato ha, in ogni momento, un valore che è esattamente uno dei valori di tale insieme. In pratica, capita spesso di avere dati i cui valori siano a loro volta insiemi di valori di qualche altro tipo.

Si consideri l'esperimento classico che consiste nel mescolare colori da sorgenti monocromatiche, come succede sullo schermo di un televisore a colori. Potremmo avere tre fonti primarie di colore, rosso, giallo e blu, che consideriamo come possibili valori di un tipo:

colorefondamentale = (rosso, giallo, blu)

Questi colori primari prossono essere mescolati in qualsiasi combinazione. Per esempio in un punto potremmo usare una combinazione di tutti e tre i colori ed indicarla così:

[*rosso, giallo, blu*]

(Dato che l'ordine di mescolamento non è importante, non lo è neppure l'ordine in cui scriviamo i colori fondamentali. Avremmo potuto scrivere [*giallo, rosso, blu*], oppure [*blu, giallo, rosso*] o qualsiasi altro ordinamento possibile – ci sono sei modi possibili di denotare questa miscela di tre colori, tutti equivalenti).

D'altra parte, potremmo usare solo due colori, per esempio una delle seguenti coppie:

[*rosso, giallo*]
[*rosso, blu*]
[*giallo, blu*]

(In questo caso ci sono due modi equivalenti di denotare una miscela di due colori).

Inoltre potremmo usare una "miscela" composta soltanto da uno dei tre possibili colori (usare insomma il colore fondamentale da solo) cioè uno dei seguenti:

[*rosso*]
[*giallo*]
[*blu*]

Infine, in qualche punto potremmo non volere colorazione alcuna, cioè una "miscela" non comprendente alcuno dei tre colori fondamentali, che denotiamo con

[ ]

Le otto possibili miscele che abbiamo qui elencato sono i soli possibili insiemi (set), o combinazioni non ordinate, dei tre colori fondamentali *rosso, giallo, blu,* e costituiscono in realtà i possibili valori di un tipo che potremmo definire

miscela = **set of** *colorefondamentale*

dove *colorefondamentale* è il *tipo-base* del *tipo-insieme miscela.*

Il PASCAL permette di definire tali tipi insieme, su un insieme ristretto di tipi-base, come segue:

tipo-insieme = "**set**" "**of**" *tipo-base.*
tipo-base = *tipo.*

Il tipo base deve essere un tipo ordinale. In passato le implementazioni del PASCAL imponevano restrizioni ulteriori sui tipi base degli insiemi, ma lo standard PASCAL non le ammette. In pratica, tuttavia, le implementazioni possono comunque imporre alcune restrizioni su insiemi di interi, restringendo di solito il tipo base ad un intervallo che sia un sottoinsieme degli interi non negativi da 0 a *N.*

Come altro esempio di tipo insieme, consideriamo ora i dati per descrivere il meccanismo di controllo di un ascensore di una costruzione a più piani. In ogni momento il meccanismo deve conservare traccia delle chiamate attive, che possono variare da nessuna chiamata (quando nessuno vuole l'ascensore), a chiamate da tutti i piani (in ora di punta). Le chiamate attive possono essere viste come valori del tipo insieme *chiamate* così definito:

piano = 1..*ultimo*;
chiamate = **set of** *piano*;

Si possono allora dichiarare variabili di tipo *chiamate*; per esempio

VAR *questoviaggio, chiamateattive* : *chiamate*

11.2. MANIPOLAZIONE DI INSIEMI

Una volta definito un tipo insieme, in un programma PASCAL si possono dichiarare e manipolare variabili di quel tipo. Oltre alla copia e all'assegnazione, il PASCAL permette di applicare a valori di un tipo insieme una serie di operazioni specifiche.

11.2.1. Costruzione

Un valore di tipo insieme si costruisce specificando i suoi elementi, o membri. Questo si ottiene mediante l'uso della forma sintattica *insieme,* che nel Cap. 4 è stata presentata come alternativa di un *fattore* entro un'*espressione.* La definizione è la seguente

insieme = " [" *lista-di-elementi* "] ".
lista-di-elementi = [*elemento* { "," *elemento* }].

elemento = *espressione* [".." *espressione*].

La forma "[ ]" denota un insieme che non contiene nessun elemento, ed è nota come *insieme vuoto.*

Un insieme non vuoto consiste di una o più denotazioni di elemento, separate da virgole, e racchiuse da parentesi quadre. Tutte le espressioni che appaiono debbono essere dello stesso tipo, che è il tipo base dell'insieme.

Una denotazione può essere costituita o da un'espressione singola, il cui valore determina il corrispondente elemento dell'insieme, oppure da due espressioni, che determinano un intervallo continuo di elementi dell'insieme.

Per esempio,

[*rosso*]	denota l'insieme di tipo *miscela* contenente il solo elemento *rosso.*
[*rosso..blu*]	denota l'insieme di tipo *miscela* contenente i tre elementi *rosso, giallo, blu.*
[1..3, *f, ultimo*]	denota l'insieme di tipo *chiamate* contenente i piani 1, 2, 3, *ultimo,* ed il valore corrente di *f,* dove *f* è una variabile di tipo *piano.*

Notare che due distinte denotazioni di elemento possono indicare lo stesso valore. Per definizione un elemento può ricorrere soltanto una volta in un insieme. Così un insieme costruito come [*I, J*] dove i valori di *I* e di *J* sono uguali, risulta un insieme di un solo elemento.

La denotazione *I..J,* dove i valori di *I* e *J* siano tali che $I > J$, non denota nessun elemento. Così [*I..J*] in questo caso è l'insieme vuoto.

Gli insiemi costruiti sono usati solitamente per inizializzare le variabili di tipo insieme; per esempio

```
chiamateattive := [ ]
miscela1 := [rosso, blu]
```

L'insieme vuoto gioca inoltre un ruolo importante nel confronto tra insiemi, mentre gli insiemi monoelemento sono usati nell'aggiornamento di insiemi come descritto nel seguito.

Poiché un insieme costruito non indica l'intervallo preciso dei valori che formano il suo tipo base, nei paragrafi successivi definiremo, mediante regole speciali, l'assegnazione e le operazioni su insiemi. Queste sono le seguenti:

(a) In un'espressione un fattore di tipo **set of** *S,* dove *S* è un sottointervallo di qualche tipo ospite *T,* è considerato di tipo **set of** *T.*

(b) Il valore di una variabile di tipo **set of** *T* è compatibile per assegnazione con un tipo della forma **set of** *S* se il valore di ogni elemento appartenente ad esso è compatibile per assegnazione col tipo *S.*

11.2.2. Test di appartenenza

Il PASCAL fornisce cinque operatori relazionali, mediante i quali si può studiare l'appartenenza di valori agli insiemi. Quattro di questi sono gli operatori, ormai fami-

liari, =, <>, <=, >=, usati con due operandi dello stesso tipo insieme. Il quinto è l'operatore **in**, usato con un operando di un tipo base ed un secondo operando del corrispondente tipo insieme.

Gli operatori = e <> sono usati per controllare l'equivalenza. Due insiemi sono uguali se contengono entrambi esattamente gli stessi elementi. Così

[*rosso, blu*] = [ ]	dà *false*
[*rosso*] <> [*blu*]	dà *true*
[*rosso, giallo, blu*] = [*rosso..blu*]	dà *true*

Gli operatori <= e >= sono usati per testare l'inclusione insiemistica. Un insieme *a* è contenuto, o incluso, in un insieme *b*, scritto *a* <= *b*, se ciascun elemento di *a* è anche elemento di *b*. Quindi

[*rosso, blu*] <= [*rosso, giallo, blu*]	dà *true*
[*rosso, blu*] <= [*rosso, blu*]	dà *true*
[*rosso, blu*] <= [*rosso, giallo*]	dà *false*
[*rosso, blu*] <= [*blu*]	dà *false*

Notare che l'insieme vuoto [ ] è incluso in qualsiasi altro insieme, cosicché

[ ] <= *b* dà *true*

qualsiasi sia il valore dell'insieme *b*.

La relazione *a* >= *b* va letta *a include b*, e dà *true* se ogni elemento di *b* è elemento di *a*, *false* altrimenti. Così

[*rosso, blu*] >= [*rosso, giallo, blu*]	dà *false*
[*rosso, blu*] >= [*rosso, blu*]	dà *true*
[*rosso, blu*] >= [*rosso, giallo*]	dà *false*
[*rosso, blu*] >= [*blu*]	dà *true*

Notate che entrambi gli operatori possono restituire *false* per qualche coppia di operandi. In ciò differiscono dalle loro controparti aritmetiche – se *a* e *b* sono interi, allora *((a <= b)* **or** *(a >= b))* deve essere *true*, ma quando *a* e *b* sono insiemi questa espressione può essere *false*.

L'operatore **in** è usato per testare la presenza di un singolo elemento di un insieme. Se *x* è un valore del tipo base ed *a* un valore del corrispondente tipo insieme, allora

x **in** *a* dà *true* se *x* è un elemento di *a*,
false altrimenti.

Per esempio

rosso **in** [*rosso, giallo*]	dà *true*
rosso **in** [*rosso*]	dà *true*
rosso **in** [*giallo, blu*]	dà *false*
rosso **in** [ ]	dà *false*

Come vedremo, gli operatori per testare l'appartenenza sono usati comunemente per controllare l'elaborazione di dati di tipo insieme; per esempio

if *questopiano* **in** *chiamateattive* **then** ... **else** ...
while *chiamateattive* = [ ] **do** ...

Tuttavia, l'operatore **in** è spesso utile in contesti dove non è necessaria la manipolazione esplicita di insiemi, come mezzo per esprimere un test di uguaglianza con parecchie alternative. Così scrivere

if *d* **in** [*lunedi, mercoledi, venerdi*] **then** ...

è più leggibile, e in molte implementazioni più efficiente, dell'oscuro

if *(d=lunedi)* **or** *(d=mercoledi)* **or** *(d=venerdi)* **then** ...

Analogamente, l'operatore **in** fornisce un modo conveniente di esprimere il test molto comune

c è una lettera

dove *c* è una variabile di tipo *char.* Se le lettere formano un sottointervallo continuo dei valori di *char,* ciò può essere scritto

if *c* **in** [*'A'..'Z'*] **then** ...

o, se no, come

```
if c in ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
         'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
    then ...
```

11.2.3. Aritmetica insiemistica

La manipolazione matematica di insiemi è di solito espressa in termini di tre operazioni, ciascuna delle quali assume due operandi di tipo insieme restituendo un terzo insieme, e cioè *unione, intersezione,* e *differenza insiemistica,* o *complemento relativo.* Anche il PASCAL fornisce tali operazioni, denotandole mediante gli operatori +, * e −. Se usati con due operandi dello stesso tipo insieme, essi producono un risultato del medesimo tipo insieme, nel modo che segue:

- *a* + *b* dà l'unione di *a* e *b,* cioè l'insieme di tutti i valori che sono in *a* o in *b* o in entrambi.
- *a* * *b* dà l'intersezione di *a* e *b,* cioè l'insieme di tutti i valori che sono in *a* ed anche in *b.*
- *a* − *b* dà il complemento relativo di *a* e *b,* cioè l'insieme di tutti i valori che sono in *a* ma non in *b.*

Per esempio,

[*rosso,blu*] + [*rosso,giallo*] dà [*rosso,giallo,blu*]
[*rosso,blu*] * [*rosso,giallo*] dà [*rosso*]
[*rosso,blu*] − [*rosso,giallo*] dà [*blu*]

Gli operatori + e − sono spesso usati con insiemi monoelemento come mezzo per aggiungere o togliere un dato elemento di un insieme esistente. Così

- *a* := *a* + [*x*] aggiunge *x* all'insieme *a* − se *a* già contiene *x* l'operazione non ha effetto.
- *a* := *a* − [*x*] toglie *x* dall'insieme *a* − se *a* non contiene *x* l'operazione non ha effetto.

La funzione (1) accetta un intero positivo *N* come parametro, e conta il numero di cifre distinte della sua rappresentazione decimale.

```
function cifredistinte (N : interononnegativo) : interopositivo;
var cifra : 0..9; cont : interopositivo;
    cifreprecedenti : set of 0..9;
begin
    cifreprecedenti := [ ]; cont := 0;
    repeat
       cifra := N mod 10;
       if not (cifra in cifreprecedenti)                    (1)
         then begin
                 cont := cont + 1;
                 cifreprecedenti := cifreprecedenti + [cifra]
              end;
       N := N div 10
    until N = 0;
    cifredistinte := cont
end
```

PROGRAMMA 12 (Servizio cuori solitari)

Un servizio elettronico per cuori solitari registra dati relativi ai suoi clienti su schede perforate, col seguente formato:

colonne 1-60	nome e indirizzo
colonna 61	sesso (M o F)
colonne 62-63	età (due cifre decimali)
colonne 64-69	interessi (una x perforata sulla colonna corrispondente denota un interesse in arte, libri, musica, teatro, politica, sport)

Si richiede un programma che legge una scheda contenente un nuovo cliente, seguita da un pacco di schede con tutti i clienti esistenti. Per esempio, le prime schede dell'input potrebbero essere quelle mostrate in (2).

```
ALEX ANDERSON      26 CALIFORNIA AVENUE, HUYTON.     M24 X    X
ANNE ENGLISH       21 HOME GARDENS, HUYTON.          F 19  XX
JULIA MONTEITH     35 HEATH AVENUE, BURY.            F 33  X  X      (2)
MAURICE SHORT      21 GREEN STREET, WIGAN.           M18X    XX
KATHLEEN BRYANS    3  BLUE AVENUE, WARRINGTON.       F 21  XX
```

Il programma dovrebbe stampare un elenco di tutti i clienti esistenti 'compatibili' col nuovo, cioè di sesso opposto, con una differenza di età minore di 10 anni e con almeno un interesse in comune.

```
begin
    leggi nuovo cliente;
    repeat                                                  (3)
       leggi cliente successivo;
       if compatibile col nuovo
```

```
            then stampa nome e indirizzo
        until eof (input)
    end
```

La forma generale del programma richiesto è mostrata in (3). Per raffinarla ulteriormente dobbiamo decidere come rappresentare i dati relativi a ciascun cliente. Chiaramente la struttura è quella di un record, con campi contenenti nome e indirizzo, sesso, età ed interessi del cliente in esame. Il nome ed indirizzo possono essere rappresentati come stringa lunga 60, il sesso come valore di un opportuno tipo a due valori, l'età come sottointervallo degli interi. Poiché gli interessi sono una combinazione qualsiasi di argomenti di una lista predefinita, una rappresentazione conveniente sarà sottoforma di insieme di elementi di un corrispondente tipo base *argomenti*. La rappresentazione completa è quindi quella mostrata in (4).

```
type sessi = (maschile, femminile);
     argomenti = (arte,libri,musica,teatro,politica,sport);
     cliente = record
                   identita : packed array [1..60] of char;
                   sesso : sessi;
                   eta : 16..99;
                   interessi : set of argomenti
               end;
```

(4)

Con tale rappresentazione, il processo di lettura dei dati di qualsiasi cliente si può esprimere con la procedura (5)

```
procedure leggischeda (var c : cliente);
var i : 1..60;
    ch : char;
    t : argomenti;
begin
    with c do
      begin
          for i := 1 to 60 do read (identita [i]);
          read (ch);
          if ch = 'M'
            then sesso := maschile
            else sesso := femminile;
          read (eta);
          interessi := [ ];
          for t := arte to sport do
          begin
              read (ch);
              if ch = 'X'
                then interessi := interessi + [ t]
          end
      end;
    readln
end
```

(5)

Notare come l'insieme interessi sia costruito per accumulazione di un nuovo elemento ogni volta che si trova una *'X'* nella posizione corrispondente. Il *readln* finale assicura che l'input sia allineato correttamente nella prima colonna della scheda successiva alla chiamata seguente di *leggischeda,* oppure alla condizione di fine file, quando è stata letta l'ultima scheda. In un vero ambiente di elaborazione la procedura *leggischeda* potrebbe pure controllare i dati letti, per esempio verificare che il sesso sia stato perforato come *'M'* o *'F'* e nient'altro, e lo stesso per gli altri campi.

Come si testa la compatibilità tra due clienti? I test d'età e sesso sono semplici. Per determinare che i due clienti abbiano almeno un interesse in comune, costruiamo l'intersezione dei loro insiemi di interessi, e controlliamo che non sia vuota. Esprimiamo così la compatibilità con la funzione (6).

```
function compatibile (c1, c2: cliente) : boolean;
begin
   compatibile := (c1.sesso <> c2.sesso) and
                  (abs (c1.eta - c2.eta) < 10) and                    (6)
                  (c1.interessi * c2.interessi <> [ ])
end
```

Il listato 12 mostra il programma completo e l'output prodotto.

LISTATO 12

```
PROGRAM SERVIZIOELABORAZIONEDATI (INPUT,OUTPUT);

TYPE SESSI      = (MASCHILE, FEMMINILE);
     ARGOMENTI  = (ARTE, LIBRI, MUSICA, TEATRO, POLITICA, SPORT);
     CLIENTE    = RECORD
                     IDENTITA  : PACKED ARRAY [1..60] OF CHAR;
                     SESSO     : SESSI;
                     ETA       : 16..99;
                     INTERESSI : SET OF ARGOMENTI;
                  END;

VAR  NUOVOCLIENTE, PROSSIMOCLIENTE : CLIENTE;

     PROCEDURE LEGGISCHEDA (VAR C : CLIENTE);
     VAR  I  : 1..60;
          CH : CHAR;
          A  : ARGOMENTI;
     BEGIN
        WITH C DO
           BEGIN
              FOR I:= 1 TO 60 DO
                 READ(IDENTITA[I]);
              READ(CH);
              IF CH = 'M'
                 THEN SESSO:= MASCHILE
                 ELSE SESSO:= FEMMINILE;
              INTERESSI:= [ ];
              FOR A:= ARTE TO SPORT DO
```

```
            BEGIN
               READ(CH);
               IF CH = 'X'
                  THEN INTERESSI:= INTERESSI + [A]
            END
         END;
      READLN
   END; (* LEGGISCHEDA *)

   FUNCTION COMPATIBILE (C1,C2 : CLIENTE) : BOOLEAN;
   BEGIN
      COMPATIBILE:= (C1.SESSO <> C2.SESSO) AND
                    (ABS(C1.ETA - C2.ETA) < 10) AND
                    (C1.INTERESSI * C2.INTERESSI <> [ ])
   END; (* COMPATIBILE *)

BEGIN (* PROGRAMMA PRINCIPALE *)
   LEGGISCHEDA(NUOVOCLIENTE);
   REPEAT
      LEGGISCHEDA(PROSSIMOCLIENTE);
      IF COMPATIBILE(NUOVOCLIENTE,PROSSIMOCLIENTE)
         THEN WRITELN(PROSSIMOCLIENTE.IDENTITA)
   UNTIL EOF(INPUT)
END.


JULIA MONTEITH    35 HEATH AVENUE, BURY.
KATHLEEN BRYANS   3  BLUE AVENUE, WARRINGTON.
LINDA SMITH       29 YELLOW STREET,BOLTON.
MARY STEWART      23 IDAHO GARDENS, BURNLEY.
```

PROGRAMMA 13 (Colorazione di una mappa)

Un continente comprende *N* nazioni, ciascuna delle quali confina con una o più delle altre. Si deve colorare una carta del continente che riporta le nazioni in modo tale che due nazioni confinanti non abbiano lo stesso colore. Occorre un programma che trovi un'opportuna colorazione. (È stato dimostrato che quattro colori sono sempre sufficienti per colorare ogni possibile disposizione di nazioni).

Il problema è tipico di una grande classe di problemi risolti per *tentativi (trial and error)*. Il metodo risolutivo consiste nel miglioramento incrementale di una soluzione parziale, esaminando una nazione alla volta, scegliendo un colore adatto per tale nazione, e procedendo a considerare la successiva. Se risulta impossibile trovare un colore opportuno per la nazione corrente, il metodo deve 'tornare indietro' *(backtrack)* all'ultima nazione già colorata per cui esista un'alternativa e ricominciare da quel punto. Questo processo è ben espresso come procedura ricorsiva, che nel problema della colorazione della mappa ha la struttura generale mostrata in (7).

BORDER RELATIONSHIPS

	1 2	3 4	5 6	7 8	9 10	11 12	13 14	15 16	17 18	19 20	21 22	23 24	25 26	27 28	29 30
1 Albania						1									1
2 Andorra					1							1			
3 Austria			1			1		1 1					1		1 1
4 Belgio					1	1			1	1					
5 Bulgaria						1						1		1	1
6 Cecoslovacchia		1			1	1				1				1	1
7 Danimarca						1									
8 Finlandia									1				1	1	
9 Francia	1	1				1		1	1			1	1		
10 Germania Est			1			1				1					
11 Germania Ovest		1 1	1	1	1 1				1	1			1		
12 Grecia	1		1											1	1
13 Irlanda											1				
14 Islanda															

15	Italia		1			1								1		1
16	Liechtenstein		1											1		
17	Lussemburgo		1			1	1									
18	Norvegia				1									1	1	
19	Paesi Bassi		1				1									
20	Polonia			1		1									1	
21	Portogallo												1			
22	Regno Unito							1								
23	Romania			1											1	1 1
24	Spagna	1				1						1				
25	Svezia				1					1						
26	Svizzera		1			1	1		1 1							
27	Turchia			1			1								1	
28	U.R.S.S.			1	1					1	1		1		1	1
29	Ungheria		1	1									1		1	1
30	Yugoslavia	1	1	1			1		1				1			1

```
procedure coloranazione (i : nazione);
begin
    for ogni colore disponibile do
    if questo colore non è stato scelto per una nazione confinante
      then begin                                                    (7)
                scegli questo colore per la nazione i;
                if i = N
                  then stampasoluzione
                  else coloranazione (i + 1);
           end
end
```

La procedura deve essere usata in un programma che prima inizializza i dati per indicare che finora non sono stati scelti colori, e poi la chiama con numero di nazione uguale ad 1.

Così com'è questa procedura è capace di trovare tutte le possibili soluzioni, perché continua ad esplorare colorazioni alternative, anche dopo che è stata stampata una soluzione completa. Se si richiede un'unica soluzione, il processo ricorsivo si può interrompere con una istruzione goto nella procedura *stampasoluzione* che è nella parte di programma che ha chiamato originariamente la procedura ricorsiva.

Per poter raffinare ulteriormente la procedura, occorre decidere come rappresentare i dati su cui opera.

Le nazioni confinanti con ciascuna nazione del continente possono essere opportunamente rappresentate in una tabella

```
confinanti : array [nazione] of insiemedinazioni
```

dove i tipi *nazione* e *insiemedinazioni* sono definiti

```
nazione = 1..N;
insiemedinazioni = set of nazione;
```

Un possibile schema di colorazione del continente può allora essere rappresentato nella tabella

```
colorato : array [colore] of insiemedinazioni
```

dove il tipo *colore* è definito, diciamo, come

```
colore = (rosso,blu,verde,giallo)
```

I passi non ancora raffinati della nostra procedura ricorsiva si esprimono facilmente in termini di tale rappresentazione. Per testare se un colore *c* sia già stato scelto per una nazione confinante *i*, costruiamo l'intersezione delle nazioni che confinano con *i* con le nazioni colorate con *c*, e verifichiamo che il risultato non sia l'insieme vuoto. Gli altri passi si codificano banalmente e sono riportati nella versione finale (8).

```
procedure coloranazione (i : nazione);
var c : colore;
begin
    for c := rosso to giallo do
      if confinanti [i] * colorato [c] = [ ]
```

```
        then
          begin                                                    (8)
              colorato [c] := colorato [c] + [i];
              if i = N then stampasoluzione
                       else coloranazione (i + 1);
              colorato [c] := colorato [c] − [i]
          end
    end
```

Il Listato 13 mostra un programma completo che usa questa procedura, con le opportune modalità per l'input della tabella dei confini e l'output della soluzione trovata. Riporta inoltre i dati prodotti per i dati d'ingresso corrispondenti alla Tab. 11.1 che rappresenta l'Europa. Se la nazione A confina con qualche altra nazione B, allora appare un '1' nella colonna corrispondente a B della riga di A, altrimenti c'è uno spazio. Così, l'Albania (nazione 1) confina solo con la Grecia (nazione 12) e la Yugoslavia (nazione 30). Le relazioni di confine sono duplicate, cioè le righe della Grecia e della Yugoslavia indicano entrambe che tali nazioni confinano con l'Albania.

I dati effettivi del programma sono una serie di linee, una per ciascuna nazione della tabella di cui sopra. Una riga consiste di un intero a due cifre che è il numero della nazione, del nome della nazione (18 caratteri), e poi di 30 caratteri (o '1' o spazio) che rappresentano le relazioni di confine tra quella nazione e le altre.

LISTATO 13

```
PROGRAM COLORAZIONEMAPPA(INPUT,OUTPUT);

LABEL 99;
CONST N = 30;
      CONTINENTE = 'EUROPA';

TYPE  NAZIONE           = 1..N;
      INSIEMEDINAZIONI  = SET OF NAZIONE;
      COLORE            = (ROSSO, BLU, VERDE, GIALLO);

VAR   CONFINANTI: ARRAY [NAZIONE] OF INSIEMEDINAZIONI;
      COLORATA  : ARRAY [COLORE] OF INSIEMEDINAZIONI;
      NOME      : ARRAY [NAZIONE] OF PACKED ARRAY [1..18] OF CHAR;
      C         : COLORE;

      PROCEDURE LEGGITABELLACONFINI;
      VAR  I, J, M : NAZIONE;
           K : 1..18; CH : CHAR;
      BEGIN
         FOR I:= 1 TO N DO
         BEGIN
            READ(J);
            FOR K:= 1 TO 18 DO READ(NOME[J][K]);
            FOR M:= 1 TO N DO
            BEGIN
               READ(CH);
               IF CH = '1'
```

```
                    THEN CONFINANTI[J]:= CONFINANTI[J] + [M]
            END;
            READLN
         END
      END; (* LEGGITABELLACONFINI *)

      PROCEDURE STAMPASOLUZIONE;
      VAR  I : NAZIONE;
           C : COLORE;
      BEGIN
         WRITELN ('COLORAZIONE DELLA CARTA DI ',CONTINENTE);
         WRITELN ('-----------------------------------');
         WRITELN;
         FOR I:= 1 TO N DO
         BEGIN
            WRITE(NOME[I],'   ');
            C:= ROSSO;
            WHILE NOT(I IN COLORATA[C]) DO
               CASE C OF
                  ROSSO  : WRITELN('ROSSO');
                  VERDE  : WRITELN('VERDE');
                  GIALLO : WRITELN('GIALLO');
                  BLU    : WRITELN('BLU')
               END
         END;
         GOTO 99;
      END; (* STAMPASOLUZIONE *)


      PROCEDURE COLORANAZIONE (I: NAZIONE);
      VAR  C : COLORE;
      BEGIN
         FOR C:= ROSSO TO GIALLO DO
            IF CONFINANTI[I] * COLORATA[C] = [ ]
               THEN BEGIN
                       COLORATA[C]:= COLORATA[C] + [I];
                       IF I = N
                          THEN STAMPASOLUZIONE
                          ELSE COLORANAZIONE (I+1);
                       COLORATA[C]:= COLORATA[C] - [I]
                    END
      END; (* COLORANAZIONE *)


BEGIN (* PROGRAMMA PRINCIPALE *)
   LEGGITABELLACONFINI;
   FOR C:= ROSSO TO GIALLO DO
      COLORATA[C]:= [];
   COLORANAZIONE(1);
   WRITELN('NON ESISTE SOLUZIONE');
   99:
END.
```

```
COLORAZIONE DELLA CARTA DI EUROPA
---------------------------------

ALBANIA                 ROSSO
ANDORRA                 ROSSO
AUSTRIA                 ROSSO
BELGIO                  ROSSO
BULGARIA                ROSSO
CECOSLOVACCHIA          BLU
DANIMARCA               ROSSO
FINLANDIA               ROSSO
FRANCIA                 BLU
GERMANIA EST            ROSSO
GERMANIA OVEST          VERDE
GRECIA                  BLU
IRLANDA                 ROSSO
ISLANDA                 ROSSO
ITALIA                  VERDE
LIECHTENSTEIN           BLU
LUSSEMBURGO             GIALLO
NORVEGIA                BLU
OLANDA                  BLU
POLONIA                 VERDE
PORTOGALLO              ROSSO
REGNO UNITO             BLU
ROMANIA                 BLU
SPAGNA                  VERDE
SVEZIA                  VERDE
SVIZZERA                GIALLO
TURCHIA                 VERDE
UNGHERIA                VERDE
U.R.S.S.                GIALLO
YUGOSLAVIA              GIALLO
```

ESERCIZI

11.1 Date le definizioni di tipo

nazioni = *(Austria, Belgio, Danimarca, Francia, Germania, Irlanda, Italia, Lussemburgo, Olanda, Norvegia, Portogallo, Regno Unito, Spagna, Svezia, Svizzera)*;
insiemedinazioni = **set of** *nazioni*;

e le dichiarazioni di variabile

Alpine, Mediterranee, Atlantiche, MaredelNord, MEC: *insiemedinazioni*;

scrivere le istruzioni di assegnazione appropriate per ciascuna variabile. Usare le variabili per costruire espressioni che denotino i seguenti insiemi:

(a) le nazioni che abbiano coste atlantiche e mediterranee;
(b) le nazioni del MEC con coste sull'Atlantico o sul Mare del Nord;
(c) le nazioni alpine non appartenenti al MEC.

Scrivere un'espressione booleana che determini se

(a) una nazione *c* appartiene al MEC;
(b) una nazione *c* ha coste sul Mediterraneo ma non sull'Atlantico;
(c) tutte le nazioni alpine con coste mediterranee appartengono al MEC.

11.2 Modificare il programma del servizio cuori solitari mostrato nel Listato 12 in modo che ammetta la compatibilità solo se esistono *due* interessi in comune.

11.3 Scrivere un programma che legge due frasi, ciascuna conclusa da un punto, e stampa una lista di tutte le lettere usate in entrambe.

11.4 Un college offre per l'ultimo anno di studi dieci corsi: Arte, Inglese, Francese, Tedesco, Storia, Geografia, Matematica, Fisica, Chimica, Biologia. Il piano di studi di ogni studente dell'ultimo anno è registrato in forma di scheda perforata col nome dello studente nelle colonne 1-20. I corsi cui lo studente si è iscritto sono indicati da una X perforata nella colonna corrispondente tra 21 e 30. Scrivere un programma che legge schede con questo formato e stampa per ciascun corso un elenco degli studenti iscritti. Si può fare l'ipotesi che non vi siano più di 100 studenti che frequentano l'ultimo anno.

11.5 Una fabbrica produce barre d'acciaio la cui lunghezza esatta non è nota fino a produzione ultimata. La divisione commerciale riceve ordini d'acquisto per barre la cui lunghezza deve essere ricavata tagliando le barre prodotte. Per una data barra prodotta di lunghezza L si sceglie dalla lista degli ordini un insieme C tale che tagliando la barra L si possano evadere tutti gli ordini contenuti in C con scarto minimo. Progettare una procedura ricorsiva che costruisca per backtracking C dato L e la lista degli ordini.

Includere tale procedura in un programma che costruisca C a partire dalla seguente lista:

ordine 1	773 mm
ordine 2	548 mm
ordine 3	65 mm
ordine 4	929 mm
ordine 5	548 mm
ordine 6	163 mm
ordine 7	421 mm
ordine 8	37 mm

e da una barra prodotta di lunghezza $L = 1848$ mm.

12.
File

12.1. LA NOZIONE DI FILE

In molte applicazioni si richiede al sistema di elaborazione l'immagazzinamento di notevoli quantità di dati, talvolta in forma semipermanente. I dati, sia perché devono poter essere conservati tra un'esecuzione di un programma ed un'altra, sia perché la memoria centrale può non essere sufficientemente grande da contenerli, sono spesso mantenuti su dispositivi di memoria secondaria, da cui il calcolatore legge gli elementi significativi man mano che vengono richiesti. Gli insiemi di dati mantenuti su dispositivi esterni di memoria, accessibili in lettura ed in scrittura al calcolatore, sono detti, con termine inglese, *file.*

Un file usato da un programma può essere *esterno* o *interno.* Un file esterno è un file il cui contenuto sopravvive all'esecuzione del programma stesso. Può essere usato per memorizzare i dati prodotti da un programma, per darli in input ad un altro programma, o ad una successiva esecuzione dello stesso programma. Anche le informazioni oggetto della comunicazione tra il calcolatore e l'uomo si possono pensare come un file esterno—i canali standard di input e di output, introdotti nel Cap. 5, sono casi particolari di file esterni in PASCAL. Come vedremo, le operazioni usate per l'input e per l'output come *read, write,* ecc., si possono applicare su altri file in modo analogo.

Nei programmi sin qui sviluppati abbiamo sempre usato implicitamente il concetto di file per l'input dei dati e per l'output dei risultati. Alcuni di essi si potrebbero migliorare con l'introduzione esplicita dei file. Si consideri il Programma 12 per il servizio cuori solitari. In esso i dati relativi ad un cliente venivano memorizzati su una scheda perforata e, per ogni nuovo cliente, veniva letto (come file standard di input) un pacco di schede, contenente i dati relativi al nuovo cliente, seguiti dai dati di tutti gli altri clienti già esistenti. Una soluzione più conveniente ed economica potrebbe essere conservare i dati sui clienti già esistenti su un dispositivo di memoria, come un disco o un nastro magnetico, e fare un'operazione di input soltanto per i dati relativi ad un nuovo cliente ad ogni esecuzione di programma. Il programma potrebbe leggere i dati relativi ai clienti già esistenti dal file dei clienti, selezionando, come prima, i clienti compatibili. Un altro programma che potrebbe essere eseguito ad intervalli regolari, aggiunge nuovi clienti al file per le selezioni successive. Nel corso della descri-

zione della manipolazione dei file presenteremo questi programmi.

Un file interno si può usare per memorizzare dati la cui dimensione è superiore a quella della memoria centrale del calcolatore, ma che non vengono più riutilizzati dopo il termine dell'esecuzione del programma. Si consideri il Programma 9, che produce una lista ordinata di concordanze di tutte le parole incontrate in un testo in input, accumulandole in un array. L'input di un testo di notevoli dimensioni potrebbe avere più parole di quante non ne possa contenere un array in qualsiasi memoria centrale. Un programma alternativo potrebbe accumulare le parole in un file su memoria di massa, ordinare il contenuto del file in maniera opportuna, e poi stamparlo ordinato, dando come risultato la lista finale delle concordanze. Il contenuto stesso del file perde d'importanza, una volta che l'operazione sia stata completata.

In PASCAL, un file è definito come una sequenza di lunghezza indefinita di valori dello stesso tipo. I file usati da un programma sono rappresentati da *variabili-file* di un corrispondente *tipo-file*. La sintassi che descrive il *tipo-file* è

tipo-file = **"file" "of"** *tipo-componente-file.*
tipo-componente-file = *tipo.*

Pertanto i valori componenti di un file possono essere valori non strutturati, o strutturati come array, record ed insiemi. Tuttavia, i valori componenti di un file non possono essere di tipo file, né avere componenti di tipo file.

Dato il tipo file

F = **file of** T

le relative variabili si dichiarano nel solito modo, cioè

f,g : *F*

f e *g* denotano quindi due file con componenti di tipo T.

Se si intende usare un file esterno, questo deve comparire nell'intestazione del programma ed in una dichiarazione di file nel blocco più esterno del programma. Abbiamo già visto che gli identificatori standard *input* ed *output* devono comparire nell'intestazione del programma, quando vengono usati, ma che essendo identificatori standard non occorre dichiararli anche come variabili. Quindi, per scrivere un programma che usa i file standard *input* ed *output* ed un altro file esterno chiamato *fileclienti,* il programma deve avere la forma

```
program appuntamenti (input, output, fileclienti);
.
var fileclienti : file of cliente;
:
```

In PASCAL un file si costruisce scrivendo o appendendo nuovi valori alla fine del file, che può inizialmente essere vuoto. Quando il file viene successivamente letto, la lettura delle sue componenti avviene esattamente nell'ordine usato in scrittura. La costruzione di un file impone, perciò, un ordinamento strettamente sequenziale delle sue componenti, alle quali si può avere accesso solo in quest'ordine. Un file PASCAL è detto quindi *file sequenziale.*

Durante l'accesso sequenziale delle componenti di un file, non si possono alternare

operazioni di lettura e di scrittura–ad ogni scansione del file si opera o solo in lettura, oppure solo in scrittura. Perciò, non si può modificare un singolo componente di un file, lasciando invariati gli altri, e scrivendo solo quel componente. Questa operazione si può realizzare soltanto facendo una copia dell'intero file, durante la quale viene fatta la modifica desiderata.

Un file PASCAL, normalmente, risiede su un dispositivo di memoria secondaria, ma ad ogni istante uno ed un solo componente è accessibile dal programma. La dichiarazione di un file *f* in un blocco introduce automaticamente una nuova variabile dello stesso tipo delle componenti del file. Questa variabile prende il nome di *buffer-file* e si denota con *f* ↑

buffer-file = variabile-file " ↑ ".
variabile-file = variabile.

In Fig. 12.1 è disegnato un file *f* ed il suo buffer-file associato.

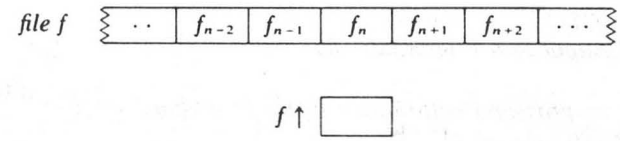


Fig. 12.1 Un file col suo buffer-file

Il buffer-file non è parte del file stesso, ma costituisce il mezzo tramite il quale vengono trasmessi i valori componenti tra il file ed il programma. Essendo una forma permessa di *variabile-componente,* un buffer-file si può usare in tutti i contesti in cui è consentito l'uso di una *variabile-componente.*

Consideriamo dapprima l'operazione di costruzione o scrittura di un file. Per preparare un file *f* per la scrittura, occorre un'istruzione

rewrite (f)

di chiamata alla procedura standard *rewrite.* Il suo effetto è di cancellare il contenuto del file *f,* cioè *f* diventa un file vuoto contenente zero componenti. Il successivo valore scritto in *f* diventa quindi il suo primo componente, ed i valori seguenti verranno scritti alla fine del file *f.* Il valore della funzione booleana standard

eof (f)

rimane vero per tutto il tempo in cui il file viene usato in scrittura, sebbene *eof* non venga mai controllata durante il processo di scrittura.

Per scrivere o appendere il valore *x* di tipo *T* alla fine del file *f,* il valore *x* deve prima essere assegnato alla variabile buffer-file *f* ↑ , cioè

f ↑ := *x*

Per appendere fisicamente il valore alla fine del file *f,* occorre un'istruzione

put (f)

di chiamata alla procedura standard *put.* Il suo effetto è di appendere il valore corrente

della variabile buffer-file $f\uparrow$ alla fine del file f; il valore del predicato *eof (f)* rimane vero, ed il valore di $f\uparrow$ diventa indefinito. Lo schema normale per la costruzione di un file di output f è riportato in (1).

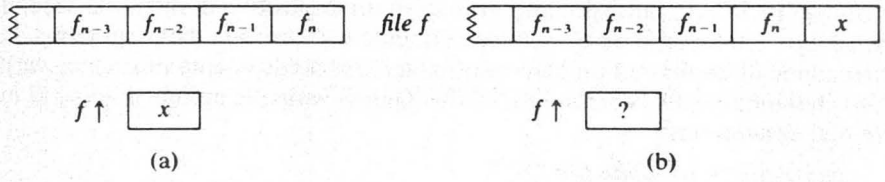


Fig. 12.2 Azione di *put (f)* : (a) prima, (b) dopo l'esecuzione

```
begin
   rewrite (f);
   while l'output non è terminato do
   begin
      f↑ := prossimo valore da mettere in output;
      put (f)
   end
end
```

(1)

Si consideri il programma basato sui file suggerito in precedenza per il servizio cuori solitari. Si richiede dapprima un programma per creare il file dei clienti già esistenti, dalle schede perforate precedentemente in uso. Un possibile soluzione è indicata in (2). Notare che il buffer del file *fileclienti* ↑ si può passare come parametro, al pari di ogni altra variabile di tipo *cliente,* alla procedura *leggischeda.*

```
program creaclienti (input, fileclienti);
type ...
     cliente = ... { come nel Programma 12 }
var fileclienti : file of cliente;
    .
    .
    .
  procedure leggischeda (var c :cliente);
  ... { come nel Programma 12 } ...
begin
    rewrite (fileclienti);
    repeat
        leggischeda (filecliente ↑);
        put (filecliente)
    until eof (input)
end.
```

(2)

La combinazione di azioni

```
f↑ := prossimo valore da mettere in output;
put (f)
```

è così frequente nella costruzione dei file, che per denotare tali azioni si usa una forma estesa della procedura standard PASCAL *write* (che è stata introdotta nel Cap. 5 per mandare valori al canale di output standard). La forma dell'istruzione diventa

write (f, x1, x2, x3, ..., xn)

I valori nella lista di espressioni *x1, x2, ...* devono essere compatibili per assegnazione con il tipo componente del file citato *f* (tranne che nel caso di un file testo, che è un caso particolare, discusso più avanti nel corso di questo capitolo).

La forma

write (f, x1, x2, ..., xn)

è equivalente a

```
begin
    write (f, x1);
    write (f, x2);
    .
    .
    write (f, xn)
end
```

e l'azione dell'istruzione procedura

write (f, x)

è definita in termini delle operazioni sul file

```
begin
    f↑ := x;
    put (f)
end
```

Introducendo un'altra variabile

prossimocliente : cliente

la parte istruzioni del nostro programma *creaclienti* si può scrivere come indicato in (3). In questo caso, l'uso della procedura *write* permette di evitare riferimenti espliciti nel programma alla variabile buffer del file.

```
begin
    rewrite (fileclienti);
    repeat
        leggischeda (prossimocliente);                    (3)
        write (fileclienti, prossimocliente)
    until eof (input)
end.
```

Consideriamo ora le operazioni necessarie per l'analisi del contenuto di un file costruito in precedenza. Per preparare un file *g* per la lettura, occorre un'istruzione

reset (g)

di chiamata alla procedura standard *reset*. Il suo effetto dipende dalla condizione che il file sia o no vuoto, cioè se contiene o no zero componenti. Se *g* non è vuoto il valore della prima componente viene assegnato alla variabile buffer del file $g\uparrow$ ed il valore del predicato *eof (g)* diventa *false*. Al contrario, se *g* è un file vuoto, *eof (g)* diventa *true* ed il valore di $g\uparrow$ è indefinito.

Il contenuto di $g\uparrow$ si può ora elaborare purché *g* non sia vuoto. Per leggere il valore componente successivo dal file *g*, si usa l'istruzione

get (g)

di chiamata alla procedura standard *get*. Il suo effetto è di assegnare il valore della successiva componente disponibile di *g* alla variabile buffer del file $g\uparrow$. Se tale componente non esiste, ovvero è stata raggiunta la fine del file *g*, *eof (g)* diventa *true*, ed il valore di $g\uparrow$ è indefinito.

Si noti che la condizione di fine-file non diventa vera finché non viene fatto un tentativo di *get* su un ipotizzato componente, dopo l'ultimo componente letto. Si può, però, compiere uno solo di questi tentativi–l'effetto di *get (g)* è definito solo quando *eof (g)* è *false* prima della sua esecuzione. Alcune implementazioni possono interrompere ogni ulteriore esecuzione di *get (g)*, segnalando un errore nel programma.

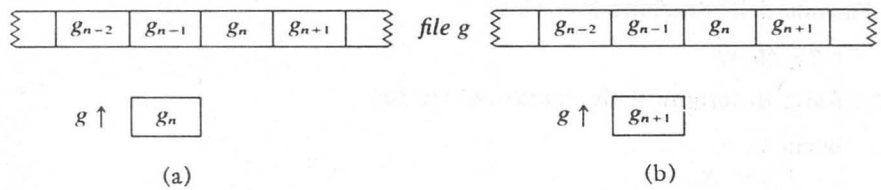


Fig. 12.3 Azione di *get (g)*: (a) prima, (b) dopo l'esecuzione

Poiché l'operazione di *reset* rende accessibile il primo componente di un file, l'operazione di lettura di un file, di solito, prende la forma mostrata in (4).

```
begin
    reset (g);
    while not eof (g) do
    begin
        elabora g↑ ;                                    (4)
        get (g)
    end
end
```

Siamo ora in grado di scrivere la nuova versione del programma per il servizio di elaborazione dati, presentata in (5), che genera una stampa dei clienti compatibili con ogni nuovo cliente, i cui dati sono forniti tramite una singola scheda di input.

```
program appuntamenti (input, output, fileclienti);
type ...
     cliente = ···
```

```
var fileclienti : file of cliente;
    nuovocliente : cliente;
  procedure leggischeda (var c : cliente);
  ... { come prima } ...;
  function compatibile (c1, c2 : cliente) : boolean;
  ... { come prima } ...;
  begin { programma principale }                    (5)
      leggischeda (nuovocliente);
      reset (fileclienti);
      while not eof (fileclienti) do
      begin
          if compatibile (fileclienti ↑ , nuovocliente)
            then writeln (filecliente ↑.identita);
          get (fileclienti)
      end
end.
```

La sequenza di azioni

```
assegna il valore di g ↑ ad una variabile;
get (g)
```

capita spesso quando si elabora un file di input; per denotare queste azioni si usa una forma particolare della procedura standard *read*. La forma dell'istruzione *read*, in questi casi, è

```
read (g, v1, v2, ..., vn)
```

in cui *v1*, *v2*, ... sono variabili.

La forma generale

```
read (g, v1, v2, ..., vn)
```

è equivalente a

```
begin
    read (g, v1);
    read (g, v2);
    .
    .
    .
    read (g, vn)
end
```

in cui l'azione dell'istruzione procedura

```
read (g, v)
```

è definita (tranne nel caso di un file testo–vedi più avanti in questo capitolo) come

```
begin
    v := g ↑ ;
    get (g)
end
```

Pertanto il valore del buffer del file $g\uparrow$ deve essere compatibile per assegnazione con il tipo della variabile *v*, quando viene eseguita *read (g, v)*. Introducendo la variabile aggiuntiva

```
prossimocliente : cliente
```

siamo in grado di riscrivere la parte istruzioni del nostro programma *dati* come riportato in (6).

```
begin
    leggischeda (nuovocliente);
    reset (filecliente);
    while not eof (filecliente) do                                    (6)
    begin
        read (fileclienti, prossimocliente);
        if compatibile (prossimocliente, nuovocliente)
          then writeln (prossimocliente.identita)
    end
end.
```

In un file PASCAL i componenti si possono scrivere solo dopo aver predisposto il file con un'operazione *rewrite*. È perciò impossibile come ultima operazione del programma precedente aggiungere nuovi clienti alla fine del file dei clienti già esistenti. Dobbiamo, invece, scrivere un altro programma, presentato in (7), che crei una copia del vecchio file clienti in un nuovo file, leggendo e riscrivendo componente per componente, e poi appenda i nuovi clienti alla fine del nuovo file.

```
program aggiorna (input, vecchiofile, nuovofile);
type ...
     cliente = ···
var vecchiofile, nuovofile : file of cliente;
    prossimocliente : cliente;
  procedure leggischeda (var c : cliente);
  ... { come prima } ...;
  begin
      reset (vecchiofile);
      rewrite (nuovofile);
      { prima copia vecchiofile in nuovofile }
      while not eof (vecchiofile) do
      begin
          read (vecchiofile, prossimocliente);                        (7)
          write (nuovofile, prossimocliente)
      end;
      { poi appendi i nuovi clienti al nuovo file }
      while not eof (input) do
      begin
          leggischeda (prossimocliente);
          write (nuovofile, prossimocliente)
      end
end.
```

Se si passa ad una procedura un file come parametro per valore, all'ingresso nella procedura viene creata una copia del contenuto dell'intero file. Ovviamente, per un file di dimensioni non trascurabili, questa operazione richiede una notevole quantità di tempo. Per questi motivi il PASCAL stabilisce che tutti i parametri di tipo file debbano essere passati come parametri *per variabile.* La funzione (8) riceve come parametro un file di numeri reali (di un tipo *filereale)* e restituisce, come risultato, la somma dei numeri reali contenuti nel file.

```
function sommafile (var f : filereale) : real;
var somma, x : real;
begin
    somma := 0;
    reset (f);
    while not eof (f) do                                    (8)
    begin
        read (f, x);
        somma := somma + x
    end;
    sommafile := somma
end
```

PROGRAMMA 14 (Aggiornamento di un file scorta)

Una ditta di distribuzione all'ingrosso mantiene un file, detto file scorta, in cui c'è un record per ciascun tipo di merce presente nel suo magazzino. Ogni record contiene le seguenti informazioni:

```
scortaarticolo = record
                     numeroarticolo : 0..99999;
                     discorta, scortanormale, ordinata,
                       limiteordinazione : 0..maxint
                 end
```

Quando la quantità di scorta, più la quantità ordinata di un articolo, è al di sotto del limite per l'ordinazione, deve essere emesso un ordine per un'ulteriore approvvigionamento dell'articolo al fine di ristabilirne la disponibilità al valore di *scorta normale.* I record nel file scorta risultano ordinati in ordine crescente di *numeroarticolo.*

Il file scorta deve essere quotidianamente aggiornato, cioè occorre produrre un nuovo file scorta che tenga conto delle transazioni (consegne e spedizioni) che sono avvenute dall'ultimo aggiornamento. Queste transazioni sono accumulate su un altro file, in cui ogni transazione rappresenta una consegna o una spedizione di articoli di un certo tipo, come mostrato in (9). Anche i record del file delle transazioni sono ordinati in ordine crescente di numero di articolo. (Il Programma 15 mostra come ottenere un file ordinato da dati raccolti giornalmente in modo disordinato).

```
transazione = record
                  numeroarticolo : 0..99999;
                  tipotransazione : (consegna, spedizione);      (9)
```

```
        quantita : 1..maxint
    end
```

Si richiede un programma per aggiornare il file scorta, in base al contenuto del file transazioni, producendo un nuovo file scorta. Gli articoli che si trovano al di sotto del loro limite per l'ordinazione, devono essere stampati in un elenco ordinato.

Poiché entrambi i file sono ordinati, il processo richiesto deve considerare, uno alla volta, gli articoli del vecchio file scorta, facendo le modifiche indicate dai successivi record del file delle transazioni. Nel progettare il programma bisogna ricordare che

(a) molti dei record del file scorta potrebbero non essere coinvolti in transazioni;
(b) lo stesso record del file scorta potrebbe essere coinvolto in più transazioni;
(c) una transazione potrebbe coinvolgere un articolo inesistente.

Assumendo che i file coinvolti nell'operazione si chiamino *vecchiascorta, nuovascorta* e *transazioni,* lo schema base del programma potrebbe essere quello riportato in (10).

```
reset (vecchiascorta);
reset (transazioni);
rewrite (nuovascorta);
while not eof (vecchiascorta) do
begin
    articolocorrente := vecchiascorta ↑.numeroarticolo;
    nuovascorta ↑ := vecchiascorta ↑ ;
    rifiuta le transazioni con numeroarticolo < articolocorrente;
    modifica nuovascorta ↑ in base alle transazioni dell'articolocorrente;      (10)
    manda l'ordine se necessario;
    put (nuovascorta);
    get (vecchiascorta)
end;
rifiuta tutte le restanti transazioni
```

Il rifiuto delle transazioni aventi un *numeroarticolo* minore dell'*articolocorrente* si può esprimere per mezzo di un ciclo-while. Dobbiamo comunque controllare, ad ogni lettura del file transazioni, che questo non sia terminato. Il ciclo richiesto potrebbe, perciò, essere quello riportato in (11) (in esso viene introdotta la variabile booleana *minnumeroarticolo,* per evitare i problemi relativi alla valutazione delle espressioni booleane, discussi nel Cap. 4).

```
minnumeroarticolo := true;
while not eof (transazioni) and minnumeroarticolo do
if transazioni ↑.numeroarticolo < articolocorrente;
  then begin
          rifiuta la transazione;                                    (11)
          get (transazioni)
       end
  else minnumeroarticolo := false
```

Analogamente, si può scrivere il frammento di programma (12) per il passo *modifica*

nuovascorta ↑ *in base alle transazioni dell'articolocorrente.*

```
stessonumeroarticolo := true;
while not eof (transazioni) and stessonumeroarticolo do
if transazioni ↑.numeroarticolo = articolocorrente
  then begin
         modifica nuovascorta ↑ in base a transazioni ↑;          (12)
         get (transazioni)
       end
  else stessonumeroarticolo := false
```

Il passo *modifica nuovascorta* ↑ *in base a transazioni* ↑ si può scrivere facilmente come in (13).

```
with nuovascorta ↑.transazioni ↑ do
case tipotransazione of
  consegna : begin
               discorta := discorta + quantita;                   (13)
               ordinati := ordinati - quantita
             end;
  spedizione : discorta := discorta - quantita
end
```

Il passo *manda l'ordine se necessario* è a sua volta facile da scrivere come indicato in (14).

```
with nuovascorta ↑ do
if discorta + ordinati < limiteordinazione
  then begin
         writeln ('ordine', scortanormale - discorta - ordinati,
                  'dell''articolo', numeroarticolo : 6);          (14)
         ordinati := limiteordinazione - discorta
       end
```

Il passo finale del programma, *rifiuta tutte le restanti transazioni,* si può esprimere con il semplice ciclo-while mostrato in (15).

```
while not eof (transazioni) do
begin
  rifiuta la transazione;
  get (transazioni)                                               (15)
end
```

Nel nostro semplice programma, esprimeremo il passo *rifiuta la transazione* con una chiamata alla procedura (16), che stampa semplicemente un messaggio di rifiuto contenente il numero dell'articolo in questione.

```
procedure rifiutatransazione;
begin
  writeln ('rifiutata una transazione per l''articolo',
           transazione ↑.numeroarticolo : 6)                      (16)
end
```

In un sistema reale, ogni record transazione dovrebbe contenere informazioni sulla data e sul luogo della transazione, in modo da poter risalire alla fonte delle transazioni rifiutate. In un sistema reale potrebbero essere richiesti altri controlli sulla validità dei dati relativi alle transazioni complicando le relative manipolazioni. Ad ogni modo, la struttura del programma, e le sue caratteristiche riguardanti la manipolazione dei file, non dovrebbero risultare molto diverse da quelle da noi sviluppate.

Il Listato 14 mostra il programma finale—l'output dell'esecuzione del programma riporta il contenuto di un file scorta campione prima dell'aggiornamento, le transazioni usate per aggiornarlo, i messaggi di errore, gli ordini prodotti durante l'aggiornamento, ed infine il contenuto del file aggiornato.

LISTATO 14

```
PROGRAM MAGAZZINO(TRANSAZIONI,VECCHIASCORTA,NUOVASCORTA,OUTPUT);

TYPE  SCORTAARTICOLO = RECORD
                         NUMEROARTICOLO : 0..99999;
                         DISCORTA,SCORTANORMALE,ORDINATI,
                         LIMITEORDINAZIONE : 0..MAXINT
                       END;
      FILESCORTA     = FILE OF SCORTAARTICOLO;
      TRANSAZIONE    = RECORD
                         NUMEROARTICOLO   : 0..99999;
                         TIPOTRANSAZIONE  : (CONSEGNA,SPEDIZIONE);
                         QUANTITA         : 1..MAXINT
                       END;
      INTESTAZIONE = PACKED ARRAY [1..9] OF CHAR;

VAR   TRANSAZIONI                 : FILE OF TRANSAZIONE;
      VECCHIASCORTA,NUOVASCORTA   : FILESCORTA;
      ARTICOLOCORRENTE            : 1..99999;
      MINNUMERO,STESSOARTICOLO    : BOOLEAN;

      PROCEDURE STAMPASCORTA(NOME : INTESTAZIONE;
                             VAR F : FILESCORTA);
      BEGIN
         RESET(F);
         WRITELN;
         WRITELN;
         WRITELN('** ',NOME,' **');
         WRITE('NUMERO EL. ');
         WRITELN('DISCORTA ORDINATI SCORTANORMALE LIMITEORDINAZIONE');
         WHILE NOT EOF(F) DO
            WITH F^ DO
            BEGIN
               WRITELN(NUMEROARTICOLO, DISCORTA:9, ORDINATI:9,
                       SCORTANORMALE:12, LIMITEORDINAZIONE:15);
               GET(F)
            END
      END; (* STAMPASCORTA *)
```

```
2 ──►PROCEDURE STAMPATRANSAZIONI;
     BEGIN
        RESET(TRANSAZIONI); Prepara alla lettura
        WRITELN;
        WRITELN;
        WRITELN('** TRANSAZIONI **');
        WHILE NOT EOF(TRANSAZIONI) DO
           WITH TRANSAZIONI^ DO
           BEGIN
              IF TIPOTRANSAZIONE = CONSEGNA
                 THEN WRITE('CONSEGNA  ')
                 ELSE WRITE('SPEDIZIONE');
              WRITELN(NUMEROARTICOLO,QUANTITA);
              GET(TRANSAZIONI)
           END
     END; (* STAMPATRANSAZIONI *)

3 ──►PROCEDURE RIFIUTATRANSAZIONE;
     BEGIN
        WRITELN('TRANSAZIONE RIFIUTATA PER NUMERO ',
                TRANSAZIONI^.NUMEROARTICOLO:6)
     END; (* RIFIUTATRANSAZIONE *)


BEGIN (* PROGRAMMA PRINCIPALE *)
   STAMPASCORTA('VECCHIA S',VECCHIASCORTA);
   STAMPATRANSAZIONI;
   RESET(VECCHIASCORTA); RESET(TRANSAZIONI);
   REWRITE(NUOVASCORTA);
   WHILE NOT EOF(VECCHIASCORTA) DO
   BEGIN
      ARTICOLOCORRENTE:= VECCHIASCORTA^.NUMEROARTICOLO;
      NUOVASCORTA^:= VECCHIASCORTA^;
      (* RESPINGI TRANSAZIONI CON
         NUMEROARTICOLO < ARTICOLOCORRENTE *)
      MINNUMERO:= TRUE;
      WHILE NOT EOF(TRANSAZIONI) AND MINNUMERO DO
         IF TRANSAZIONI^.NUMEROARTICOLO < ARTICOLOCORRENTE
            THEN BEGIN
                    RIFIUTATRANSAZIONE; GET(TRANSAZIONI)
                 END
            ELSE MINNUMERO:= FALSE;
      (* AGGIORNA NUOVASCORTA^ IN BASE ALLE TRANSAZIONI
         DELL' ARTICOLO CORRENTE                    *)
      STESSOARTICOLO:= TRUE;
      WHILE NOT EOF(TRANSAZIONI) AND STESSOARTICOLO DO
         IF TRANSAZIONI^.NUMEROARTICOLO = ARTICOLOCORRENTE
            THEN BEGIN
                    WITH NUOVASCORTA^, TRANSAZIONI^ DO
                       CASE TIPOTRANSAZIONE OF
                          CONSEGNA   :
                             BEGIN
                                DISCORTA:= DISCORTA + QUANTITA;
                                ORDINATI:= ORDINATI - QUANTITA
                             END;
```

```
                        SPEDIZIONE :
                          DISCORTA:= DISCORTA - QUANTITA
                    END;
                  GET(TRANSAZIONI)
               END
          ELSE STESSOARTICOLO:= FALSE;
          (* AUMENTA L' ORDINAZIONE SE NECESSARIO *)
          WITH NUOVASCORTA^ DO
          IF DISCORTA + ORDINATI < LIMITEORDINAZIONE
             THEN BEGIN
                    WRITELN('ORDINARE',
                            SCORTANORMALE-ORDINATI-DISCORTA,
                            ' DEL NUMERO',NUMEROARTICOLO:6);
                    ORDINATI:= SCORTANORMALE - DISCORTA
                  END;
          PUT(NUOVASCORTA); GET(VECCHIASCORTA)
END;
   (* RESPINGI TUTTE LE RIMANENTI TRANSAZIONI *)
   WHILE NOT EOF(TRANSAZIONI) DO
   BEGIN
      RIFIUTATRANSAZIONE;
      GET(TRANSAZIONI)
   END;
   STAMPASCORTA('NUOVA S',NUOVASCORTA)
END.
```

```
** VECCHIA S **
NUMERO EL. DISCORTA ORDINATI SCORTANORMALE LIMITEORDINAZIONE
     11081     5450     7000            0              5000
     11202     5430     8000         2500              6000
     23934     6230     8000         2500              6000
     28454     1270     5000.        3500              4000
     36666     9090     9500            0              7000
     37775     1000     2000         1000              1500
     39399     4620     6000            0              4500
     42000     3550     5000         1400              4000
     42111     4460     8000         3500              6000
     42281     3750     6000         1500              4500
     43327     8260     9000            0              7000
     53553     8000     9000            0              7000
     55376     2800     5000         2000              4000
     57862     1370     3000         1150              2000
     59097     3540     5000         1460              4000


** TRANSAZIONI **
CONSEGNA       10754       1000
SPEDIZIONE     11081        350
SPEDIZIONE     11081       1240
SPEDIZIONE     11081       2000
SPEDIZIONE     11202       1500
SPEDIZIONE     23934       2000
CONSEGNA       28454       3500
SPEDIZIONE     28454        500
```

```
CONSEGNA        29334        500
SPEDIZIONE      36666       2000
SPEDIZIONE      37775         50
SPEDIZIONE      42000        500
SPEDIZIONE      42000        300
SPEDIZIONE      42111        500
CONSEGNA        42111       3500
SPEDIZIONE      42281        450
SPEDIZIONE      55376        520
SPEDIZIONE      55376        470
CONSEGNA        57862       1150
SPEDIZIONE      59907        540
TRANSAZIONE RIFIUTATA PER NUMERO  10754
TRANSAZIONE RIFIUTATA PER NUMERO  29334
TRANSAZIONE RIFIUTATA PER NUMERO  59907


** NUOVA S    **
NUMERO EL. DISCORTA ORDINATI SCORTANORMALE LIMITEORDINAZIONE
     11081     1860     7000           0            5000
     11202     3930     8000        2500            6000
     23934     4230     8000        2500            6000
     28454     4270     1500        3500            4000
     36666     7080     9500           0            7000
     37775      950     2000        1000            1500
     39399     4620     6000           0            4500
     42000     2750     5000        1400            4000
     42111     7460     4500        3500            6000
     42281     3300     6000        1500            4500
     43327     8260     9000           0            7000
     53553     8000     9000           0            7000
     55376     1810     5000        2000            4000
     57862     2520     1850        1150            2000
     59097     3540     5000        1460            4000
```

PROGRAMMA 15 (Ordinamento di un file)

Nel Programma 14 abbiamo riscontrato la necessità di produrre un file transazioni le cui componenti fossero ordinate in ordine crescente di numero dell'articolo, a partire da un file di transazioni inizialmente non ordinato. L'operazione di ordinamento di un file è richiesta comunemente nella manipolazione dei file, e può essere considerata da un punto di vista più generale.

Dato un file di record, la cui definizione è riportata in (17), si richiede un programma che ordini i record in ordine crescente rispetto al valore del campo chiave di ciascun record.

```
type elemento = record
                    chiave : tipochiave;
                    resto : tipoqualsiasi
                end;
var filedati : file of elemento;
```
(17)

Useremo, per realizzare il processo, un metodo noto col nome di *ordinamento per fusione naturale (natural merge sort)*. Per descrivere come opera questo metodo definiamo alcuni termini.

Data una sequenza $k_1, ..., k_n$ di valori della chiave, una sottosequenza ordinata crescente, o *segmento*, $k_i, ..., k_l$ è tale che

$$k_{i-1} > k_i$$
$$k_j \leqq k_{j+1} \quad \text{per ogni } j = i, ..., l-1$$
$$k_l > k_{l+1}$$

La *fusione* è la combinazione di due o più segmenti in un singolo segmento. I segmenti godono della proprietà ovvia che, se 2 sequenze di N segmenti vengono fuse, il risultato è una singola sequenza di N segmenti. Perciò, durante l'ordinamento per fusione, il numero totale di segmenti viene dimezzato ad ogni *passo*.

C	82'	48'	14	15	84'	25	77'	13	72'	4	51'	19	27	43	57'	53'	
distribuzione																	
A	82'	14	15	84'	13	72'	19	27	43	57'							
B	48'	25	77'	4	51	53'											
fusione																	
C	48	82'	14	15	25	77	84'	4	13	51	53	72'	19	27	43	57'	
distribuzione																	
A	48	82'	4	13	51	53	72'										
B	14	15	25	77	84'	19	27	43	57'								
fusione																	
C	14	15	25	48	77	82	84'	4	13	19	27	43	51	53	57	72'	
distribuzione																	
A	14	15	25	48	77	82	84'										
B	4	13	19	27	43	51	53	57	72'								
fusione																	
C	4	13	14	15	19	25	27	43	48	51	53	57	77	72	82	84'	

Fig. 12.4 Effetto dell'ordinamento per fusione naturale

Supponiamo che la sequenza iniziale di record non ordinati sia in un file *C*, che l'output ordinato debba apparire su *C*, e che *A* e *B* siano due file ausiliari, da usare per l'ordinamento. In un ambiente di elaborazione dati reale, i dati effettivi vengono copiati dal file originale *filedati* in *C* prima dell'ordinamento e, al termine dell'operazione, i dati in *C* vengono ricopiati nel *filedati*. Questo viene fatto per ragioni di sicurezza, nel caso che accada un qualsiasi evento imprevisto, durante l'esecuzione del programma di ordinamento, che potrebbe causare la perdita dei dati originali.

Ogni passo dell'ordinamento è costituito da una *fase distributiva*, in cui vengono distribuiti alternativamente in *A* ed in *B* i segmenti, e da una *fase di fusione*, in cui i segmenti che si trovano in *A* ed in *B* vengono fusi in *C*. Per esempio, la Fig. 12.4 mostra i valori della chiave dei record di un file *C* nella loro posizione iniziale, *A* e *B* dopo ogni fase distributiva, e *C* dopo ogni fase di fusione. (L'elemento finale di un segmento è marcato con un simbolo primo).

Si osservi che in questo esempio occorrono tre passi, e che l'ordinamento termina quando c'è un solo segmento in *C* dopo una fase di fusione.

Introducendo la definizione globale

```
type tipofile = file of elemento
```

siamo in grado di formulare la procedura (18), per realizzare l'ordinamento per fusione naturale di *C*.

```
procedure ordinamentoperfusionenaturale (var C : tipofile);
var numerosegmenti : integer;
    A, B : tipofile;
begin
    repeat
        reset (C);
        rewrite (A);
        rewrite (B);
        distribuisci;
        reset (A);
        reset (B);
        rewrite (C);
        numerosegmenti := 0;
        fusione
    until numerosegnenti = 1
end
```

(18)

Procediamo ora con il raffinamento delle fasi *distribuisci* e *fusione*, come procedure locali di *ordinamentoperfusionenaturale*. L'azione realizzata da *distribuisci*, riportata in (19), è di copiare i segmenti da *C* ad *A* e *B* alternativamente, finché *C* non è stato letto completamente (assumiano che *C* non sia vuoto).

```
procedure distribuisci;
begin
    repeat
        copiasegmento (C, A);
```

(19)

```
            if not eof (C)
              then copiasegmento (C, B)
        until eof (C)
    end
```

La procedura *copiasegmento*, sviluppata nel seguito, copia un segmento di record dal file che ha come primo parametro, al file che ha come secondo parametro.

Sebbene si possa pensare che il numero dei segmenti distribuiti su *A* e su *B* differisca al più di 1, in realtà questo non è sempre vero, perché segmenti consecutivi distribuiti su un file possono in effetti combinarsi, formando un singolo segmento su quel file (vedi l'esempio in cui 4, 51 e 53 sono distribuiti su *B* come segmenti distinti, ma poi vengono a formare su *B* un singolo segmento). Quindi la fase di fusione (20) combina i segmenti di *A* e di *B*, fino a che non raggiunge la fine di uno dei file, dopo di che la coda dell'altro file deve essere copiata su *C*.

```
procedure fusione;
begin
    while not (eof (A) or eof (B)) do
    begin
        fusionesegmentidaAeB;
        numerosegmenti := numerosegmenti + 1
    end;
    while not eof (A) do                                    (20)
    begin
        copiasegmento (A, C);
        numerosegmenti := numerosegmenti + 1
    end;
    while not eof (B) do
    begin
        copiasegmento (B, C);
        numerosegmenti := numerosegmenti + 1
    end
end
```

Le azioni della procedura *copiasegmento* (21) sono espresse in termini di una procedura *copia*, che trasferisce un valore di tipo *elemento* dal file che costituisce il primo parametro al file che costituisce il secondo parametro, e determina il momento in cui termina un segmento. Perció, viene introdotta una variabile *finesegmento* che indica se un segmento è terminato o meno. Il valore di questa variabile è determinato da ogni chiamata di *copia*. Notare che entrambi i parametri di tipo file vanno specificati come parametri per variabile.

```
procedure copiasegmento (var sorgente, destinazione : tipofile);
begin
    repeat
        copia (sorgente, destinazione)                      (21)
    until finesegmento
end
```

La procedura *fusionesegmentidaAeB* produce in *C* un singolo segmento, per fusione di un segmento di *A* ed uno di *B*. Essa opera confrontando il campo chiave dei record nei corrispondenti segmenti di *A* e *B* e chiama *copia* per trasferire i record selezionati in *C*. Questo processo termina con l'esaurimento di uno dei due segmenti, il che provoca la copia della coda dell'altro in *C*. Se *tipochiave* è un tipo non strutturato, o un tipo stringa (come nel nostro caso), l'operazione di confronto si esprime banalmente con l'operatore <; per un tipo strutturato, invece, si hanno diversi campi chiave, e l'operazione di confronto va espressa in modo più complicato.

```
procedure fusionesegmentidaAeB;
begin
    repeat
        if A ↑.chiave < B ↑.chiave
          then begin
                 copia (A, C);
                 if finesegmento then copiasegmento (B, C)
               end
          else begin
                 copia (B, C);
                 if finesegmento then copiasegmento (A, C)
               end
    until finesegmento
end
```
(22)

La procedura *copia* (23) si scrive facilmente in termini delle operazioni base sui file del PASCAL. Per determinare la fine di un segmento, occorre conservare il valore della chiave dell'ultimo record copiato da un file, per confrontarlo con il suo successore

```
procedura copia (var sorgente, destinazione : tipofile);
var elementocopiato : elemento;
begin
    elementocopiato := sorgente ↑ ;
    get (sorgente);
    { read (sorgente, elementocopiato) }
    destinazione ↑ := elementocopiato;
    put (destinazione);
    { write (destinazione, elementocopiato) }
    if eof (sorgente)
      then finesegmento := true
      else finesegmento := elementocopiato.chiave > sorgente ↑.chiave
end
```
(23)

La procedura di ordinamento per fusione naturale risulta quindi completamente sviluppata e può essere incorporata nel programma (24), che copia il file da ordinare nel file *C*, chiama *ordinamentoperfusionenaturale*, ed alla fine copia il file ordinato *C* nel file originale. La procedura *copiafile* trasferisce i record tra i file che costituiscono i suoi due parametri, segnalando il valore della chiave per ogni record trasferito.

```
begin
   copiafile (filedati, C);
   ordinamentoperfusionenaturale;                    (24)
   copiafile (C, filedati)
end
```

Il Listato 15 mostra il programma completo con la lista dei campi chiave generata dalla procedura *copiafile,* per verificare la riuscita dell'ordinamento.

LISTATO 15

```
PROGRAM ORDINAMENTO(FILEDATI,OUTPUT);

TYPE TIPOCHIAVE     = 0..99999;
     TIPOQUALSIASI  = RECORD
                         TRANSAZIONE : (CONSEGNA,SPEDIZIONE);
                         AMMONTARE   : 1..MAXINT
                      END;
     ELEMENTO       = RECORD
                         CHIAVE : TIPOCHIAVE;
                         RESTO  : TIPOQUALSIASI
                      END;
     TIPOFILE       = FILE OF ELEMENTO;

VAR  FILEDATI : TIPOFILE;
     C        : TIPOFILE;

     PROCEDURE ORDINAMENTOPERFUSIONENATURALE(VAR C : TIPOFILE);
     VAR  NUMEROSEGMENTI : 0..MAXINT;
          A, B           : TIPOFILE;
          FINESEGMENTO   : BOOLEAN;

          PROCEDURE COPIA (VAR SORGENTE,DESTINAZIONE : TIPOFILE);
          VAR  ELEMENTOCOPIATO : ELEMENTO;
          BEGIN
             ELEMENTOCOPIATO:= SORGENTE^;
             GET(SORGENTE);
             DESTINAZIONE^:= ELEMENTOCOPIATO;
             PUT(DESTINAZIONE);
             IF EOF(SORGENTE)
                THEN FINESEGMENTO:= TRUE
                ELSE FINESEGMENTO:= ELEMENTOCOPIATO.CHIAVE >
                                    SORGENTE^.CHIAVE
          END; (* COPIA *)

          PROCEDURE COPIASEGMENTO(VAR SORGENTE,DESTINAZIONE :
                                                  TIPOFILE);
          BEGIN
             REPEAT
                COPIA(SORGENTE,DESTINAZIONE)
             UNTIL FINESEGMENTO
          END; (* COPIASEGMENTO *)
```

```
      PROCEDURE DISTRIBUISCI;
      BEGIN
         REPEAT
            COPIASEGMENTO(C,A);
            IF NOT EOF(C)
               THEN COPIASEGMENTO(C,B)
         UNTIL EOF(C)
      END; (* DISTRIBUISCI *)

      PROCEDURE FUSIONE;

         PROCEDURE FUSIONESEGMENTIAEB;
         BEGIN
            REPEAT
               IF A^.CHIAVE < B^.CHIAVE
                  THEN BEGIN
                          COPIA(A,C);
                          IF FINESEGMENTO
                             THEN COPIASEGMENTO(B,C)
                       END
                  ELSE BEGIN
                          COPIA(B,C);
                          IF FINESEGMENTO
                             THEN COPIASEGMENTO(A,C)
                       END
            UNTIL FINESEGMENTO
         END; (* FUSIONESEGMENTIAEB *)

      BEGIN
         WHILE NOT (EOF(A) OR EOF(B)) DO
         BEGIN
            FUSIONESEGMENTIAEB;
            NUMEROSEGMENTI:= NUMEROSEGMENTI + 1
         END;
         WHILE NOT EOF(A) DO
         BEGIN
            COPIASEGMENTO(A,C);
            NUMEROSEGMENTI:= NUMEROSEGMENTI + 1
         END;
         WHILE NOT EOF(B) DO
         BEGIN
            COPIASEGMENTO(B,C);
            NUMEROSEGMENTI:= NUMEROSEGMENTI + 1
         END
      END; (* FUSIONE *)

   BEGIN (* ORDINAMENTOPERFUSIONENATURALE *)
      REPEAT
         RESET(C);
         REWRITE(A);
         REWRITE(B);
         DISTRIBUISCI;
         RESET(A);
         RESET(B);
         REWRITE(C);
```

```
        NUMEROSEGMENTI:= 0;
        FUSIONE
    UNTIL NUMEROSEGMENTI = 1
END; (* ORDINAMENTOPERFUSIONE NATURALE *)

    PROCEDURE COPIAFILE (VAR F,G : TIPOFILE);
    BEGIN
       RESET(F);
       REWRITE(G);
       WHILE NOT EOF(F) DO
       BEGIN
          WRITELN(F^.CHIAVE);
          G^:= F^;
          PUT(G);
          GET(F)
       END
    END; (* COPIAFILE *)


BEGIN (* PROGRAMMA PRINCIPALE *)
   WRITELN('** CHIAVI NON ORDINATE **');
   COPIAFILE(FILEDATI,C);
   ORDINAMENTOPERFUSIONENATURALE(C);
   WRITELN;
   WRITELN;
   WRITELN('** CHIAVI ORDINATE **');
   COPIAFILE(C,FILEDATI)
END.
```

```
** CHIAVI NON ORDINATE **
 57862
 29334
 42111
 42000
 36666
 11202
 11081
 55376
 28454
 23934
 42111
 11081
 11081
 55376
 28454
 42281
 42000
 37775
 10754
 59907
```

```
** CHIAVI ORDINATE **
 10754
 11081
 11081
 11081
 11202
 23934
 28454
 28454
 29334
 36666
 37775
 42000
 42000
 42111
 42111
 42281
 55376
 55376
 57862
 59907
```

12.2. FILE TESTO

Per agevolare la manipolazione di testi di caratteri, il PASCAL fornisce il tipo file standard *text*. I file che vengono dichiarati di questo tipo si chiamano *text file*.

Un file testo è formato da una sequenza di linee–ciascuna delle quali contenente valori di tipo *char*–separate da speciali caratteri di controllo delle linee. Ogni linea contiene 0 o più valori di tipo carattere. I canali standard di *input* e di *output*, descritti nel Cap. 5, sono strutturati in questo modo e gli identificatori *input* ed *output*, usati per denotare questi canali, sono in realtà predichiarati come identificatori di file testo in tutte le implementazioni PASCAL, nel seguente modo

input, output : text

Pertanto le procedure *write, writeln, read, readln, page* e le funzioni *eoln* (fine della linea) e *eof* (fine del file), usate nel Cap. 5 con i canali *input* ed *output*, sono in realtà casi particolari d'uso di strumenti standard più generali per manipolare i file testo. L'insieme completo delle procedure e delle funzioni applicabili ai file testo è riassunto nel seguito, con opportuni riferimenti al Cap. 5.

reset e rewrite

Come gli altri file, i file testo devono essere preparati per la scrittura e per la lettura, usando le procedure standard *reset* e *rewrite*. Ad ogni modo, le istruzioni di procedura

reset (input)
rewrite (output)

si assumono eseguite automaticamente all'inizio dei programmi che usano i file stan-

dard, e non occorre quindi scriverle in tali casi. L'effetto di un'operazione *reset,* o *rewrite,* sui file standard *input* ed *output,* in qualsiasi punto del programma, è definito dall'implementazione, e potrebbe essere un errore.

eof, get e put

La funzione standard *eof* e le procedure standard *get* e *put* si possono applicare ad un file testo, come ad ogni altro file. Allo stesso modo, un file testo *f* ha associato un buffer file *f*↑, che è una variabile che rappresenta il carattere corrente accessibile del file. Il buffer di un file testo differisce da quello di un file non di tipo testo solo per quanto verrà spiegato nel seguito a proposito di *eoln.*

Come già osservato nel Cap. 5, l'uso del predicato *eof* senza un parametro file fa riferimento al file standard *input.* Quindi

eof e *eof (input)*

sono equivalenti.

L'esistenza del buffer per il file standard di input consente maggiore flessibilità nell'input di informazioni di tipo carattere, di quanta ne consente l'uso della sola procedura standard *read.* Si può ispezionare il successivo carattere disponibile del canale di input, dentro *input*↑, prima che sia copiato in una variabile *c* e prima dell'avanzamento nel canale di input, come accade con l'istruzione *read (c).*

eoln

Quando il carattere corrente di un file testo è un carattere di fine linea, la funzione booleana standard

eoln (f)

restituisce il valore *true,* ed il valore di tipo carattere del buffer del file *f*↑ è uno spazio. In tutti gli altri casi, il risultato di *eoln (f)* è *false.*

Come già osservato nel Cap. 5, l'uso di *eoln* senza un parametro file fa riferimento al file standard *input.* Pertanto

eoln e *eoln (input)*

sono equivalenti.

read e readln

Le istruzioni procedura standard

read (f,v1,v2,...,vn)
readln (f,v1,v2,...,vn)
readln (f)

si possono usare con qualsiasi file testo *f.*

La forma

read (f,v1,v2,...vn)

è equivalente a

```
begin
    read (f,v1);
    read (f,v2);
    :
    :
    read (f,vn)
end
```

La forma

readln (f)

è definita dalla sequenza di operazioni base sui file testo (25).

```
begin
    while not eoln (f) do get (f);
    get (f)                                    (25)
end
```

In tal modo, $f\uparrow$ viene posizionato sul primo carattere della linea successiva.

Per un file testo l'effetto di

read (f,v)

dipende dal tipo della variabile *v* nel seguente modo:

(a) se *v* è di tipo *char* l'effetto è definito, come per gli altri file non di tipo testo, da

```
v := f↑ ;
get (f)
```

(b) se *v* è di tipo *real* o *integer,* o un suo sottointervallo, l'effetto è quello illustrato nel Cap. 5 per il file standard di input. Viene considerata una sequenza di caratteri del file *f* tale da denotare un reale o un intero, eventualmente con segno (come definito nel Cap. 3), preceduto da un numero qualsiasi di spazi (o di caratteri di fine linea). Il valore denotato dalla sequenza viene assegnato a *v*, ed il primo carattere dopo la sequenza viene lasciato come $f\uparrow$. Se i caratteri letti non denotano un carattere valido, oppure nel caso in cui la condizione di fine del file, *eof (f)*, diventa vera prima della lettura di una sequenza di caratteri significativa, l'effetto di *read (f,v)* è un errore del programma.

Come già osservato nel Cap. 5, l'uso di *read* e *readln,* senza il parametro file *f,* implica il riferimento al file standard *input.*

write e writeln

Le istruzioni di procedura standard

```
write (f,x1,x2,...xn)
writeln (f,x1,x2,...,xn)
writeln (f)
```

si possono usare con qualsiasi file testo *f.*

La forma

```
write (f,x1,x2,...,xn)
```

è equivalente a

```
begin
    write (f,x1);
    write (f,x2);
    .
    .
    .
    write (f,xn)
end
```

La forma

writeln (f,x1,x2,...,xn)

viene definita equivalente a

```
begin
    write (f,x1,x2,...,xn);
    writeln (f)
end
```

L'effetto di

writeln (f)

è di appendere un carattere di fine linea al file *f*, ed è definito solo per i file testo. Se si stampa il file, i caratteri dopo il carattere di fine linea appariranno sulla linea successiva. Se il file viene ridato in input ad un programma PASCAL, la condizione di fine linea sopra descritta si verificherà non appena viene incontrato il carattere di fine linea.

L'effetto della forma

write (f,x)

in cui *f* è un file testo, è la scrittura in *f* di una sequenza di caratteri che denota il valore specificato dal parametro *x*. Tale parametro deve assumere la forma di un *valore-output*, definita nel Cap. 5, che determina il valore da mettere in output ed il formato in cui deve essere rappresentato.

Come osservato nel Cap. 5, l'uso di *write* e *writeln*, senza il parametro file *f*, implica il riferimento al file standard *output*.

page

La chiamata di procedura standard

page (f)

si può usare coi file testo da inviare ad un dispositivo di output che consente la paginazione, come una stampante di linee. Il suo unico effetto è di provocare un salto del dispositivo all'inizio della nuova pagina. L'uso di page senza un parametro file

page

fa riferimento al file standard *output*.

```
procedura copiafile (var f,g : text);
var c : char;
begin
    reset (f);
    rewrite (g);
    begin { copia una linea da f a g }
        while not eof (f) do
        begin
            read (f,c);
            write (g,c)
        end;
        readln (f);
        writeln (g)
    end
end
```
(26)

La procedura (26), dati due file testo *f* e *g* come parametri, copia il contenuto di *f* in *g*, in modo tale da mantenere la struttura a linee del testo del file *f*. Notare che l'effetto di questa procedura, quando viene chiamata con uno dei file standard *input* o *output* come parametro, è definito dall'implementazione, poiché in essa vengono eseguite le operazioni di *reset* e *rewrite* sui parametri. In alcune implementazioni una chiamata di questo genere può provocare un errore. In tal caso, per copiare nel file *output* o dal file *input* bisogna usare una procedura apposita.

Come spiegato nel Cap. 5 per il file *input*, si assume che un file testo che deve essere letto sia formato da un numero intero di linee. Quando invece un file testo deve essere scritto, se l'ultima azione prima di un'operazione di *reset* o della fine dell'esecuzione del programma non è un *writeln*, il *writeln* viene inserito implicitamente. Questo garantisce che, per le successive operazioni di lettura, la struttura in linee del contenuto del file sia corretta.

PROGRAMMA 16 (Un text editor)

La maggior parte dei sistemi di calcolo fornisce agli utenti un programma standard chiamato "editor". Un editor assiste l'utente nella stesura di file di informazioni, sottoforma di testo. Le correzioni e le modifiche dei dati si possono specificare con comandi di editor semplici e allo stesso tempo potenti. L'editor obbedisce a questi comandi, creando un "nuovofile" in cui viene riportato il contenuto del "vecchiofile", aggiornato in base all'effetto dei comandi.

Vogliamo descrivere la costruzione di un semplice editor che fa uso di quattro tipi di comandi – *T* (trascrizione), *C* (cancellazione), *I* (inserzione) e *E* (fine). Ciascuno di questi comandi compie un'azione sul contenuto del vecchiofile per produrre il nuovofile. I comandi utilizzano un contatore del numero di linea teorico che fa riferimento ad una particolare linea del vecchiofile. Le linee di un file si considerano numerate a partire da zero in avanti. Inizialmente zero si assume come numero di linea, cioè all'inizio viene puntata la prima linea. I comandi vengono forniti all'editor tramite il file standard input.

Per poter illustrare le azioni dei comandi, assumiamo l'esistenza di un vecchiofile contenente le seguenti linee di dati:

JIM SHORT
DAVE ANDERSON
JOHN HUGHES
PETER HALLIDAY
FRED DAVIES

Il comando *T* appare da solo su una linea, ed ha il seguente formato:

T numerolinea

Il suo effetto è di copiare un testo, senza modificarlo, dal vecchiofile al nuovofile. La copia inizia alla linea indicata dal valore corrente del numero di linea teorico, e prosegue sino alla linea *numerolinea,* senza includerla. Per cui, se

T2

è il primo comando per l'editor, le linee

JIM SHORT
DAVE ANDERSON

del vecchiofile vengono copiate nel nuovofile, ed il valore del contatore di linea diventa 2.

Anche il comando *C* si trova da solo su una linea, ed il suo formato è

C n

con *n* intero positivo. Il suo effetto è di incrementare di *n* il valore del contatore di linea, cioè fa saltare le successive n linee del vecchiofile. Il comando *C* si può quindi usare per cancellare le linee di un file.

Dopo l'esecuzione dei tre comandi seguenti sul vecchiofile dato

T1
C1
T3

il nuovofile conterrà

JIM SHORT
JOHN HUGHES

ed il valore del contatore di linea sarà 3.

Il comando *I* serve per inserire nuove linee di testo nel nuovofile. Il suo formato è

I n

su una linea separata, in cui *n* è un intero positivo che sta ad indicare il numero di linee di testo che devono essere inserite. Il comando è seguito dalle *n* linee. Applicando quindi la serie di comandi

T1
C1
I2

BILL COOK
COLIN JAMES

al vecchiofile si otterrà un nuovofile contenente

JIM SHORT
BILL COOK
COLIN JAMES

ed il valore del contatore di linea sarà 2.

Il comando *E* denota la fine dei comandi di editing, ed appare da solo su una linea. Il suo effetto è di copiare il resto del vecchiofile, a partire dalla posizione indicata dal valore corrente del contatore di linee, nel nuovofile. Per cui dopo l'esecuzione di

T2
C2
E

il nuovofile conterrà

JIM SHORT
DAVE ANDERSON
FRED DAVIES

Costruiamo ora il programma PASCAL per implementare questo editor. Il vecchiofile ed il nuovofile saranno file testo esterni al programma e i comandi di editor saranno letti dal canale standard di *input,* cioè il file *input.* Si dovrà produrre l'eco dei comandi di editor nel file *output,* in cui si dovranno riportare anche le informazioni sugli errori presenti nei comandi ed individuati dall'editor. Possono capitare cinque tipi di errori che sono indicati dai seguenti messaggi:

1. **** *comando non riconosciuto*
2. **** *numero di linea specificato minore del numero di linea corrente*
3. **** *vecchiofile finito in anticipo*
4. **** *input non sufficiente per l'inserzione*
5. **** *comando di fine edit mancante*

L'errore 1 capita quando si incontra un comando di editing diverso da *T, I, C,* o *E.* L'errore 2 si verifica quando il comando *T* specifica una linea il cui numero è minore del contatore di linea corrente. L'errore 3 si riferisce ad un tentativo di trascrivere o cancellare più linee di quante ne siano presenti nel vecchiofile. L'errore 4 si ha quando un comando *I* specifica più linee di dati nuovi di quante ne sono specificate nel file di input. L'errore 5, infine, è dovuto alla mancanza del comando *E*—questo può indicare che non sono stati predisposti tutti i comandi desiderati.

L'intestazione del programma specifica i file coinvolti, cioè

program *editor (vecchiofile, nuovofile, input, output)*

in cui *nuovofile* e *vecchiofile* sono file testo dichiarati nel blocco del programma

var *vecchiofile, nuovofile* : *text*

L'azione base del programma è riportata in (27), e consiste in un ciclo che riconosce ed esegue i comandi di editor richiesti dall'utente.

```
begin
    inizializza i file ed il puntatore di linea corrente;
    fineediting := false;
    repeat
        leggi e manda l'eco del comando;                    (27)
        if il comando è valido
          then esegui il comando di editing
          else errore (1)
    until fineediting
end
```

fineediting è una variabile booleana usata per indicare che l'editing è terminato, o in modo normale o a causa di un errore. *errore* è una procedura che stampa il messaggio corrispondente all'errore individuato e provoca l'uscita dal programma editor (assegnando a *fineediting* il valore *true*). Il suo parametro per valore indica l'errore trovato.

L'istruzione *inizializza i file ed il puntatore di linea corrente* si esprime immediatamente nel modo seguente

```
reset (vecchiofile);
rewrite (nuovofile);
puntatorelineacorrente := 0
```

in cui

```
puntatorelineacorrente := 0..maxint
```

Per *leggi e manda l'eco del comando,* dobbiamo introdurre la variabile

```
operatore : char
```

per cui l'istruzione diventa

```
read (operatore);
write (operatore)
```

Un comando è valido se l'operazione sull'insieme

```
operatore in [ 'T', 'I', 'C', 'E' ]
```

dà come risultato il valore *true.*

L'istruzione *esegui il comando di editing* si esprime mediante l'istruzione-case (28) che discrimina le diverse azioni associate ai quattro comandi di editing.

```
case operatore of
    'T' : esegui la trascrizione;
    'C' : esegui la cancellazione;                          (28)
    'I' : esegui l'inserzione;
    'E' : fai terminare l'editing
end
```

L'istruzione *esegui la trascrizione* è mostrata in (29). Il processo di trascrizione deve garantire che non avvenga un tentativo di leggere oltre la fine di vecchiofile (errore 3).

Se l'errore non è riscontrato, viene fatta la copia dal vecchiofile, a partire dalla linea indicata da *puntatorelineacorrente* fino al *numerolinea* specificato.

```
begin
    readln (numerolinea);
    writeln (numerolinea);
    if numerolinea < puntatorelineacorrente                    (29)
       then errore (2)
       else trascrivi da vecchiofile in nuovofile
end
```

In (30) è riportato l'ulteriore raffinamento di questo processo.

```
begin
    while not eof (vecchiofile) and
              (puntatorelineacorrente < numerolinea) do        (30)
    begin
        copialinea (vecchiofile);
        puntatorelineacorrente := puntatorelineacorrente + 1
    end;
    if puntatorelineacorrente < numerolinea then errore (3)
end
```

La procedura *copialinea* copia la linea successiva del file specificato come parametro in *nuovofile.*

Si osservi che il test di fine del file è

puntatorelineacorrente < numerolinea

e non

eof (vecchiofile)

poiché *eof (vecchiofile)* e *puntatorelineacorrente >= numerolinea* possono diventare *true* contemporaneamente, e cioè quando viene eseguito un comando per trascrivere il file fino alla fine che non è un errore.

L'istruzione *esegui la cancellazione* determina il numero delle linee da cancellare, e salta quel numero di linee in *vecchiofile.* Deve inoltre controllare che non vi siano letture oltre la fine di *vecchiofile.* Per il salto delle linee, si può scrivere un programma analogo a quello per l'operazione di trascrizione, a meno della chiamata di *copialinea,* che va sostituita dalla chiamata

readln (vecchiofile)

Il processo *esegui l'inserzione* determina il numero di linee di dati da inserire in *nuovofile,* e copia tale numero di linee dal file di input in *nuovofile*–deve comunque controllare che non vi siano tentativi di leggere dopo la fine del file di input (errore 4), come mostrato in (31).

```
begin
    readln (numerolineedainserire);
```

```
    i := 0;
    while (i < numerolineedainserire) and not eof (input) do
    begin
        copialinea (input);
        i := i + 1                                          (31)
    end;
    if i < numerolineedainserire then errore (4)
end
```

La terminazione dell'editor richiede due azioni, cioè

```
begin
    copia il resto di vecchiofile in nuovofile;
    fineediting := true
end
```

Se l'utente dimentica il comando *E,* il ciclo del programma principale non termina normalmente ed il canale di input diventa vuoto. Per controllare questa possibilità il ciclo

```
repeat
    :
    :
until fineediting
```

deve terminare sotto una condizione estesa, e diventa

```
repeat
    :
    :
until fineediting or eof (input)
```

Se il ciclo non termina per via di un comando *E,* bisogna segnalare l'errore 5

```
if not fineediting then errore (5)
```

La costruzione della procedura errore è immediata. La sua intestazione è

```
procedure errore (numero : numeroerrore);
```

ed il suo effetto è di stampare il messaggio di errore corrispondente al valore del suo parametro. Nella procedura viene inoltre assegnato a *fineediting* il valore *true,* in modo da far terminare il programma.

La procedura

```
procedure copialinea (var f : text)
```

legge semplicemente la successiva linea di *f,* carattere per carattere, e la trasferisce in *nuovofile.*

Il Listato 16 mostra il programma editor completo, cui è stata aggiunta la procedura *stampa* che viene chiamata per stampare il contenuto di *vecchiofile,* prima che su di esso operi l'editor, ed il *nuovofile* prodotto dall'editor. Il lettore può così verificare la correttezza del programma per l'insieme di comandi dato.

LISTATO 16

```
PROGRAM EDITOR(VECCHIOFILE,NUOVOFILE,INPUT,OUTPUT);

TYPE NUMEROERRORE = 1..5;

VAR  VECCHIOFILE,NUOVOFILE : TEXT;
     PUNTATORE, LINEEDAINSERIRE,
     NUMEROLINEA, N , I     : 0..MAXINT;
     FINEEDIT               : BOOLEAN;
     OPERATORE              : CHAR;

     PROCEDURE STAMPA(VAR F : TEXT);
     VAR  CH : CHAR;
          LINEA : 0..MAXINT;
     BEGIN
        RESET(F);
        LINEA:= 0;
        WHILE NOT EOF(F) DO
        BEGIN
           WRITE(LINEA:4);
           LINEA:= LINEA + 1;
           WHILE NOT EOLN(F) DO
           BEGIN
              READ(F,CH); WRITE(CH)
           END;
           READLN(F); WRITELN
        END
     END; (* STAMPA *)

     PROCEDURE COPIALINEA (VAR F : TEXT);
     VAR  CH : CHAR;
     BEGIN
        WHILE NOT EOLN(F) DO
        BEGIN
           READ(F,CH); WRITE(NUOVOFILE,CH)
        END;
        READLN(F); WRITELN(NUOVOFILE)
     END; (* COPIALINEA *)

     PROCEDURE ERRORE(NUMERO : NUMEROERRORE);
     BEGIN
        WRITELN; WRITE('****    ');
        CASE NUMERO OF
           1 : WRITELN('COMANDO ',OPERATORE,' NON RICONOSCIUTO');
           2 : WRITELN('NUMERO SPECIFICATO',
                       'MINORE DEL NUMERO DI LINEA CORRENTE');
           3 : WRITELN('VECCHIOFILE ESAURITO PREMATURAMENTE');
           4 : WRITELN('L'' INPUT NON E'' SUFFICIENTE',
                       'ALL'' INSERIMENTO');
           5 : WRITELN('COMANDO DI FINE EDIT MANCANTE')
        END;
        WRITELN('**** EDIT INTERROTTO');
        FINEEDIT:= TRUE
     END; (* ERRORE *)
```

```
BEGIN (* PROGRAMMA PRINCIPALE *)
   WRITELN;
   WRITELN('**** CONTENUTO DI VECCHIOFILE');
   STAMPA(VECCHIOFILE);
   WRITELN;
   PUNTATORE:= 0;
   FINEEDIT:= FALSE;
   RESET(VECCHIOFILE);
   REWRITE(NUOVOFILE);
   WRITELN('**** LISTA DEI COMANDI DI EDIT');
   REPEAT
      READ(OPERATORE);
      WRITE(OPERATORE);
      IF NOT((OPERATORE = 'T') OR (OPERATORE = 'C') OR
             (OPERATORE = 'I') OR (OPERATORE = 'E'))
         THEN ERRORE(1)
         ELSE CASE OPERATORE OF
                'T' :
                BEGIN (* TRASCRIZIONE *)
                   READLN(NUMEROLINEA);
                   WRITELN(NUMEROLINEA:4);
                   IF NUMEROLINEA < PUNTATORE
                      THEN ERRORE(2)
                      ELSE
                         BEGIN
                            WHILE NOT EOF(VECCHIOFILE)
                                  AND (PUNTATORE < NUMEROLINEA) DO
                            BEGIN
                               COPIALINEA(VECCHIOFILE);
                               PUNTATORE:= PUNTATORE + 1 .
                            END;
                            IF PUNTATORE < NUMEROLINEA
                               THEN ERRORE(3)
                         END
                END;
                'C' :
                BEGIN (* CANCELLAZIONE *)
                   READLN(N);
                   WRITELN(N:4);
                   NUMEROLINEA:= PUNTATORE + N;
                   WHILE NOT EOF(VECCHIOFILE)
                         AND (PUNTATORE < NUMEROLINEA) DO
                      BEGIN
                         READLN(VECCHIOFILE);
                         PUNTATORE:= PUNTATORE + 1
                      END;
                   IF PUNTATORE < NUMEROLINEA
                      THEN ERRORE(3)
                END;
                'I' :
                BEGIN (* INSERIMENTO *)
                   READLN(LINEEDAINSERIRE);
                   WRITELN(LINEEDAINSERIRE:4);
                   I:= 0;
```

```
                WHILE NOT EOF(INPUT) AND (I < LINEEDAINSERIRE) DO
                BEGIN
                   COPIALINEA(INPUT);
                   I:= I + 1
                END;
                IF I < LINEEDAINSERIRE
                   THEN ERRORE(4)
             END;
             'E' :
             BEGIN (* FINE EDIT *)
                READLN;
                WHILE NOT EOF(VECCHIOFILE) DO
                   COPIALINEA(VECCHIOFILE);
                FINEEDIT:= TRUE
             END
          END
   UNTIL FINEEDIT OR EOF(INPUT);
   WRITELN; WRITELN;
   WRITELN('**** CONTENUTO DI NUOVOFILE');
   STAMPA(NUOVOFILE);
   WRITELN;
   IF NOT FINEEDIT
      THEN ERRORE(5)
END.
```

```
**** CONTENUTO DI VECCHIOFILE
   0    JIM SHORT
   1    DAVE ANDERSON
   2    JOHN HUGUES
   3    PETER HALLIDAY
   4    FRED DAVIES

**** LISTA DEI COMANDI DI EDIT
T   1
C   1
I   2
T   4
C   1
E

**** CONTENUTO DI NUOVOFILE
   0    JIM SHORT
   1    BILL COOK
   2    COLIN JAMES
   3    JOHN HUGUES
   4    PETER HALLIDAY
```

ESERCIZI

12.1 Usando il tipo record e la procedura di input richiesti dall'esercizio 10.3, scrivere un programma che legge un numero non specificato di nomi di studenti, con i voti relativi, e li memorizza in un file di record esterno, chiamato *Classe*1.

12.2 Adattare la procedura di ordinamento per fusione naturale, presentata nel Programma 15, per ordinare il file *Classe*1, creato nell'esercizio 12.1, in ordine alfabetico rispetto ai nomi degli studenti.

12.3 Scrivere un programma per leggere i record relativi agli studenti dal file *Classe*1, ordinato nell'esercizio 12.2, e tabularli nel formato usato nel Programma 2.

12.4 Usare i programmi sviluppati negli esercizi 12.1 e 12.2 per creare un secondo file ordinato di record, *Classe*2, per un'altra classe di studenti. Scrivere un programma che legge i file ordinati *Classe*1 e *Classe*2, e crea un unico file ordinato, *Classi,* contenente i record di tutti gli studenti.

12.5 Scrivere un programma che

(a) legge una sequenza di nomi di studenti, e li memorizza in un file interno chiamato *Richieste*;
(b) ordina il file in ordine alfabetico;
(c) tabula i record del file *Classi*, creato nell'esercizio 12.4, relativi agli studenti i cui nomi compaiono nel file *Richieste.*

12.6 Scrivere un programma che legge due file testo, *Pagina*1 e *Pagina*2, e li stampa uno accanto all'altro, nella forma:

*Pagina*1	*Pagina*2
.....	.....
.....	.....
.....	.....
.....	.....
.....	.....
	.....
	.....

I file stampati devono conservare la struttura a linee del corrispondente file di input. Si può assumere che, nei file, non vi siano linee più lunghe di metà della lunghezza massima ammessa dall'implementazione per le linee stampate.

13.
Puntatori

13.1. LA NOZIONE DI PUNTATORE

I tipi array, record e insieme, introdotti nei Capp. 9-11, permettono di descrivere strutture dati la cui forma e le cui dimensioni sono determinate a priori, ed ai cui componenti si fa riferimento in modo standard. I file, descritti nel Cap. 12, sono una struttura di dati la cui dimensione può variare, ma la cui forma ed i metodi di accesso sono fissati, per consentire un particolare tipo di implementazione. In molti problemi di programmazione si richiedono strutture dati la cui forma e le cui dimensioni devono poter variare durante il loro periodo di vita, ed i cui metodi di accesso sono legati alle caratteristiche del problema in questione.

Strutture di questo genere vengono spesso realizzate sotto forma di *strutture dati collegate,* in cui i componenti individuali della struttura sono *collegati,* o *puntano,* ad altri componenti. Nella Fig. 13.1 sono riportate alcune tipiche strutture collegate.

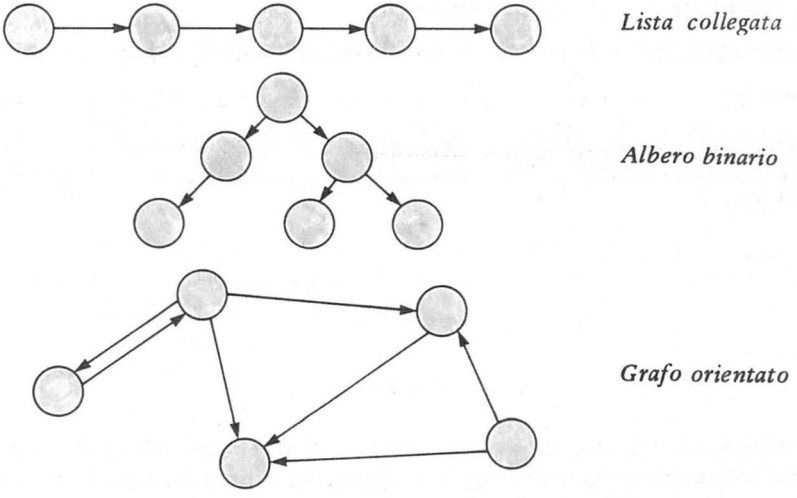


Fig. 13.1 Strutture dati collegate

Tali strutture, normalmente, crescono e si riducono dinamicamente durante l'esecuzione di un programma. Siccome il numero dei componenti della struttura non è fissato, né lo sono le connessioni tra i componenti, gli elementi della struttura devono essere creati e collegati dinamicamente durante l'esecuzione del programma.

Risulterebbe poco pratico, nel caso di un linguaggio per applicazioni di tipo generale, come il PASCAL, fornire un insieme di tipi di dato in grado di coprire tutte le possibili classi di strutture collegate. Il PASCAL, quindi, offre un unico meccanismo, chiamato *puntatore,* per consentire al programmatore di realizzare tali strutture, e di definire le operazioni su di esse applicabili.

Mentre le variabili di tutti i tipi di dati finora considerati in PASCAL sono *statiche,* nel senso che vengono dichiarate nel programma e denotate da un identificatore, e che la memoria ad esse associata continua ad esistere per tutto il tempo di vita del blocco in cui sono dichiarate, questo non è ugualmente vero per le variabili componenti delle strutture collegate. Queste variabili componenti vengono generate e distrutte *dinamicamente,* durante l'esecuzione del programma, e ad esse non si fa riferimento per mezzo di un identificatore dichiarato dall'utente. Ad esse si fa invece riferimento per mezzo di *variabili puntatore* ausiliarie, che puntano dinamicamente alla variabile creata. Ad esempio, la variabile puntatore *p* in Fig. 13.2 si dice che *punta,* o *fa riferimento,* alla variabile il cui valore è *x*. I modi per creare dinamicamente la variabile di valore *x* verranno illustrati nel seguito.

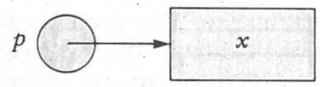


Fig. 13.2 Una variabile puntatore

In PASCAL un *tipo-puntatore* si definisce così

tipo-puntatore = "↑" *identificatore-tipo.*

Un programma PASCAL può, perciò, contenere una definizione di tipo nella forma

P = ↑ *T*

Si dice che i valori di tipo *P* fanno riferimento, o puntano, a variabili del tipo denotato da *T,* che deve essere un identificatore di tipo definito altrove. Il tipo *P* si dice legato al tipo *T.*

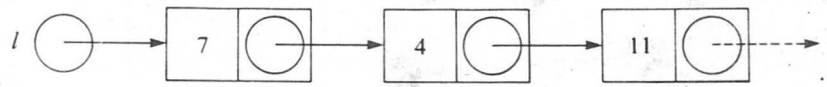


Fig. 13.3

Si consideri, ad esempio, una lista di interi, in cui ogni componente della lista punta all'elemento successivo, come in Fig. 13.3. Questa struttura è formata da un puntatore *l* al primo elemento della lista e da un certo numero di componenti collegati insieme

per mezzo di puntatori. In tal modo, ogni componente della lista risulta costituito da due dati: un valore di tipo *integer* ed un valore che fa riferimento ad un altro componente della lista. I componenti della lista sono pertanto di un tipo record al quale attribuiamo il nome *componentelista,* di due campi: un intero *i* ed un puntatore *successivo.* Chiamando, infine, il tipo puntatore *puntatorelista* siamo in grado di definirlo come riportato in (1).

```
type puntatorelista = ↑ componentelista;
     componentelista = record
                          i : integer;                    (1)
                          successivo : puntatorelista
                       end
```

Si osservi che la definizione del tipo *componentelista* viene dopo quella del tipo *puntatorelista,* in ogni record del tipo *componentelista,* permette la costruzione di una catenentelista. Questa situazione si verifica normalmente, nella definizione di tipi puntatore e tipi puntati e per questo motivo il PASCAL permette, in tali circostanze, l'uso di un identificatore di tipo prima della sua definizione. Tale definizione deve apparire più oltre, nella stessa parte di definizione di tipi. Questo è l'unico caso, in PASCAL, in cui un identificatore può essere usato prima della sua definizione.

Pertanto, nell'esempio precedente, l'inclusione di un campo puntatore di tipo *puntatorelista* in ogni record del tipo *componentelista,* permette la costruzione di una catena lineare di record, ciascuno con un elemento *i.* Come vedremo, l'inclusione di due o più puntatori in un componente, consente in modo analogo la costruzione di una notevole varietà di strutture non lineari.

La variabile *l,* che punta al primo componente della lista, si dichiara come variabile statica—come qualsiasi altro tipo un tipo puntatore si può usare per dichiarare una variabile puntatore statica:

```
var p : P;
    ⋮
```

Ad ogni modo, la dichiarazione di una variabile puntatore *p* non crea alcuna variabile cui punta *p,* ma soltanto la possibilità di farlo. La creazione della variabile si ottiene con una chiamata alla procedura standard *new.*

```
new (p)
```

L'effetto di questa chiamata è di creare una nuova variabile di tipo *T,* e di assegnare alla variabile *p* un valore che le permette di puntarla.

Le variabili create in questo modo saranno chiamate nel seguito *variabili-puntate,* e sono così definite

variabile-puntata = variabile-puntatore " ↑ ".
variabile-puntatore = variabile.

Un *variabile-puntata,* cioè una *variabile-puntatore* seguita dalla freccia verso l'alto ↑, denota la variabile cui punta la variabile puntatore.

Supponiamo che *l* sia una variabile del tipo *puntatorelista,* definito in precedenza. Dopo una chiamata *new (l)*

l ↑	denota la variabile del tipo *componentelista* creata,
l ↑*.i*	denota il componente intero di quella variabile, e
l ↑*.successivo*	denota il componente puntatore (vedi Fig. 13.4).

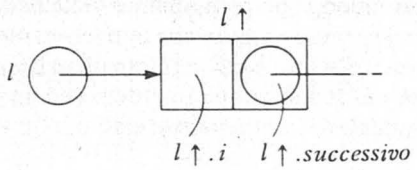


Fig. 13.4

I valori puntatore, e le variabili da essi puntate, si creano perciò per mezzo della procedura *new,* dopodiché vengono usati per far riferimento a queste variabili. I valori puntatore si possono anche copiare, con una assegnazione, in un'altra variabile puntatore.

Supponiamo che p e q siano variabili puntatore che fanno riferimento a diversi componenti di una lista, come in Fig. 13.5 (a). L'esecuzione dell'assegnazione

$q := p$

porterà alla situazione di Fig. 13.5 (b). Notare la differenza tra questa assegnazione e quella delle variabili puntate. Partendo dalla stessa situazione iniziale, l'esecuzione di

$q \uparrow := p \uparrow$

darebbe luogo alla situazione di Fig. 13.5 (c).

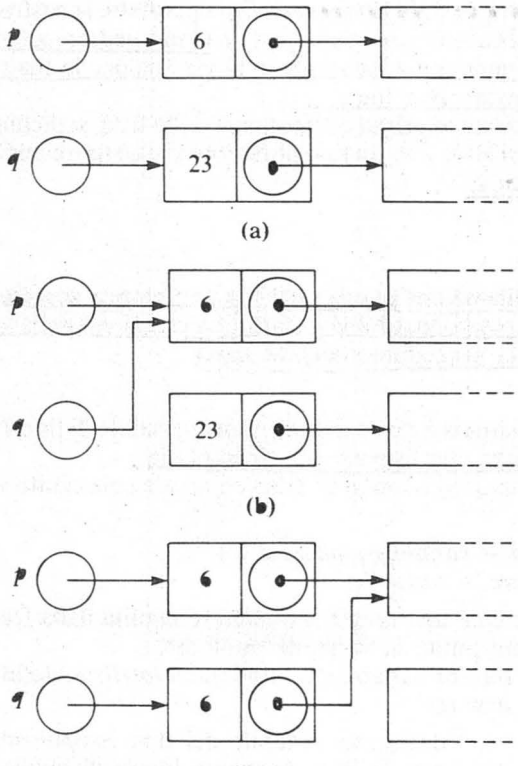


Fig. 13.5

Parecchie variabili-puntatore potrebbero, quindi, assumere lo stesso valore, cioè puntare alla stessa variabile. Per verificare se due puntatori fanno riferimento alla stessa variabile, si può fare un confronto usando gli operatori di uguaglianza o di disuguaglianza = e < >; per esempio

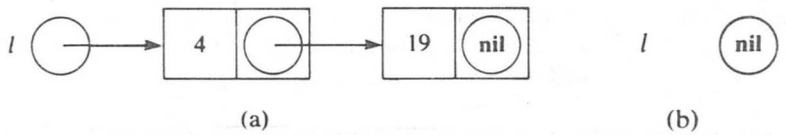
if $p = q$ **then** $p\uparrow.i := 0$

In Fig. 13.5 (b) $p = q$ è *true*
In Fig. 13.5 (c) $p = q$ è *false*

In alcuni casi è necessario indicare esplicitamente che un puntatore non punta nulla. A tal fine il PASCAL fornisce un valore particolare, **nil**, che si può assegnare ad una variabile puntatore,

p := **nil**

Il valore **nil** si può, ad esempio, usare nel componente finale di una lista di interi, per indicare che non ci sono altri interi nella lista, come in Fig. 13.6 (a), o assegnare alla variabile *l* stessa, ad indicare che la lista è vuota, come in Fig. 13.6 (b).



(a) (b)

Fig. 13.6

Nel manipolare una lista, si deve controllare l'occorrenza eventuale di un valore **nil**; per esempio

if $l <>$ **nil then** ...

L'effetto provocato da un tentativo di accesso ad una variabile puntata da una *variabile-puntatore,* il cui valore è **nil**, è indefinito in PASCAL, ma è chiaramente un errore di programmazione. Le implementazioni possono mettere a disposizione strumenti per individuare simili tentativi durante l'esecuzione di un programma.

Una volta creata dalla procedura *new,* una *variabile-puntatore* rimane in vita finché non viene esplicitamente distrutta da una chiamata alla procedura predefinita *dispose*

dispose (p)

in cui *p* è un puntatore alla variabile che si intende distruggere.

Si osservi che il tempo di vita di una variabile creata da *new* non sta in alcuna relazione con il blocco in cui viene creata, né con il blocco in cui si trova la dichiarazione delle *variabili-puntatore* che ad essa fanno riferimento. È compito del programmatore, al fine di economizzare la memoria, garantire il rilascio, con la procedura dispose, delle variabili indesiderate, prima che i puntatori ad esse siano perduti.

Dopo il rilascio di una variabile puntata, il valore di tutte le variabili-puntatore che facevano riferimento ad essa risultano indefiniti. Qualsiasi uso di uno di questi puntatori 'fluttuanti' diventa quindi un grave errore di programmazione. Per la stessa ragione si commette un errore rilasciando una variabile puntata, mentre viene usata (oppu-

re uno dei suoi componenti viene usato) come parametro per variabile o come variabile-record, in un'istruzione-with. Spesso le implementazioni non sono in grado di individuare tali errori, per cui il programmatore deve stare molto attento per evitarli.

13.2. PROGRAMMAZIONE DI UNA PILA

Per illustrare l'uso dei puntatori, consideriamo l'organizzazione di una *pila (stack)*. Una pila è una struttura dati costituita da un numero variabile di componenti dello stesso tipo. Un nuovo componente si può inserire sulla pila (operazione *push*), e si può eliminare il componente sulla sommità della pila (operazione *pop*), in modo che l'elemento eliminato sia l'ultimo che era stato inserito.

Per rappresentare una pila, ad esempio di caratteri, usiamo una catena di variabili di tipo record, ciascuna delle quali contenente un carattere ed un puntatore al record successivo della catena. Le definizioni di tipo che occorrono sono riportate in (2).

```
type puntatorepila = ↑ componentepila;
     componentepila = record
                        c : char;                    (2)
                        successivo : puntatorepila
                      end
```

I record vengono collegati tra loro nell'ordine in cui i caratteri vanno eliminati dalla pila, mediante una variabile puntatore *pila* che fa riferimento, in ogni momento, al prossimo record da eliminare

```
var pila : puntatorepila;
```

Pertanto, se i caratteri *'A'*, *'B'* e *'C'* sono stati inseriti sulla pila nell'ordine, la rappresentazione è quella di Fig. 13.7. Notare che nel campo puntatore dell'ultimo record compare il valore **nil**, ad indicare che non esistono altri record nella catena. Una variabile pila, cui sia stato assegnato il valore **nil**, rappresenta una pila *vuota,* cioè che non contiene alcun elemento.

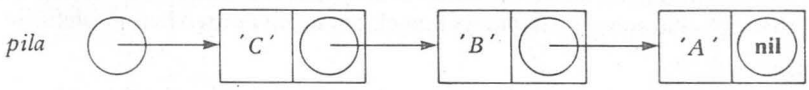


Fig. 13.7

L'operazione di inserimento di un nuovo carattere sulla pila si può esprimere con la procedura (3).

```
procedure push (x : char);
var nuovocomponente : puntatorepila;
begin
  new (nuovocomponente);
  with nuovocomponente ↑ do
```

```
    begin
      c := x;                                          (3)
      successivo := pila
    end;
    pila := nuovocomponente
  end
```

Consideriamo la rappresentazione della pila in Fig. 13.7; l'esecuzione della chiamata *push ('D')* produrrà la rappresentazione di Fig. 13.8.

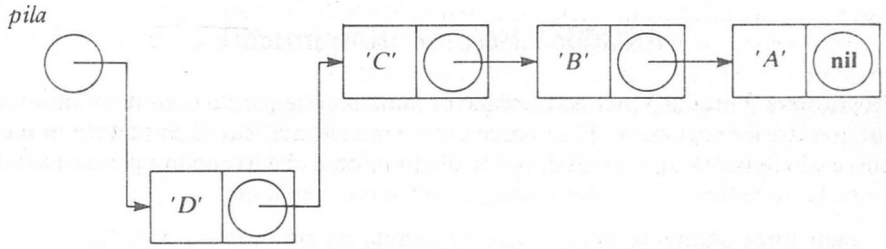


Fig. 13.8

L'operazione di eliminazione di un carattere dalla pila, e la sua assegnazione ad una variabile x di tipo carattere, si può esprimere con la procedura (4).

```
procedure pop (var x : char);
var vecchiocomponente : puntatorepila;
begin
    vecchiocomponente := pila;
    x := vecchiocomponente ↑.c;                        (4)
    pila := vecchiocomponente ↑.successivo;
    dispose (vecchiocomponente)
end
```

L'esecuzione di questa procedura sulla pila di Fig. 13.8 ripristina la situazione di Fig. 13.7. La procedura *pop* dovrebbe essere eseguita solo quando *pila* < > **nil**, cioè quando la pila non è vuota. Un ugual numero di operazioni di *push* e di *pop* su una pila inizialmente vuota restituiscono una pila vuota, cioè *pila* = **nil**.

L'uso di questa pila è illustrato dal frammento di programma seguente, che legge una sequenza di caratteri fino al punto e stampa la sequenza in ordine inverso.

```
pila := nil;
repeat
    read (ch);
    push (ch)
until ch = '.';
repeat
    pop (ch);
    write (ch)
until pila = nil
```

Usando record collegati da puntatori per realizzare una pila, si ha la garanzia di usare in ogni momento per la pila solo la quantità di memoria richiesta dal numero corrente dei suoi componenti, detto altezza. Al contrario, l'uso di un array richiede la definizione, a priori, della quantità di memoria sufficiente per una pila della massima altezza presunta, che rimane occupata per tutto il tempo di vita della pila. Un vantaggio offerto dal meccanismo dei puntatori è certamente la flessibilità nell'uso della memoria. Il Programma 17, che segue, illustra altre due situazioni che evidenziano tale caratteristica di flessibilità.

PROGRAMMA 17 (Riferimenti)

Il Programma 9 produce una concordanza ordinata delle parole trovate in un testo in input, mentre il Programma 11 produce una stampa formattata di un testo in input, riconoscendo le parole, i simboli di punteggiatura, ecc., che lo compongono. Estendiamo ora le specifiche del programma 11 nel modo seguente:

(a) ogni linea stampata deve essere preceduta da un numero di linea;
(b) la stampa completa del testo deve essere seguita da una lista di riferimenti (cross reference) contenente tutte le parole usate in ordine alfabetico e, per ciascuna di esse, una lista di tutte le linee in cui compare.

La struttura complessiva richiesta in questo caso (5), è un'estensione di quella concepita per il Programma 11.

```
begin
  leggiprossimoelemento;
  repeat
      questoelemento := prossimoelemento;
      leggiprossimoelemento;
      scriviquestoelemento;                                    (5)
      if questoelemento è una parola
        then registra la sua occorrenza
  until prossimoelemento è finetesto;
  stampa la tabella ordinata delle parole e delle loro occorrenze
end.
```

La struttura delle procedure *leggiprossimoelemento* e *scriviprossimoelemento* e le loro strutture dati restano invariate, con due eccezioni:

1. poiché le parole devono essere memorizzate e confrontate in altri punti del programma, bisogna introdurre un tipo (e un nome) per rappresentarle, cioè

   lettereparola = **packed array** [1..16] **of** *char*

   ed usarlo nel tipo *elementotesto*. Per facilitare i successivi confronti, conviene inoltre riempire le parole con spazi, come viene fatto nella procedura *leggiprossimoelemento*.

2. La procedura *scriviquestoelemento* deve ora far precedere tutte le linee stampate da un opportuno numero di linea. Questo si può realizzare:

   (a) introducendo una variabile globale

   questalinea : *numerolinea*

   per rappresentare in ogni momento il numero della linea stampata, in cui

   numerolinea = 0..9999;

   (b) inizializzando questa variabile a zero;
   (c) modificando la procedura *prendinuovalinea* come riportato in (6).

```
procedure prendinuovalinea;
begin
   writeln;
   questalinea := questalinea + 1;                    (6)
   write (questalinea : 5, '      ');
   spazioadestra := maxoutput - 7
end
```

Consideriamo ora le parti di registrazione e di stampa dei riferimenti delle parole presenti nel testo. Chiaramente serve una struttura dati in cui registrare le occorrenze durante la stampa del testo, per poterle poi a loro volta stampare.

Una struttura logica per questa tabella si può ottenere creando, per ogni parola distinta individuata nel testo, un record contenente la lista delle occorrenze di quella parola. Il processo di registrazione nella tabella delle occorrenze di una parola si può quindi suddividere in due passi:

cerca la parola nella tabella;
aggiungi il numero della linea alla sua lista delle occorrenze

in cui *cerca una parola nella tabella* deve creare un nuovo record, se la parola non era già stata precedentemente registrata ed inizializzare la sua lista di occorrenze vuota.

Nel Programma 9, per registrare le parole distinte trovate nel testo in input, abbiamo usato un semplice array monodimensionale. Questa soluzione ha lo svantaggio che bisogna stabilire un limite sul numero di parole distinte che si possono incontrare nel testo, ed un vettore di tali dimensioni va usato qualsiasi sia il numero di parole distinte in un particolare testo in input. Con i puntatori possiamo, invece, costruire una forma alternativa di tabella, in cui il numero degli elementi è, in qualsiasi momento esattamente quello richiesto dall'input.

```
puntatoreparola = ↑ recordparola;
recordparola = record
                  lettere : lettereparola;
                  occorrenze : listaoccorrenze;          (7)
                  successiva : puntatoreparola
               end
```

Potremmo usare per le parole una lista lineare di record con le definizioni di tipo

(7). La lista completa delle parole risulta quindi rappresentata da un'unica variabile-puntatore, che punta alla prima parola della lista:

listaparole : puntatoreparola

Poiché le parole devono essere stampate in ordine alfabetico, è logico costruire la lista in questo ordine. Il processo di ricerca nella lista di una data parola si può esprimere come una chiamata alla procedura (8).

```
procedure cercaparola (parolacercata : lettereparola;
                       var parolatrovata : puntatoreparola);          (8)
begin
   cerca nella lista finché non trovi una parola >= parolacercata oppure finché la
                                   lista non è finita;
   crea un nuovo record per la parola se serve
end
```

La ricerca nella lista richiede un ciclo, che termina quando si incontra una parola >= *parolacercata,* oppure quando la lista è finita. In entrambi i casi può essere necessario per la successiva inserzione un puntatore alla parola precedente, per cui il codice risulta quello riportato in (9).

```
questaparola := listaparole;
parolaprecedente := nil;
posizionetrovata := false;
while not posizionetrovata and (questaparola <> nil) do
   if questaparola ↑. lettere <= parolacercata                        (9)
     then posizionetrovata := true
     else begin
             parolaprecedente := questaparola;
             questaparola := questaparola ↑. successiva
          end
```

crea un nuovo record per la parola se serve si può quindi esprimere come in (10).

```
if posizionetrovata
  then if questaparola ↑. lettere = parolacercata
         then parolacercata := questaparola
         else inserisci un nuovo record (prima di questaparola)       (10)
  else inserisci un nuovo record alla fine della lista
```

In entrambi i casi gli elementi precedente e seguente il nuovo record sono indicati da *parolaprecedente* e *questaparola,* per cui il processo di inserzione si può esprimere come una chiamata alla procedura locale (11).

```
procedure inserisciparola;
var parolanuova : puntatoreparola;
begin
   new (parolanuova);
   with parolanuova ↑ do
   begin
```

```
            lettere := parolacercata;
            occorrenze := nessuna;
            successiva := questaparola
        end;
        if parolaprecedente = nil                                    (11)
            then listaparole := nil
            else parolaprecedente ↑.successiva := parolanuova;
        parolatrovata := parolanuova
    end
```

Notare che l'inserzione di un nuovo record, nella lista ordinata, si ottiene senza alcun movimento o copia degli altri record–un vantaggio significativo della rappresentazione a lista collegata. Inoltre, l'istruzione (scritta informalmente)

```
occorrenze := nessuna
```

non si può precisare finché non si determina la rappresentazione per la lista delle occorrenze.

Come si può rappresentare la lista delle occorrenze di una parola? La lunghezza di questa lista può variare in modo considerevole–ci saranno parole con una o due occorrenze, mentre per altre, usate comunemente, è necessario costruire una lista molto lunga. Queste variazioni di lunghezza si possono di nuovo gestire con una rappresentazione basata sui puntatori. I componenti della lista di occorrenze (che sono semplici numeri di linea) vengono aggiunti alla lista nello stesso ordine in cui devono essere stampati, cioè in ordine crescente di numero di linea. Una lista lineare di questo genere prende il nome di *coda* e si può rappresentare con una catena di record con i puntatori che scorrono dal primo all'ultimo e due puntatori ausiliari che indicano ad ogni istante il primo e l'ultimo record. In PASCAL, una lista di occorrenze si può realizzare con la definizione (12).

```
puntatorelista = ↑ recordlinea;
recordlinea = record
                  linea : numerolinea;
                  lineasuccessiva : puntatorelista                   (12)
              end;
listaoccorrenze = record
                      prima, ultima : puntatorelista
                  end
```

La lista delle occorrenze, per una parola presente sulle linee 3, 21 e 247, è rappresentata in Fig. 13.9.

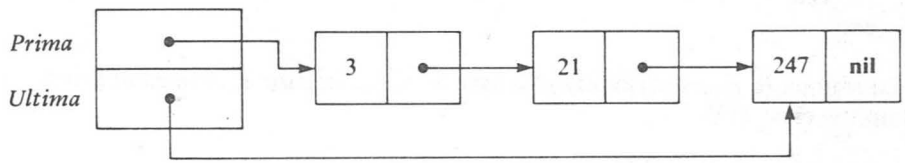


Fig. 13.9

L'aggiunta della linea corrente a una data lista di occorrenze si può esprimere con la procedura (13).

```
procedure aggiungiquestalinea (var lista : listaoccorrenze);
var occorrenza : puntatorelista;
begin
    new (occorrenza);
    with occorrenza ↑ do
    begin
        linea := questalinea;                                    (13)
        lineasuccessiva := nil
    end;
    with lista do
    begin
        if prima = nil
            then prima := occorrenza
            else ultima ↑.lineasuccessiva := occorrenza;
        ultima := occorrenza
    end
end
```

Il processo di stampa della lista ordinata di elementi collegati è anch'esso un semplice ciclo di attraversamento trasversale della lista. Nel nostro caso, la lista può essere distrutta man mano che viene stampata; il processo richiesto è quello riportato in (14).

```
procedure stampatabella;
var questaparola, parolasuccessiva : puntatoreparola;
begin
    page;
    writeln ('crossreference di parole e occorrenze');
    writeln; writeln;
    parolasuccessiva := listaparole;
    while parolasuccessiva <> nil do                             (14)
    begin
        questaparola := parolasuccessiva;
        stampa le lettere e le occorrenze di questaparola ↑;
        parolasuccessiva := questaparola ↑.successiva;
        dispose (questaparola)
    end
end
```

La stampa (e la distruzione) della lista delle occorrenze è, in linea di principio, il semplice ciclo (15).

```
prossimaoccorrenza := occorrenze.prima;
repeat
```

```
    questaoccorrenza := prossimaoccorrenza;
    write (questaoccorrenza ↑.linea);
    prossimaoccorrenza := questaoccorrenza ↑.lineasuccessiva;          (15)
    dispose (questaoccorrenza)
until prossimaoccorrenza = nil
```

Non possiamo tuttavia assumere che la lista dei numeri di linea, per ogni parola, sia di lunghezza minore od uguale di quella della linea di stampa. Quando occorre usare più di una linea indentiamo il primo numero in modo tale da non coprire la lista ordinata delle parole, come mostrato in Fig. 13.10.

```
⋮
TAMELY    9     113
THE       1       5      11   ...
         93      99     106   ...
        111
TOWN     46
⋮
```

Fig. 13.10

La stampa del testo nel nuovo formato, con i numeri di linea, fa di nuovo uso dell'intera larghezza di stampa, che vale *maxoutput* caratteri. Usando la stessa larghezza per la stampa del crossreference, abbiamo *maxparola* caratteri per le parole e *maxlista* (= *maxoutput* − *maxparola*) caratteri per la lista delle occorrenze. Se si introduce una variabile *spazioadestra,* come nel Programma 11, ad indicare il numero delle rimanenti posizioni di stampa sulla linea corrente, il processo completo di stampa di una parola e della sua lista di occorrenze è quello riportato in (16).

```
procedure stampaoccorrenze (lettere : lettereparola; occorrenze : listaoccorrenze);
var spazioadestra : 0..maxilista;
    questaoccorrenza, prossimaoccorrenza : puntatorelista;
begin
    write (lettere);
    spazioadestra := maxlista;
    prossimaoccorrenza := occorrenze.prima;
    repeat                                                              (16)
       questaoccorrenza := prossimaoccorrenza;
       if spazioadestra > 6
         then begin
                 writeln;
                 write ( ' ' : maxparola);
                 spazioadestra := maxilista
              end;
       write (questaoccorrenza ↑.linea : 6);
       spazioadestra := spazioadestra − 6;
       prossimaoccorrenza := questaoccorrenza ↑.lineasuccessiva;
       dispose (questaoccorrenza)
```

```
    until prossimaoccorrenza = nil;
    writeln
  end
```

Il Listato 17 mostra il programma completo, e l'output da esso prodotto per lo stesso input usato per i Programmi 9 e 11.

LISTATO 17

```
PROGRAM CROSSREFERENCE(INPUT,OUTPUT);

CONST MAXINPUT   = 81;
      FINELINEA  = '^';
      MAXOUTPUT  = 60;
      MAXPAROLA  = 16;
      MAXLISTA   = 44;

TYPE  TIPOELEMENTO   = (PAROLA,PUNTEGGIATURA,
                        IMPAGINAZIONE,FINETESTO);
      LETTEREPAROLA = PACKED ARRAY [1..MAXPAROLA] OF CHAR;
      ELEMENTO = RECORD
                   CASE TIPO : TIPOELEMENTO OF
                      PAROLA : (LUNGHEZZA : 1..MAXPAROLA;
                                LETTERE : LETTEREPAROLA);
                      PUNTEGGIATURA : (SIMBOLO : CHAR);
                      IMPAGINAZIONE : (FORMA :
                                       (LINEABIANCA,PARAGRAFO));
                      FINETESTO : ()
                 END;
      AMPIEZZALINEA   = 1..MAXINPUT;
      NUMEROLINEA     = 0..9999;
      PUNTATORELISTA  = ^RECORDLINEA;
      RECORDLINEA     = RECORD
                          LINEA : NUMEROLINEA;
                          LINEASUCCESSIVA : PUNTATORELISTA
                        END;
      LISTAOCCORRENZE = RECORD
                          PRIMA,ULTIMA : PUNTATORELISTA
                        END;
      PUNTPAROLA      = ^RECORDPAROLA;
      RECORDPAROLA    = RECORD
                          LETTERE : LETTEREPAROLA;
                          OCCORRENZE : LISTAOCCORRENZE;
                          SUCCESSIVA : PUNTPAROLA
                        END;

VAR   QUESTOELEMENTO,
      PROSSIMOELEMENTO : ELEMENTO;
      LINEA            : ARRAY [AMPIEZZALINEA] OF CHAR;
      PUNTATORE        : AMPIEZZALINEA;
      SPAZIOADESTRA    : 0..MAXOUTPUT;
      QUESTALINEA      : NUMEROLINEA;
      LISTA,
      QUESTAPAROLA     : PUNTPAROLA;
```

```
PROCEDURE LEGGIPROSSIMOELEMENTO;

   PROCEDURE LEGGILINEA;
   VAR  I : AMPIEZZALINEA;
   BEGIN
      I:= 1;
      WHILE NOT EOLN(INPUT) DO
      BEGIN
         READ(LINEA[I]);
         I:= I + 1
      END;
      READLN;
      LINEA[I]:= FINELINEA;
      PUNTATORE:= 1
   END; (* LEGGILINEA *)

   PROCEDURE CERCAELEMENTO;
   VAR  L : 0..MAXPAROLA;
   BEGIN
      WITH PROSSIMOELEMENTO DO
         CASE LINEA[PUNTATORE] OF
            'A','B','C','D','E','F','G',
            'H','I','J','K','L','M','N',
            'O','P','Q','R','S','T','U',
            'V','W','X','Y','Z' :
            BEGIN
               TIPO:= PAROLA;
               L:= 0;
               REPEAT
                  L:= L + 1;
                  LETTERE[L]:= LINEA[PUNTATORE];
                  PUNTATORE:= PUNTATORE + 1
               UNTIL (LINEA[PUNTATORE] < 'A')
                  OR (LINEA[PUNTATORE] > 'Z');
               LUNGHEZZA:= L;
               WHILE L < MAXPAROLA DO
               BEGIN
                  L:= L + 1;
                  LETTERE[L]:= ' '
               END
            END;
            ',','.',';',':' :
            BEGIN
               TIPO:= PUNTEGGIATURA;
               SIMBOLO:= LINEA[PUNTATORE];
               PUNTATORE:= PUNTATORE + 1
            END
         END
   END; (* CERCAELEMENTO *)

BEGIN (* LEGGIPROSSIMOELEMENTO *)
   WHILE LINEA[PUNTATORE] = ' ' DO
      PUNTATORE:= PUNTATORE + 1;
   IF LINEA[PUNTATORE] = FINELINEA
      THEN IF EOF(INPUT)
```

```
                THEN PROSSIMOELEMENTO.TIPO:= FINETESTO
                ELSE BEGIN
                        LEGGILINEA;
                        WHILE LINEA[PUNTATORE] = ' ' DO
                           PUNTATORE:= PUNTATORE + 1;
                        WITH PROSSIMOELEMENTO DO
                           IF LINEA[PUNTATORE] = FINELINEA
                              THEN BEGIN
                                      TIPO:= IMPAGINAZIONE;
                                      FORMA:= LINEABIANCA
                                   END
                              ELSE IF PUNTATORE > 2
                                      THEN BEGIN
                                              TIPO:= IMPAGINAZIONE;
                                              FORMA:= PARAGRAFO
                                           END
                                      ELSE CERCAELEMENTO
                     END
        ELSE CERCAELEMENTO
END; (* LEGGIPROSSIMOELEMENTO *)

PROCEDURE SCRIVIQUESTOELEMENTO;
VAR  SPAZIONECESSARIO : 1..18;
     I : 1..16;
     PROCEDURE PRENDILINEANUOVA;
     BEGIN
        WRITELN;
        QUESTALINEA:= QUESTALINEA + 1;
        WRITE(QUESTALINEA:5,' ');
        SPAZIOADESTRA:= MAXOUTPUT - 7
     END; (* PRENDILINEANUOVA *)


BEGIN
   WITH QUESTOELEMENTO DO
      CASE TIPO OF
         PAROLA :
         BEGIN
            IF PROSSIMOELEMENTO.TIPO = PUNTEGGIATURA
               THEN SPAZIONECESSARIO:= LUNGHEZZA + 2
               ELSE SPAZIONECESSARIO:= LUNGHEZZA + 1;
            IF SPAZIONECESSARIO > SPAZIOADESTRA
               THEN PRENDILINEANUOVA
               ELSE IF SPAZIOADESTRA <> MAXOUTPUT
                       THEN BEGIN
                               WRITE(' ');
                               SPAZIOADESTRA:= SPAZIOADESTRA-1
                            END;
            FOR I:= 1 TO LUNGHEZZA DO
               WRITE(LETTERE[I]);
            SPAZIOADESTRA:= SPAZIOADESTRA - LUNGHEZZA
         END;
         PUNTEGGIATURA :
         BEGIN
            WRITE(SIMBOLO);
```

```
            SPAZIOADESTRA:= SPAZIOADESTRA - 1
         END;
         IMPAGINAZIONE :
         BEGIN
            IF SPAZIOADESTRA < MAXOUTPUT
               THEN PRENDILINEANUOVA;
            IF FORMA = LINEABIANCA
               THEN PRENDILINEANUOVA
               ELSE BEGIN
                       WRITE(' ');
                       SPAZIOADESTRA:= SPAZIOADESTRA - 2
                    END
         END
      END
END; (* SCRIVIQUESTOELEMENTO *)

PROCEDURE CERCAPAROLA(PAROLACERCATA : LETTEREPAROLA;
                      VAR PAROLATROVATA : PUNTPAROLA);
VAR  QUESTAPAROLA,PAROLAPRECEDENTE : PUNTPAROLA;
     POSIZIONETROVATA : BOOLEAN;

     PROCEDURE INSERISCIPAROLA;
     VAR  PAROLANUOVA : PUNTPAROLA;
     BEGIN
        NEW(PAROLANUOVA);
        WITH PAROLANUOVA^ DO
        BEGIN
           LETTERE:= PAROLACERCATA;
           WITH OCCORRENZE DO
           BEGIN
              PRIMA:= NIL;
              ULTIMA:= NIL
           END;
           SUCCESSIVA:= QUESTAPAROLA
        END;
        IF PAROLAPRECEDENTE = NIL
           THEN LISTA:= PAROLANUOVA
           ELSE PAROLAPRECEDENTE^.SUCCESSIVA:= PAROLANUOVA;
        PAROLATROVATA:= PAROLANUOVA
     END; (* INSERISCIPAROLA *)

BEGIN
   QUESTAPAROLA:= LISTA;
   PAROLAPRECEDENTE:= NIL;
   POSIZIONETROVATA:= FALSE;
   WHILE NOT POSIZIONETROVATA AND (QUESTAPAROLA <> NIL) DO
      IF QUESTAPAROLA^.LETTERE <> PAROLACERCATA
         THEN POSIZIONETROVATA:= TRUE
         ELSE BEGIN
                 PAROLAPRECEDENTE:= QUESTAPAROLA;
                 QUESTAPAROLA:= QUESTAPAROLA^.SUCCESSIVA
              END;
      IF POSIZIONETROVATA
         THEN IF QUESTAPAROLA^.LETTERE = PAROLACERCATA
                 THEN PAROLATROVATA:= QUESTAPAROLA
```

```
                ELSE INSERISCIPAROLA
        ELSE INSERISCIPAROLA
END; (* CERCAPAROLA *)

PROCEDURE AGGIUNGIQUESTALINEA(VAR LISTA : LISTAOCCORRENZE);
VAR  OCCORRENZA : PUNTATORELISTA;
BEGIN
   NEW(OCCORRENZA);
   WITH OCCORRENZA^ DO
   BEGIN
      LINEA:= QUESTALINEA;
      LINEASUCCESSIVA:= NIL
   END;
   WITH LISTA DO
   BEGIN
      IF PRIMA = NIL
         THEN PRIMA:= OCCORRENZA
         ELSE ULTIMA^.LINEASUCCESSIVA:= OCCORRENZA;
      ULTIMA:= OCCORRENZA
   END
END; (* AGGIUNGIQUESTALINEA *)

PROCEDURE STAMPATABELLA;
VAR  QUESTAPAROLA,SUCCESSIVAPAROLA: PUNTPAROLA;

     PROCEDURE STAMPAOCCORRENZE(LETTERE : LETTEREPAROLA;
                                OCCORRENZE : LISTAOCCORRENZE);
     VAR  SPAZIOADESTRA : 0..MAXLISTA;
          QUESTAOCCORRENZA,
          SUCCESSIVAOCCORRENZA : PUNTATORELISTA;
     BEGIN
        WRITE(LETTERE);
        SPAZIOADESTRA:= MAXLISTA;
        SUCCESSIVAOCCORRENZA:= OCCORRENZE.PRIMA;
        REPEAT
           QUESTAOCCORRENZA:= SUCCESSIVAOCCORRENZA;
           IF SPAZIOADESTRA < 6
              THEN BEGIN
                      WRITELN;
                      WRITE(' ':MAXPAROLA);
                      SPAZIOADESTRA:= MAXLISTA
                   END;
           WRITE(QUESTAOCCORRENZA^.LINEA:6);
           SPAZIOADESTRA:= SPAZIOADESTRA - 6;
           SUCCESSIVAOCCORRENZA:= QUESTAOCCORRENZA^.LINEASUCCESSIVA;
           DISPOSE(QUESTAOCCORRENZA)
        UNTIL SUCCESSIVAOCCORRENZA = NIL;
        WRITELN
     END; (* STAMPAOCCORRENZE *)

     BEGIN (* STAMPATABELLA *)
        PAGE(OUTPUT);
        WRITELN('CROSSREFERENCE DI PAROLE ED OCCORRENZE');
        WRITELN;
        WRITELN;
```

```
        SUCCESSIVAPAROLA:= LISTA;
        WHILE SUCCESSIVAPAROLA <> NIL DO
        BEGIN
           QUESTAPAROLA:= SUCCESSIVAPAROLA;
           WITH QUESTAPAROLA^ DO
              STAMPAOCCORRENZE(LETTERE,OCCORRENZE);
           SUCCESSIVAPAROLA:= QUESTAPAROLA^.SUCCESSIVA;
           DISPOSE(QUESTAPAROLA)
        END
     END; (* STAMPATABELLA *)


BEGIN (* PROGRAMMA PRINCIPALE *)
   LINEA[1]:= FINELINEA;
   PUNTATORE:= 1;
   SPAZIOADESTRA:= 0;
   LISTA:= NIL;
   QUESTALINEA:= 0;
   LEGGIPROSSIMOELEMENTO;
   REPEAT
      QUESTOELEMENTO:= PROSSIMOELEMENTO;
      LEGGIPROSSIMOELEMENTO;
      SCRIVIQUESTOELEMENTO;
      IF QUESTOELEMENTO.TIPO = PAROLA
         THEN BEGIN
                 CERCAPAROLA(QUESTOELEMENTO.LETTERE,QUESTAPAROLA);
                 AGGIUNGIQUESTALINEA(QUESTAPAROLA^,OCCORRENZE)
              END
   UNTIL PROSSIMOELEMENTO.TIPO = FINETESTO;
   STAMPATABELLA
END.
```

```
 1 QUOT SINT GENERA PRINCIPATUM ET QUIBUS NOBIS
 2 ACQUIRANTUR
 3
 4  TUTTI GLI STATI, TUTTI E DOMINI CHE HANNO AVUTO ED
 5 HANNO IMPERIO SOPRA GLI UOMINI, SONO STATI E SONO O
 6 REPUBBLICHE O PRINCIPATI.
 7
 8
 9   I PRINCIPATI SONO, O EREDITARII, DE QUALI EL
10 SANGUE DEL LORO SIGNORE NE SIA SUTO LUNGO TEMPO
11 PRINCIPE, O E SONO NUOVI.


CROSSREFERENCE DI PAROLE ED OCCORRENZE


ACQUIRANTUR          2
AVUTO                4
CHE                  4
DE                   9
DEL                 10
```

```
DOMINI          4
E               4     5     11
ED              4
FL              9
EREDITARII      9
ET              1
GENERA          1
GLI             4     5
HANNO           4     5
I               9
IMPERIO         5
ILORO          10
LUNGO          10
MODIS           1
NE             10
NUOVI          11
O               5     6     9     11
PRINCIPATI      6     9
PRINCIPATUM     1
PRINCIPE       11
QUALI           9
QUIBUS          1
QUOT            1
REPUBBLICHE     6
SANGUE         10
SIA            10
SIGNORE        10
SINT            1
SONO            5     5     9     11
SOPRA           5
STATI           4     5
SUTO           10
TEMPO          10
TUTTI           4     4
UOMINI          5
```

13.3. STRUTTURE NON LINEARI

Finora abbiamo illustrato come costruire tre forme di strutture a lista lineari:

(a) la *pila,* su cui si opera ad una sola estremità;
(b) la *coda,* su cui gli elementi vengono inseriti ad un estremo e prelevati dall'altro;
(c) la *lista ordinata,* in cui si possono inserire elementi in qualsiasi punto.

Tutte queste strutture si possono realizzare con sequenze di record, ciascuno dei quali contiene un puntatore al successivo elemento della lista.

I record della lista ordinata di parole, costruita nel Programma 17, contengono, invece, anche un puntatore (o meglio due puntatori) alla corrispondente coda delle occorrenze. La struttura complessiva è illustrata in Fig. 13.11.

Vista nel suo insieme, questa risulta una struttura non lineare e mostra chiaramente che un nodo o record di una struttura collegata con due o più puntatori ad altri nodi crea una potenziale non linearità. I puntatori del PASCAL consentono pertanto, sen-

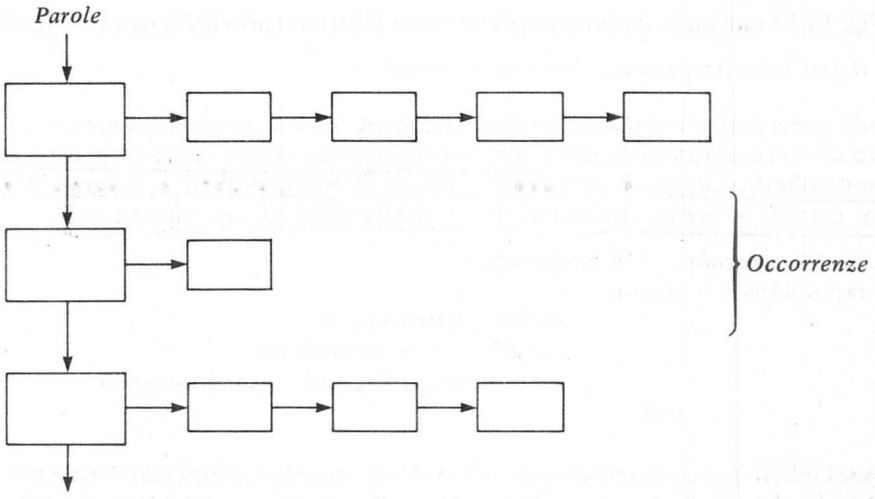


Fig. 13.11

za ulteriori estensioni, di costruire strutture non lineari di arbitraria complessità.

Il Programma 17 offre l'opportunità di illustrare l'uso (giudizioso) dei puntatori per creare strutture non lineari. La rappresentazione della lista ordinata di parole, sottoforma di lista lineare collegata, permette di limitare la quantità di memoria utilizzata a quella richiesta dalla lista in ogni istante e di inserire nella lista ordinata nuove parole senza doverne trasferire altre già presenti nella lista. Tuttavia, il semplice collegamento lineare implica che nella lista è possibile soltanto cercare una parola, o determinare la posizione corretta in cui inserirne una nuova, facendo un attraversamento lineare della lista. Nel caso di liste di una certa lunghezza, tale procedimento di ricerca lineare richiede tempi di esecuzione elevati.

Una ricerca più veloce, mantenendo la flessibilità di inserzione e di uso della memoria offerta dalla lista collegata, si può ottenere memorizzando la lista sottoforma di *albero binario.*

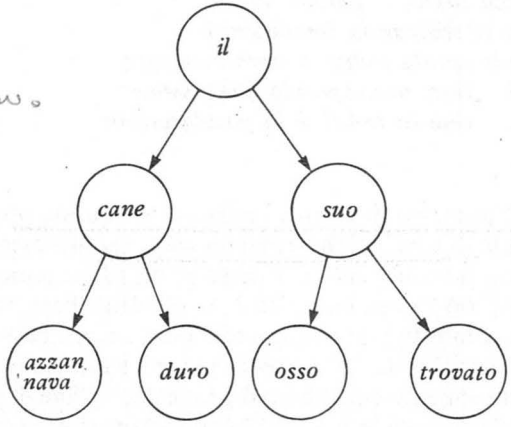


Fig. 13.12 Albero binario

La Fig. 13.12 mostra la rappresentazione come albero binario delle parole della frase

il suo cane azzannava il duro osso trovato

Il nodo per la parola *il* si chiama *radice* dell'albero, mentre quelli per *azzannava, duro, osso* e *trovato* si chiamano *foglie*. L'albero è ordinato in modo che le parole che stanno nel sottoalbero a sinistra di un nodo precedono, in ordine alfabetico, la parola in quel nodo, mentre le parole del sottoalbero a destra sono ad essa successive.

```
puntatoreparola = ↑ recordparola;
recordparola = record
                    lettere : lettereparola;
                    occorrenze : listaoccorrenze;                (17)
                    predecessori, successori : puntatoreparola
               end
```

Quest'albero si può rappresentare, in PASCAL, usando i record e puntatori definiti in (17). L'albero complessivo delle parole risulta, quindi, rappresentato da un'unica variabile puntatore

```
puntatorealbero : puntatoreparola;
```

Un albero vuoto ed i sottoalberi delle foglie si rappresentano con il valore **nil**.

L'albero binario è una struttura inerentemente ricorsiva, perché ogni nodo è definito in termini di altri due alberi della stessa forma. Il processo di ricerca di un elemento, e quello di creazione di un nuovo elemento in quest'albero si esprimono con la semplice procedura ricorsiva (18).

```
procedure ricercaparola (var questoalbero : puntatoreparola);
begin
   if questoalbero è vuoto
      then crea una foglia come questoalbero
      else if parola radice > parola cercata                     (18)
              then ricercaparola (predecenti)
              else if parola radice < parola cercata
                      then ricercaparola (successori)
                      else la radice è la parolacercata
end
```

Ad ogni passo del processo di ricerca realizzato da questa procedura si dimezza l'area in cui continua la ricerca, perché vengono scelti i predecessori o i successori del nodo corrente. Questa riduzione ad ogni passo porta ad un tempo di ricerca medio proporzionale a $\log_2 N$, con N pari al numero di parole dell'albero. In pratica, prestazioni logaritmiche sono garantite solo se l'albero è bilanciato, cioè ha, in ogni nodo, un ugual numero di parole nei sottoalberi predecessore e successore. Nel processo di inserzione visto, il bilanciamento dell'albero dipende dall'ordine di inserzione delle parole. Ad esempio, la frase usata in Fig. 13.12 porta ad un albero esattamente bilanciato, ma non tutti i dati si possono scegliere così bene. Tuttavia, le parole che si incon-

trano in un testo ordinario di certe dimensioni sono distribuite in modo sufficientemente casuale da produrre alberi abbastanza bilanciati, per cui si può assumere che il tempo medio di ricerca si avvicini a quello teorico $\log_2 N$.

La procedura *cercaparola* del Programma 17 si può quindi riscrivere come procedura locale ricorsiva (19).

```
procedure cercaparola (parolacercata : lettereparola;
            var parolatrovata : puntatoreparola);
  procedure ricercaparola (var questoalbero : puntatoreparola);
  ... come riportata in (18) ...
begin
  ricercaparola (alberoparole)                                    (19)
end
```

In questo contesto è semplice scrivere in dettaglio la procedura ricorsiva, come in (20).

```
procedure ricercaparola (var questoalbero : puntatoreparola);
begin
  if questoalbero = nil
    then begin
           new (questoalbero);
           with questoalbero ↑ do
           begin
             lettere := parolacercata;
             occorrenze := nessuna;
             predecessori := nil;
             successori := nil                                    (20)
           end;
           parolatrovata := questoalbero
         end
    else
         with questoalbero ↑ do
           if lettere > parolacercata
             then ricercaparola (predecessori)
             else if lettere < parolacercata
                    then ricercaparola (successori)
                    else parolatrovata := questoalbero
end;
```

Il processo di stampa della lista ordinata, rappresentata come albero binario, si esprime anch'esso con una procedura ricorsiva. Per un albero non vuoto i predecessori nel sottoalbero a sinistra vanno stampati prima della parola del nodo radice ed i successori dopo di essa. Se l'albero delle parole deve essere distrutto man mano che viene stampato, la procedura risultante è la (21).

```
procedure ristampaparola (questoalbero : puntatoreparola);
begin
   if questoalbero <> nil
      then with questoalbero ↑ do
           begin
              ristampaparola (predecessori);
              stampaoccorrenze (lettere, occorrenze);                (21)
              ristampaparola (successori);
              dispose (questoalbero)
           end
end
```

Il corpo della procedura principale *stampatabella,* usata nel Programma 17, diventa quindi quello riportato in (22).

```
begin
   page;
   writeln ('crossreference di parole e occorrenze');                (22)
   writeln; writeln;
   ristampaparola (alberoparole)
end
```

La lista lineare di parole usata nel Programma 17, si può così rimpiazzare con un albero binario, ridefinendo il tipo record *componenteparola,* e riscrivendo le procedure *cercaparola* e *stampatabella* che manipolano questi record. Il programma risultante dovrebbe essere più veloce nel caso di testi in input di lunghezza significativa.

La struttura ad albero binario è stata ottenuta introducendo, in ogni *componenteparola,* due componenti di tipo *puntatoreparola* al posto di uno. In modo analogo si possono realizzare una notevole varietà di strutture non lineari, usando un insieme di puntatori di collegamento tra i nodi componenti. La potenza che offre il meccanismo dei puntatori va però usata con attenzione, poiché la complessità delle strutture dati può facilmente oltrepassare i limiti della nostra comprensione, dando luogo, nei programmi che le manipolano, ad errori difficili da diagnosticare.

13.4. DIMENSIONAMENTO DELLA MEMORIA

Le variabili cui fanno riferimento i puntatori sono spesso record varianti, le cui varianti, in generale, richiedono quantità di memoria diverse per i vari componenti. Pertanto, una variabile di tipo record in cui viene cambiata la variante richiede una allocazione di memoria pari a quella occorrente per la sua variante più grande. Tuttavia, spesso capita che la variante di un record, creata da una particolare chiamata di *new*, sia nota a priori e rimanga fissa per tutto il suo tempo di vita. In questo caso si può pensare di usare soltanto la memoria richiesta da quella particolare variante al momento della creazione della variabile di tipo record. Il PASCAL consente di dimensionare l'occupazione di memoria di un record variante per mezzo di una chiamata in forma estesa della procedura standard *new*

new (p, t1, t2, ..., tn)

in cui *t1*, *t2*, ..., *tn* sono costanti che specificano il valore di selezione delle particolari varianti richieste. I valori di selezione devono essere elencati nell'ordine di occorrenza nella definizione del tipo record, cioè dalla variante più esterna a quella più interna. Ad ogni modo, si può omettere la specifica delle varianti più interne se si vuol permettere loro di cambiare.

Ad esempio, supponiamo di avere il tipo puntatore

```
puntatorepersona = ↑ persona
```

con il tipo *persona* definito come nel Cap. 10, cioè

```
type generepersona = nazionale, straniero;
     condizione = (pernascita, naturalizzato);
     persona = record
                  nome : nomepersona;
                  datadinascita : data;
                  case origine : generepersona of
                     nazionale : (luogonascita : nomeluogo;
                                  case qualifica : condizione of
                                    pernascita : ( );
                                    naturalizzato : (numero : integer;
                                         datanaturalizzazione : data));
                     straniero : (statorigine : nomeluogo;
                                  datadiingresso : data;
                                  portodiingresso : nomeluogo)
               end
```

Data la variabile puntatore

```
p : puntatorepersona
```

new (p)	crea un nuovo record che può avere entrambe le varianti *nazionale* e *straniero,* ed entrambe le sottovarianti *pernascita* e *naturalizzato*;
new (p, nazionale)	crea un nuovo record che deve avere la variante *nazionale,* ma che può avere entrambe le sottovarianti *pernascita* e *naturalizzato*;
new (p, nazionale, pernascita)	crea un nuovo record che deve avere la variante *nazionale* e la sottovariante *pernascita.*

Il dimensionamento di record varianti, fatto in questo modo, può consentire un notevole risparmio, in termini di memoria, nei programmi che manipolano un considerevole numero di questi record. Occorre comunque tenere ben presente che:

(a) Sebbene la forma estesa della chiamata di *new* fissi le varianti, non assegna, tuttavia, il valore al corrispondente campo di selezione, operazione che rimane a carico del programmatore.

(b) Una volta determinata da una chiamata di *new* in forma estesa, la variante di un

record non deve in alcun modo essere cambiata, per esempio assegnando un valore al campo di selezione.

(c) Per la stessa ragione, l'assegnazione di intere variabili di tipo record, create con una chiamata in forma estesa della procedura *new,* non sono permesse; si possono, però, assegnare valori ai singoli componenti.

La corrispondente forma estesa della procedura standard *dispose* è

dispose (p, t1, t2, ..., tn)

Il PASCAL standard impone che i record creati con la forma estesa della procedura *new* siano rilasciati con una chiamata di *dispose,* in forma estesa, con un insieme di costanti di selezione che implichino la stessa variante.

ESERCIZI

13.1 Il programma riportato nel Listato 17 usa un'unica variabile puntatore *listaparole.* Modificare il programma, in modo che usi un array

listaparole : **array** ['A' .. 'Z'] **of** *puntatoreparola*

i cui elementi siano puntatori al primo componente di una lista collegata di parole che iniziano con la lettera corrispondente.

13.2 In un settore di un ufficio, gli impiegati sono sistemati in stanze occupate da una o più persone, con un telefono per stanza. Una rubrica di numeri di telefono, interni al settore, è disponibile su un file, ciascun record del quale contiene il nome di un impiegato (di 16 caratteri) ed il suo numero di telefono. Il file è in ordine alfabetico in base al nome degli impiegati. Scrivere un programma che legge la rubrica, e stampa un elenco degli impiegati di ogni stanza (con il relativo numero di telefono). Gli elenchi vanno stampati in ordine crescente di numero telefonico, ed i nomi degli impiegati di ogni stanza in ordine alfabetico.

13.3 Riscrivere il Programma 10, in modo che la classifica risulti memorizzata come un array di puntatori, ciascuno dei quali faccia riferimento al corrispondente record *datisquadra.* che effetto ha questa variazione sull'efficienza del programma?

13.4 Un polinomio di grado arbitrario si rappresenta come una sequenza di coppie coefficiente-grado, di solito in ordine decrescente di grado. Ad esempio, il polinomio

$$x^8 + 5x^6 - 7x^5 + 6x + 1$$

si può rappresentare

(1,8), (5,6), (−7,5), (6,1), (1,0)

Definire un tipo puntatore, per poter memorizzare questa sequenza in una lista collegata.

Scrivere una procedura *leggipolinomio,* che legge un intero positivo che indica il numero di termini di un polinomio, seguito dalle coppie ordinate coefficiente-grado. Per esempio, la precedente rappresentazione costituirebbe l'input

```
5 1 8 5 6 -7 5 6 1 1 0
```

La procedura deve costruire la rappresentazione a lista collegata del polinomio.

Scrivere due funzioni *somma* e *prodotto*, che prendono come parametri due puntatori a rappresentazioni di polinomi, e danno come risultato il polinomio somma ed il polinomio prodotto rispettivamente. Usare tali funzioni in un programma in grado di calcolare la somma ed il prodotto dei seguenti polinomi

$$3x^2 + 9x + 1$$
$$x^3 - 3x^2 - 5x$$

dando il risultato nella forma

```
SOMMA     = X ↑ 3 + 4X + 1
PRODOTTO  = 3X ↑ 5 −41X ↑ 3 −48X ↑ 2 −5X
```

Appendice 1

Diagrammi sintattici

La sintassi del PASCAL viene descritta spesso per mezzo di diagrammi sintattici. Le sequenze di simboli ammesse per un costrutto linguistico si possono descrivere con un diagramma sintattico, che è un grafo orientato con un ingresso ed un'uscita. Ogni cammino nel grafo definisce una sequenza di simboli ammissibile.

Per esempio, la struttura di un programma PASCAL completo è definita dalla Fig. A1. L'occorrenza del nome di un altro diagramma, come, in Fig. 1, *identificatore, lista-identificatori* e *blocco,* indica che è ammessa in quel punto qualsiasi sequenza di simboli definita da quel diagramma. I nomi dei diagrammi sono sempre in corsivo; tutti gli altri simboli appartengono al linguaggio.

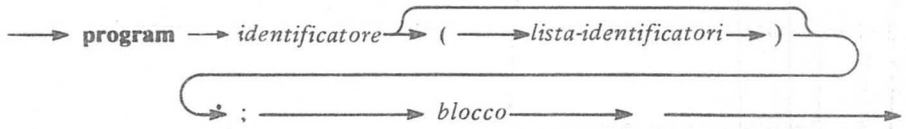


Fig. A1 *programma*

Sequenze alternative o ripetute di simboli sono indicate da biforcazioni o cicli nel diagramma sintattico corrispondente. Per esempio, un identificatore è definito dalla Fig. A2. I nomi *lettera* e cifra denotano rispettivamente una qualsiasi delle 52 lettere maiuscole o minuscole e le 10 cifre.

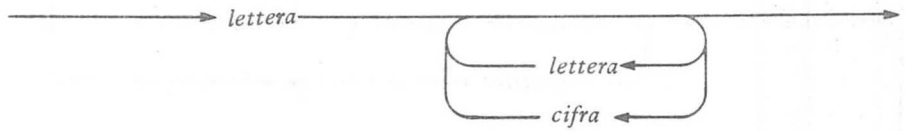


Fig. A2 *identificatore*

I diagrammi delle Figg. A3-26 definiscono la sintassi completa del PASCAL. Tutti i nomi che cominciano con *identificatore,* come *identificatore-procedura, identificatore-*

variabile, ecc., sono sintatticamente equivalenti a *identificatore,* ed indicano semplicemente la classe degli identificatori dichiarati che possono essere usati in quel punto.

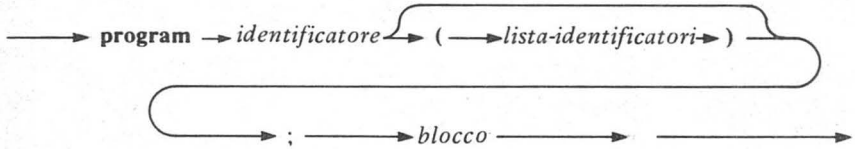


Fig. A3 *programma*

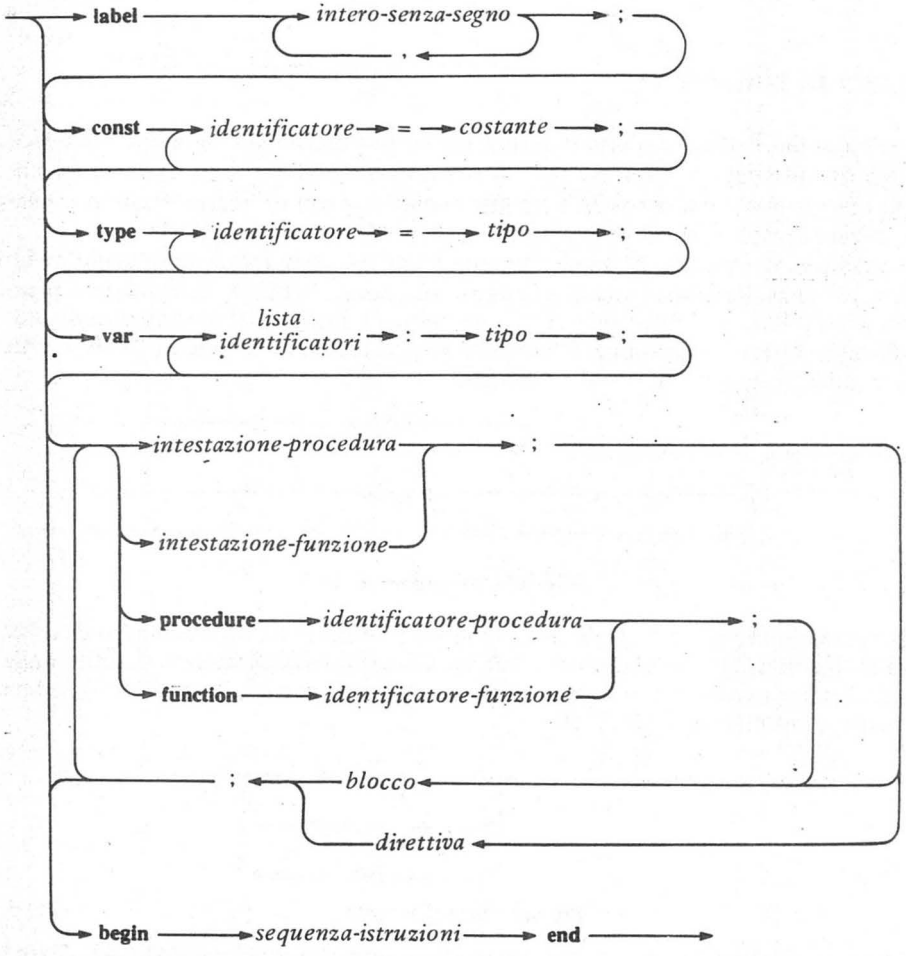


Fig. A4 *blocco*

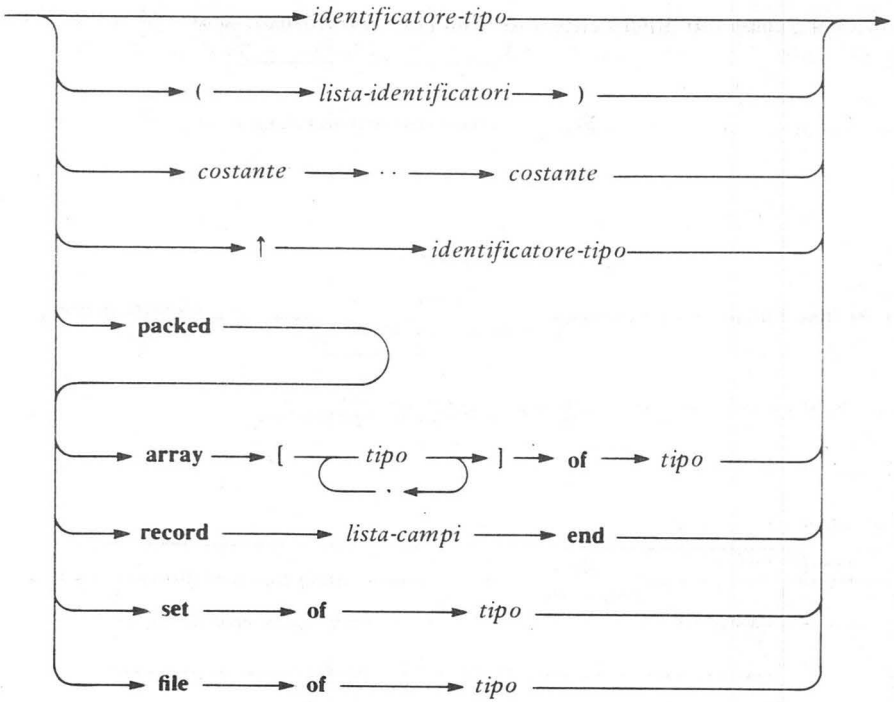


Fig. A5 *tipo*

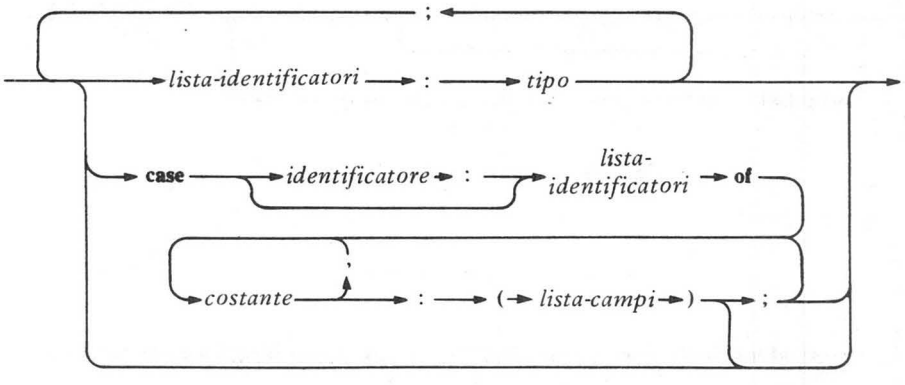


Fig. A6 *lista-campi*

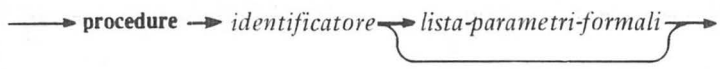


Fig. A7 *intestazione-procedura*

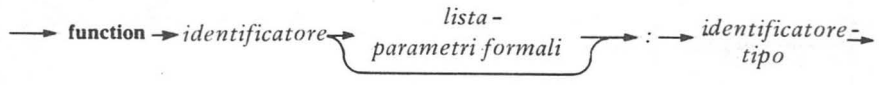


Fig. A8 *intestazione-funzione*

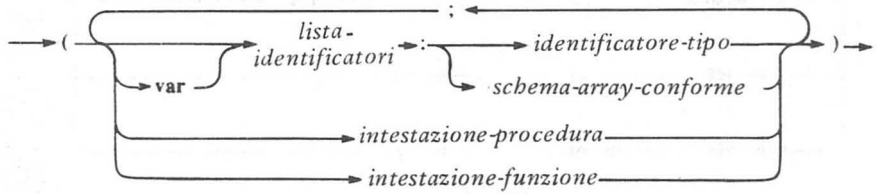


Fig. A9 *lista-parametri-formali*

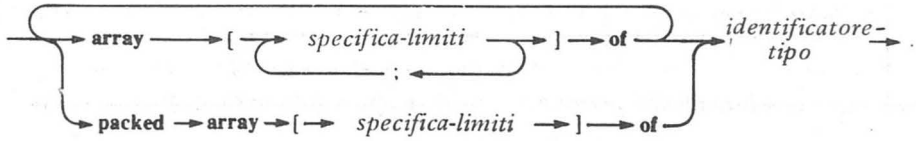


Fig. A10 *schema-array-conforme*

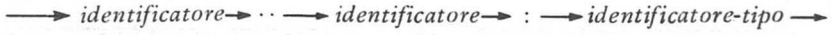


Fig. A11 *specifica-limiti*

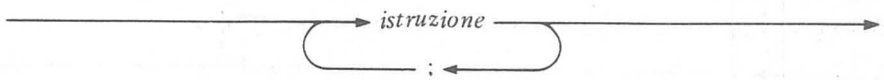


Fig. A12 *sequenza-istruzioni*

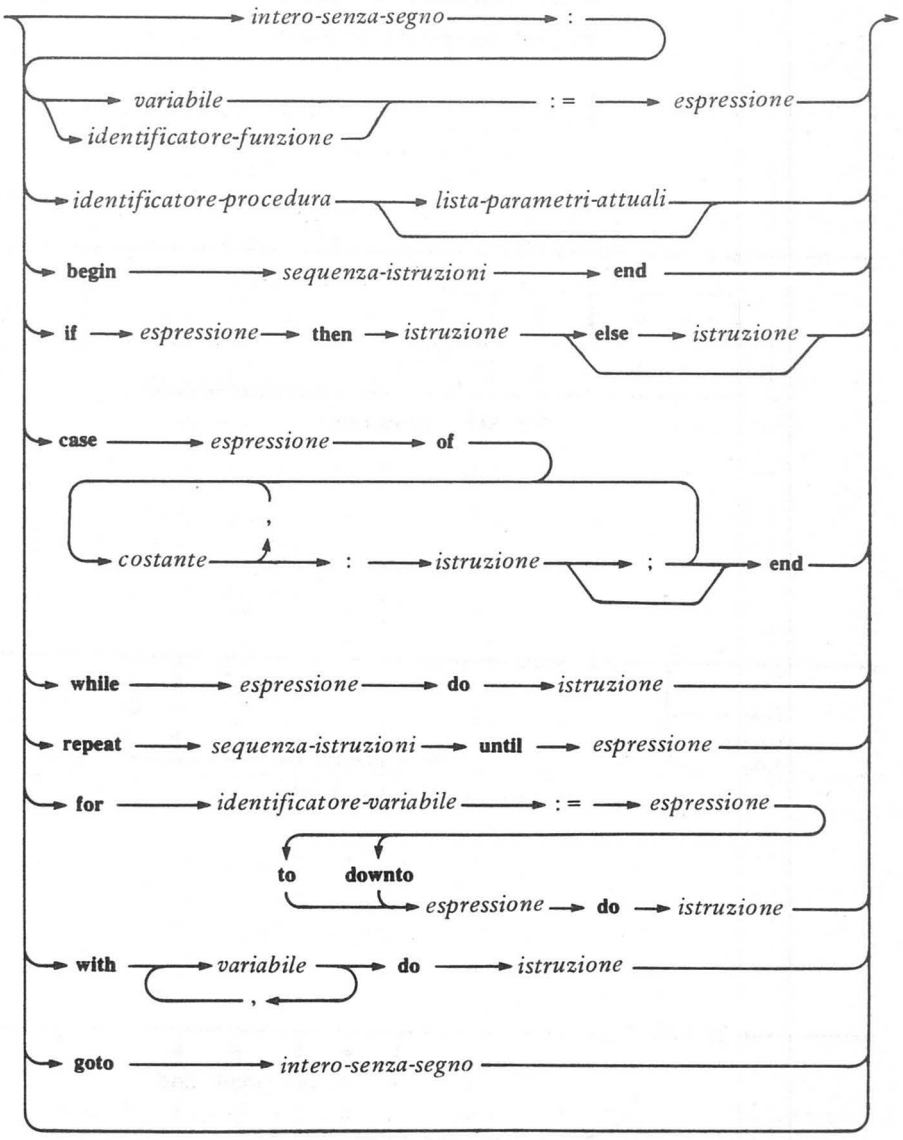


Fig. A13 *istruzione*

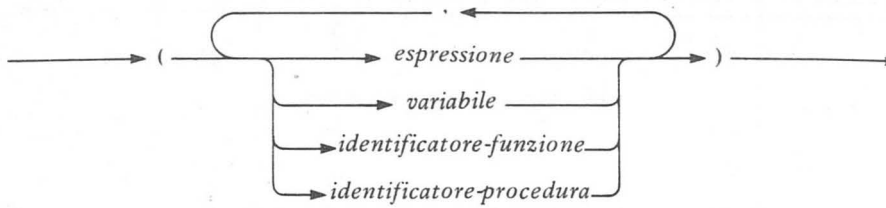


Fig. A14 *lista-parametri-attuali*

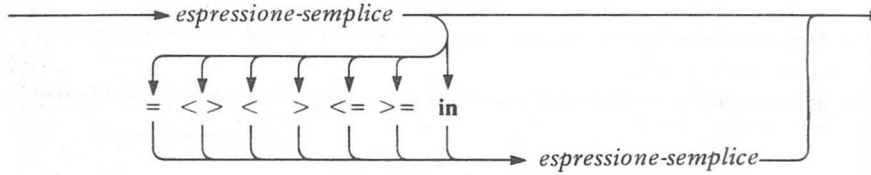


Fig. A15 *espressione*

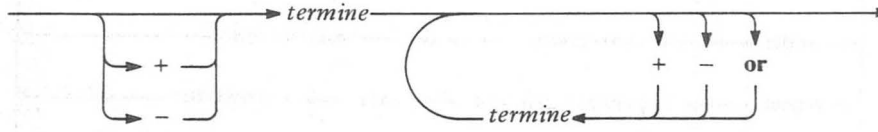


Fig. A16 *espressione-semplice*

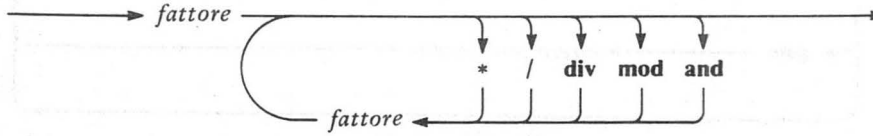


Fig. A17 *termine*

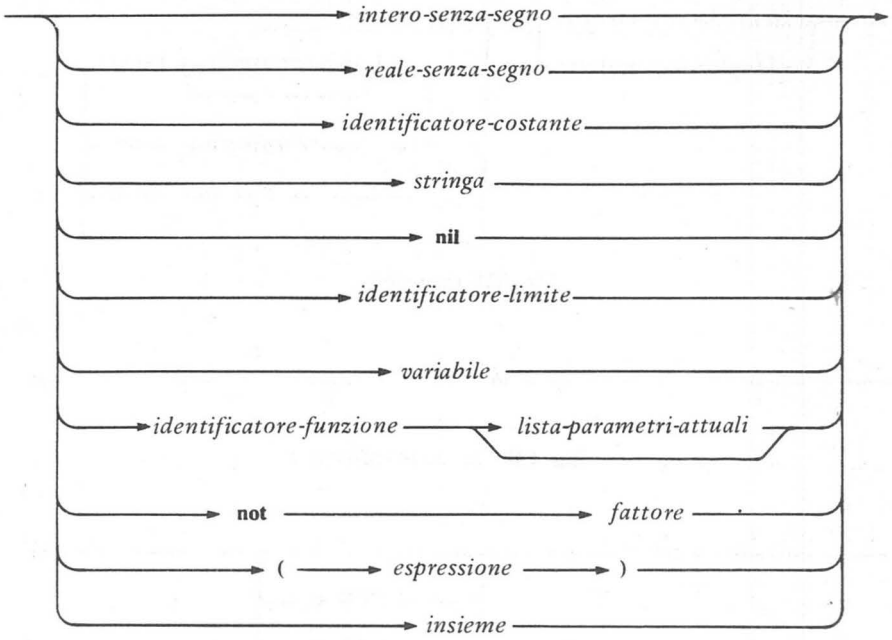


Fig. A18 *fattore*

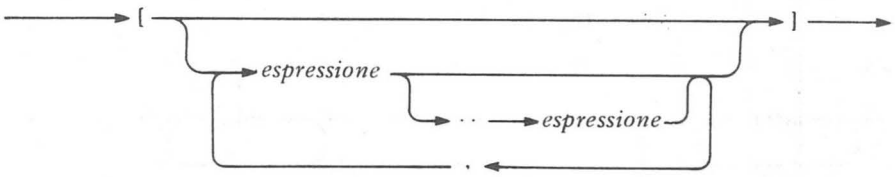


Fig. A19 *insieme*

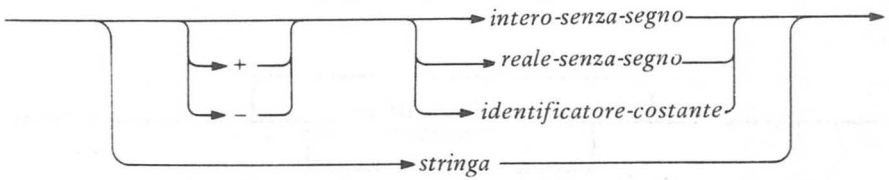


Fig. A20 *costante*

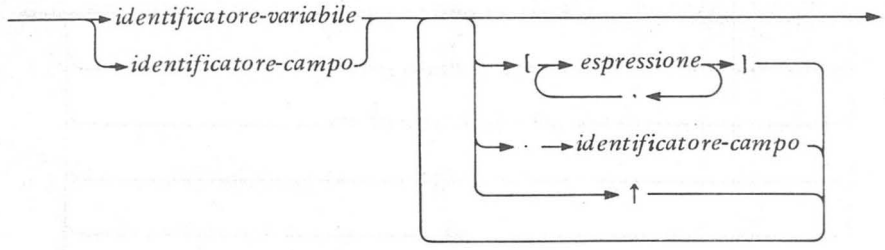


Fig. A21 *variabile*

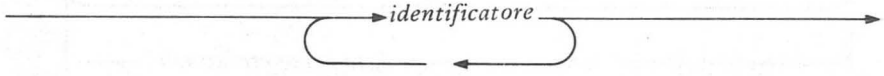


Fig. A22 *lista-identificatori*

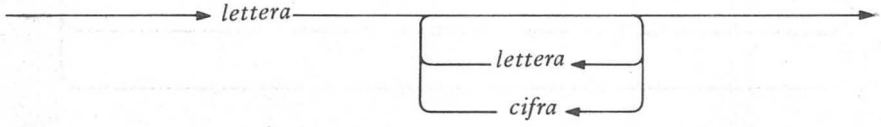


Fig. A23 *identificatore e direttiva*

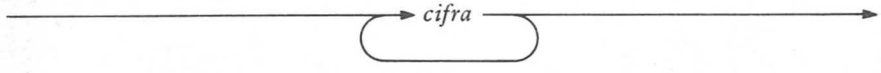


Fig. A24 *intero-senza-segno*

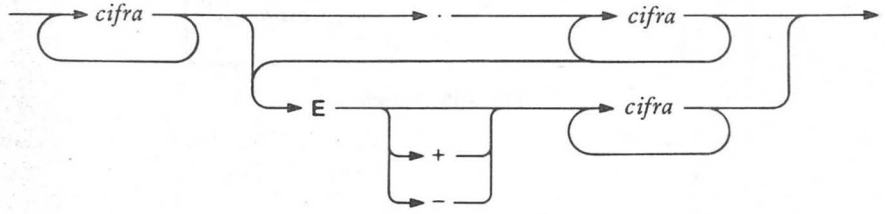


Fig. A25 *reale-senza-segno*

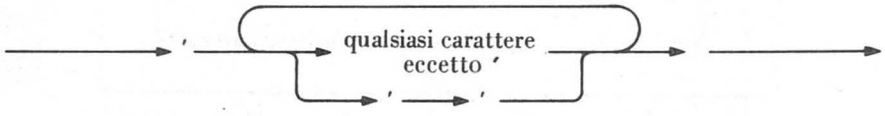


Fig. A26 *stringa*

Appendice 2

Soluzione di esercizi scelti

```
Esercizio 2.1

Interi validi in PASCAL :
      275      6000      27365982      0

Reali validi in PASCAL :
      7.4      6E3      275.0      0.001
      10E-4    0.074E3  0.1E999    0.0620

Numeri non validi in PASCAL :
    3,475     c'e' una virgola
    .1475     non ci sono cifre prima del punto decimale
    275.      non ci sono cifre dopo il punto decimale

I seguenti numeri denotano lo stesso valore :
    0.001     10E-4
Le coppie di numeri
      275     275.0
     6000       6E3
non denotano lo stesso valore perche' il primo numero denota
un valore intero, mentre il secondo un valore reale.

I seguenti numeri non sono validi per molte implementazioni:
     27365982     e' un intero troppo grande
                   (ma e' accettabile come reale)
      0.1E999     e' un reale troppo grande


Esercizio 2.3

La linea
'LEI E' 'LA MOGLIE DI BIANCHI'
contiene due stringhe, 'LEI E' e 'LA MOGLIE DI BIANCHI'.
Le altre linee contengono stringhe valide che denotano le
seguenti sequenze di caratteri :
GINO BIANCHI
sei + uno = sedici
IL CANE DI SUA MOGLIE
''''


Esercizio 3.1

     67 div 8          valida - valore 8
     12 * 2.75         valida - valore 33.0
     67 mod 8          valida - valore 3
     (-3) div 8        valida - valore 0
```

```
succ(66.3)          non valida - succ non e' definito per
                     valori reali
sqr(2.5)            valida - valore 6.25
'A' < 'Z'           valida - valore true
pred('7')           valida - valore '6'
67 / 8              valida - valore 8.375
6.7 div 8           non valida - div deve avere operandi
                                 interi
sqr(10E4)           valida - valore 100E8
sqr(100001)         valida - valore 10000200001
sqrt(6.25)          valida - valore 2.5
sqrt(4)             valida - valore 2.0 (reale)
succ(true)          non valida - true non ha un valore
                                 successore
exp(1.0E60)         valida - un valore reale molto grande!
```

I valori di sqr(100001) ed exp(1.0E60) possono risultare rispettivamente al di fuori dell'intervallo dei valori interi e reali consentiti dall'implementazione.

Esercizio 3.2

```
identificatori_costante  : linesmax  richiestastampa
                           true  singola  doppia
                           tripla

identificatori_tipo      : posizionelinea  spaziatura
                           char

identificatori_variabile : questocar  ultimocar
                           questaposizione
                           spaziaturaattuale
```

Esercizio 3.3

```
const
   pollicipermetro  = 39.37;
   simbolocorrente  = '$'  ;
   gradiperradiante = 57.30;
   limitedivelocita = 60   ; {km orari}
```

Esercizio 3.4

```
type
   eta = 0..120;
   sesso = (maschile,femminile);
   altezza = real; {metri}
   peso = 0..150; {kg}
   statocivile = (nonsposato,sposato,vedovo,divorziato);
```

Esercizio 3.5

```
var
   cameredaletto,sossiorni : 1..10;
   riscaldamentocentrale : (nafta,elettrico,
                            combustibilesolido,gas,nessuno);
   garage : boolean;
```

Esercizio 4.1

Espressioni con parentesi	Valore
6.75 - (12.3 / 3)	2.65
(6 * 11) - (42 div 5)	58
((175 mod 15) div 3) * 65	195
13 + (7 * 5) - ((4 * 5) div 2)	38
((11 mod 4) div 2) <> 0	true
('A' >= 'Z') or (('9' >= '8') and ('A' < 'I'))	true

Esercizio 4.2

I valori finali di X, Y e Z sono 17, 217 e 34, nell'ordine.

Esercizio 4.3

(a) (i <= 1) and (i <= 100) {oppure (0 < i) and (i < 101)}

(b) (J mod k = 0) or (k mod J = 0) {con J e k <> 0}

(c) (y <> 1900) and (y mod 4 = 0)

Esercizio 4.4

(a) si usa una variabile reale t :
```
t := x; x := y; y := t
```

(b)
```
somma := X + Y + Z;
prodotto := X * Y * Z;
media := somma / 3
{assumendo var somma,prodotto : integer; media : real}
```

(c)
```
I := 100 * (ord(H) - ord('0')) + 10 * (ord(T) -
     ord('0')) + (ord(U) - ord('0'))
```

Esercizio 5.1

Valori delle variabili dopo ogni sequenza di istruzioni di input :

	X	Y	Z	I	J	K	c
(a)	12.75	24.7	-33E10				
(b)	12.75	-33E10	0.075				
(c)	12.75	24.7	-33E10	0			
(d)				12	-33	0	
(e)				12	75	24	

Esercizio 5.2

```
writeln ('NUMERO      INTERVALLO      MEDIA');
writeln (cont:5, min:10, '..', max:2, media:11:1)
```

Esercizio 5.3

```
program incrementasomma(input,output);
var somma : real;
begin
   read(somma);
   writeln('somma originale =', somma : 7:2);
   writeln('somma incrementata =', somma * 1.08 : 7:2)
end.
```

Questo programma funziona solo se l'input e' della forma ddd.dd, per cui si puo' modificare nel modo seguente :

```
program incrementasomma(input,output);
var dollari,centesimi : 0..maxint;
    ch : char;
    somma : real;
begin
   read(dollari,ch,centesimi);
   somma := dollari + centesimi / 100;
   writeln('somma originale =', somma : 7:2);
   writeln(somma incrementata =', somma * 1.08 : 7:2)
end.
```

Esercizio 5.4

```
program calcolaresto(input,output);
var costo,offerta : real;
begin
   read(costo,offerta);
   writeln('costo','$':7,costo:6:2);
   writeln('offerta','$':5,offerta:6:2);
   writeln('resto','$':7,offerta-costo:6:2)
end.
```

Esercizio 6.2

```
PROGRAM ESERCIZIO62(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE L'AMMONTARE DI UN PRESTITO E
   STAMPA LA TABELLA DEI PAGAMENTI CONTENENTE IL NUMERO
   DEL PAGAMENTO, L'INTERESSE MENSILE, L'INTERESSE GIA'
   PAGATO ED IL RESIDUO DA PAGARE *)

CONST INTERESSEANNUALE = 8; (* PER CENTO *)

VAR NUMERO : 1..MAXINT;
```

```
    INTERESSE,RESTITUZIONE,PRESTITO,PAGAMENTO,
    RESIDUO,INTERESSEMENSILE : REAL;

BEGIN
   READ(PRESTITO);
   WRITE('AMMONTARE PRESTATO = ',PRESTITO : 12:2);
   PAGAMENTO := PRESTITO / 100;
   WRITELN(' ' : 10,'RESTITUZIONE MENSILE =',PAGAMENTO :
                                                    10:2);
   WRITELN; WRITELN;
   WRITELN(' NUMERO INTERESSE RESTITUZIONE RESIDUO');
   WRITELN;
   INTERESSEMENSILE := INTERESSEANNUALE / 100 / 12;
   NUMERO := 1;
   RESIDUO := PRESTITO;
   REPEAT
      INTERESSE := INTERESSEMENSILE * RESIDUO;
      RESTITUZIONE := PAGAMENTO - INTERESSE;
      RESIDUO := RESIDUO - RESTITUZIONE;
      WRITELN(NUMERO : 7, INTERESSE : 10:2, RESTITUZIONE :
            10:2, RESIDUO : 10:2);
      NUMERO := NUMERO + 1
   UNTIL RESIDUO + RESIDUO * INTERESSEMENSILE <= PAGAMENTO;
   WRITELN; WRITELN;
   WRITELN('ULTIMO PAGAMENTO = ',RESIDUO + RESIDUO *
                                 INTERESSEMENSILE : 10:2)
END.


Esercizio 6.3

PROGRAM ESERCIZIO63(OUTPUT);

(* QUESTO PROGRAMMA STAMPA UNA PIRAMIDE DI NUMERI *)

VAR MAX,SUCCESSIVO : CHAR;

BEGIN
   FOR MAX := '1' TO '9' DO
   BEGIN
      WRITE(' ' : 20 - ORD(MAX) + ORD ('0'));
      FOR SUCCESSIVO := '1' TO MAX DO WRITE(SUCCESSIVO);
      FOR SUCCESSIVO := PRED(MAX) DOWNTO '1' DO
         WRITE(SUCCESSIVO);
      WRITELN
   END
END.


Esercizio 6.6

PROGRAM ESERCIZIO66(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE E STAMPA LINEE DI TESTO,
   DETERMINANDO IL NUMERO DELLE LINEE, DELLE PAROLE E DELLE
```

```
   FRASI IN ESSO CONTENUTE. SI ASSUME CHE LE LETTERE DA A A
   Z SIANO 26 VALORI CONSECUTIVI DEL TIPO CHAR *)

VAR PAROLE,LINEE,FRASI : 0..MAXINT;
    CH : CHAR;

BEGIN
   FRASI := 0; LINEE := 0; PAROLE := 0;
   WHILE NOT EOF(INPUT) DO
   BEGIN
      WHILE NOT EOLN(INPUT) DO
      BEGIN
         READ(CH); WRITE(CH);
         IF (CH >= 'A') AND (CH <= 'Z')
            THEN BEGIN (* SCANSIONE DI UNA PAROLA *)
                    PAROLE := PAROLE + 1;
                    REPEAT
                       READ(CH); WRITE(CH)
                    UNTIL (CH < 'A') OR (CH > 'Z') OR
                                                EOLN(INPUT)
                 END;
         IF CH = '.'
            THEN FRASI := FRASI + 1
      END;
      LINEE := LINEE + 1;
      READLN; WRITELN
   END;
   WRITELN; WRITELN;
   WRITELN('IL TESTO CONTIENE :');
   WRITELN;
   WRITELN(LINEE, ' LINEE');
   WRITELN(FRASI, ' FRASI');
   WRITELN(PAROLE, ' PAROLE')
END.


Esercizio 7.3

PROGRAM ESERCIZIO73(INPUT,OUTPUT);

(* PROGRAMMA PER L'INPUT, LA STAMPA E LA SOMMA DI UNA
   SEQUENZA DI NUMERI OTTALI POSIZIONATI UNO PER LINEA *)

TYPE INTERONONNEGATIVO = 0..MAXINT;

VAR NUMERO,SOMMA : INTERONONNEGATIVO;

   PROCEDURE LEGGIOTTALE(VAR I : INTERONONNEGATIVO);
   VAR CH : CHAR;
       N : INTERONONNEGATIVO;
   BEGIN
      (* SALTA I CARATTERI FINO AD UNA CIFRA OTTALE *)
      REPEAT READ(CH) UNTIL (CH > '0') AND (CH < '7');
      N := 0;
```

```
      REPEAT
         N := N * 8 + (ORD(CH) - ORD('0'));
         READ(CH)
      UNTIL (CH < '0') OR (CH > '7');
      I := N
   END; (* LEGGIOTTALE *)

   PROCEDURE SCRIVIOTTALE(I : INTERONONNEGATIVO);
   VAR FATTORE : INTERONONNEGATIVO;
   BEGIN
      FATTORE := 1;
      WHILE I DIV FATTORE >= 8 DO FATTORE := FATTORE * 8;
      REPEAT
         WRITE (CHR(I DIV FATTORE + ORD('0')));
         I := I MOD FATTORE;
         FATTORE := FATTORE DIV 8
      UNTIL FATTORE = 0
   END; (* SCRIVIOTTALE *)

BEGIN
   WRITELN('SEQUENZA DI NUMERI OTTALI .....');
   WRITELN;
   SOMMA := 0;
   WHILE NOT EOF(INPUT) DO
   BEGIN
      LEGGIOTTALE(NUMERO); READLN;
      SCRIVIOTTALE(NUMERO); WRITELN;
      SOMMA := SOMMA + NUMERO
   END;
   WRITELN;
   WRITE('SOMMA OTTALE DELLA SEQUENZA = ');
   SCRIVIOTTALE(SOMMA); WRITELN
END.
```

Esercizio 7.5

```
PROGRAM ESERCIZIO75(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE UNA CIFRA E STAMPA UNA LISTA DI
   TUTTI I NUMERI COMPRESI TRA 1 E 100 TALI CHE LA
   RAPPRESENTAZIONE DECIMALE DEL NUMERO, IL SUO QUADRATO ED
   IL SUO CUBO CONTENGANO TUTTI LA CIFRA SPECIFICATA. *)

TYPE CIFRE = 0..9;

VAR C : CIFRE;
    BASE,QUADRATO,CUBO : 1..MAXINT;

   FUNCTION PRESENTE (CIFRA : CIFRE; NUMERO : INTEGER) :
                                                  BOOLEAN;
   VAR CIFRASUCCESSIVA : CIFRE;
   BEGIN
      REPEAT
         CIFRASUCCESSIVA := NUMERO MOD 10;
```

```
         NUMERO := NUMERO DIV 10
      UNTIL (CIFRASUCCESSIVA = CIFRA) OR (NUMERO = 0);
      PRESENTE := CIFRASUCCESSIVA = CIFRA
   END; (* PRESENTE *)

BEGIN
   READ(C);
   WRITELN('CIFRASPECIFICATA = ', C : 1);
   WRITELN; WRITELN;
   WRITELN('NUMERO   QUADRATO    CUBO');
   FOR BASE := 1 TO 100 DO
      IF PRESENTE(C,BASE)
         THEN BEGIN
                 QUADRATO := SQR(BASE);
                 IF PRESENTE(C,QUADRATO)
                    THEN BEGIN
                            CUBO := QUADRATO * BASE;
                            IF PRESENTE(C,CUBO)
                               THEN WRITELN(BASE : 4,
                                            QUADRATO : 11,
                                            CUBO : 11)
                         END
              END
END.
```

Esercizio 7.6

```
PROGRAM ESERCIZIO76(OUTPUT);

(* QUESTO PROGRAMMA GENERA LA PIRAMIDE DI NUMERI
   DELL'ESERCIZIO 6.3 USANDO UN CICLO ED UNA PROCEDURA
   RICORSIVA. *)

TYPE INTERVALLO = '1'..'9';

VAR MAX : INTERVALLO;

   PROCEDURE GENERA(M,N : INTERVALLO);
   (* GENERA LA STRINGA DI CIFRE
      M,M+1,...,N-1,N,N-1,...,M+1,M *)
   BEGIN
      IF M = N
         THEN WRITE(M);
         ELSE BEGIN
                 WRITE(M);
                 GENERA(SUCC(M),N);
                 WRITE(M)
              END
   END; (* GENERA *)

BEGIN
   FOR MAX := '1' TO '9' DO
   BEGIN
      WRITE(' ' : 20 - ORD(MAX) + ORD('0'));
```

```
      GENERA('1',MAX);
      WRITELN
   END
END.
```

Esercizio 9.1

```
PROGRAM ESERCIZIO91(INPUT,OUTPUT);

(* VERSIONE ESTESA DELL'ESERCIZIO 6.6 CHE CONTA E STAMPA
   ANCHE IL NUMERO DI OCCORRENZE DELLE LETTERE *)

VAR PAROLE,LINEE,FRASI : 0..MAXINT;
    CH : CHAR;
    CONT : ARRAY ['A'..'Z'] OF 0..MAXINT;
    INDICE : 'A'..'Z';
    ULTIMALETTERA : BOOLEAN;
BEGIN
   FRASI := 0; LINEE := 0; PAROLE := 0;
   FOR INDICE := 'A' TO 'Z' DO CONT[INDICE] := 0;
   WHILE NOT EOF(INPUT) DO
   BEGIN
      ULTIMALETTERA := FALSE;
      WHILE NOT EOLN(INPUT) DO
      BEGIN
         READ(CH); WRITE(CH);
         IF (CH >= 'A') AND (CH <= 'Z')
            THEN BEGIN
                    IF NOT ULTIMALETTERA
                       THEN BEGIN
                               PAROLE := PAROLE + 1;
                               ULTIMALETTERA := TRUE;
                            END;
                    CONT[CH] := CONT[CH] + 1
                 END
            ELSE BEGIN
                    IF CH = '.'
                       THEN FRASI := FRASI + 1;
                    ULTIMALETTERA := FALSE
                 END
      END;
      LINEE := LINEE + 1;
      READLN; WRITELN
   END;
   WRITELN; WRITELN;
   WRITELN('IL TESTO CONTIENE :');
   WRITELN;
   WRITELN(LINEE,'LINEE');
   WRITELN(FRASI,'FRASI');
   WRITELN(PAROLE,'PAROLE');
   WRITELN; WRITELN;
   WRITELN('OCCORRENZE DELLE LETTERE'); WRITELN;
   FOR INDICE := 'A' TO 'Z' DO
      WRITELN(INDICE,' :',CONT[INDICE] : 5)
END.
```

Esercizio 9.3

```
PROGRAM ESERCIZIO93(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE UN TESTO IN INPUT E STAMPA IL
   NUMERO DI OCCORRENZE DELLE COPPIE DI LETTERE ADIACENTI
   CHE COMPAIONO NEL TESTO. SI ASSUME CHE LE LETTERE DA A
   A Z SIANO 26 VALORI CONSECUTIVI DEL TIPO CHAR *)

TYPE LETTERA = 'A'..'Z';

VAR CONT : ARRAY [LETTERA,LETTERA] OF 0..MAXINT;
    PRIMA,SECONDA : LETTERA;
    QUESTOCAR,ULTIMOCAR : CHAR;
    ULTIMALETTERA : BOOLEAN;

BEGIN
   FOR PRIMA := 'A' TO 'Z' DO
      FOR SECONDA := 'A' TO 'Z' DO
         CONT[PRIMA,SECONDA] := 0;
   WHILE NOT EOF(INPUT) DO
   BEGIN
      ULTIMALETTERA := FALSE; (* ALL'INIZIO DI UNA LINEA *)
      WHILE NOT EOLN(INPUT) DO
      BEGIN
         READ(QUESTOCAR); WRITE(QUESTOCAR);
         IF (QUESTOCAR >= 'A') AND (QUESTOCAR <= 'Z')
            THEN IF ULTIMALETTERA
                    THEN CONT[ULTIMOCAR,QUESTOCAR] :=
                            CONT[ULTIMOCAR,QUESTOCAR] + 1;
                    ELSE ULTIMALETTERA := TRUE
            ELSE ULTIMALETTERA := FALSE;
         ULTIMOCAR := QUESTOCAR
      END;
      READLN; WRITELN
   END;
   WRITELN; WRITELN;
   WRITELN('OCCORRENZE DELLE COPPIE DI LETTERE ....');
   FOR PRIMA := 'A' TO 'Z' DO
      FOR SECONDA := 'A' TO 'Z' DO
         IF CONT[PRIMA,SECONDA] <> 0
           THEN
           WRITELN(PRIMA,SECONDA,'-',CONT[PRIMA,SECONDA]:4)
END.
```

Esercizio 9.4

```
PROGRAM ESERCIZIO94(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE UNA SEQUENZA DI COPPIE DI PAROLE
   E STAMPA OGNI COPPIA IN ORDINE ALFABETICO *)

CONST LUNGHEZZAPAROLA = 16;
      PAROLAVUOTA = '                ';
      (* SPAZI PARI A LUNGHEZZAPAROLA *)
```

```
TYPE PAROLA = PACKED ARRAY [1..LUNGHEZZAPAROLA] OF CHAR;

VAR PAROLA1,PAROLA2 : PAROLA;

   PROCEDURE LEGGI(VAR LAPAROLA : PAROLA);
   VAR I : 0..LUNGHEZZAPAROLA;
       CH : CHAR;
   BEGIN
      LAPAROLA := PAROLAVUOTA;
      REPEAT READ(CH) UNTIL CH <> ' ';
      I := 0;
      REPEAT
         I := I + 1;
         LAPAROLA[I] := CH;
         READ(CH)
      UNTIL (CH = ' ') OR EOLN(INPUT);
      IF EOLN(INPUT) AND (CH <> ' ')
         THEN LAPAROLA[I+1] := CH
   END; (* LEGGI *)

BEGIN
   WHILE NOT EOF(INPUT) DO
   BEGIN
      LEGGI(PAROLA1);
      LEGGI(PAROLA2);
      IF PAROLA1 > PAROLA2
         THEN WRITE(PAROLA2,' ',PAROLA1)
         ELSE WRITE(PAROLA1,' ',PAROLA2);
      READLN; WRITELN
   END
END.


Esercizio 10.2

PROGRAM ESERCIZIO102(INPUT,OUTPUT);

(* IL PROGRAMMA LEGGE QUATTRO COPPIE DI VALORI CHE
   RAPPRESENTANO I VERTICI DI UN QUADRILATERO ORDINATI IN
   SENSO ORARIO E DETERMINA SE E' UN QUADRATO, UN RETTANGOLO
   O ALTRO. *)

TYPE COORDINATE = RECORD
                     X,Y : 1..100;
                  END;
     PUNTO = (A,B,C,D);

VAR VERTICE : ARRAY [PUNTO] OF COORDINATE;
    P : PUNTO;

   FUNCTION LUNGHEZZAALQUADRATO(P1,P2 : PUNTO) : INTEGER;
   BEGIN
      LUNGHEZZAALQUADRATO := SQR(VERTICE[P1].X -
                                 VERTICE[P2].X) +
                             SQR(VERTICE[P1].Y -
```

```
                                        VERTICE[F2].Y)
   END (* LUNGHEZZAALQUADRATO *)

BEGIN
   WRITE('I PUNTI .... ');
   FOR P := A TO D DO
      WITH VERTICE[P] DO
      BEGIN
         READ(X,Y);
         WRITE('(', X : 3, Y : 3, ')')
      END;
   WRITE(' RAPPRESENTANO UN ');
   IF (LUNGHEZZAALQUADRATO(A,B) = LUNGHEZZAALQUADRATO(C,D))
   AND (LUNGHEZZAALQUADRATO(B,C) = LUNGHEZZAALQUADRATO(D,A))
   AND (LUNGHEZZAALQUADRATO(A,C) = LUNGHEZZAALQUADRATO(B,D))
     THEN
      (* LATI OPPOSTI E DIAGONALI SONO UGUALI *)
      IF LUNGHEZZAALQUADRATO(A,B) = LUNGHEZZAALQUADRATO(B,C)

         THEN WRITE('QUADRATO')
         ELSE WRITE('RETTANGOLO')
     ELSE WRITE('QUADRILATERO');
   WRITELN; WRITELN
END.


Esercizio 10.4

PROGRAM ESERCIZIO104(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE UNA SEQUENZA DI DESCRIZIONI DI
   FIGURE GOEMETRICHE E STAMPA LA DESCRIZIONE DI QUELLA DI
   AREA MAGGIORE *)

CONST PI = 3.14159;

TYPE TIPOFIGURA = (CERCHIO,RETTANGOLO,QUADRATO,TRIANGOLO);
     FIGURA = RECORD
                 CASE FORMA : TIPOFIGURA OF
                    CERCHIO : (RAGGIO : REAL);
                    RETTANGOLO : (LATO1,LATO2 : REAL);
                    QUADRATO : (LATO : REAL);
                    TRIANGOLO : (A,B,C : REAL)
              END;

VAR QUESTAFIGURA,MAXFIGURA : FIGURA;
    AREAFIGURA,MAXAREA : REAL;

   PROCEDURA LEGGIFIGURA(VAR UNAFIGURA : FIGURA);
   VAR CH : CHAR;
   BEGIN
       WITH UNAFIGURA DO
       BEGIN
         READ(CH);
```

```
    IF CH = 'C'
      THEN BEGIN
             FORMA := CERCHIO;
             READ(RAGGIO)
           END
      ELSE IF CH = 'R'
             THEN BEGIN
                    FORMA := RETTANGOLO;
                    READ(LATO1,LATO2)
                  END
             ELSE IF CH = 'Q'
                    THEN BEGIN
                           FORMA := QUADRATO;
                           READ(LATO)
                         END
                    ELSE IF CH = 'T'
                           THEN
                             BEGIN
                               FORMA := TRIANGOLO;
                               READ(A,B,C)
                             END
                           ELSE
                             BEGIN
                               WRITELN('FIGURA IGNOTA');
                               FORMA := CERCHIO;
                               RAGGIO := 0.0
                             END
  END
END; (* LEGGIFIGURA *)

FUNCTION AREA(UNAFIGURA : FIGURA) : REAL;
VAR S : REAL;
BEGIN
  WITH UNAFIGURA DO
    CASE FORMA OF
      CERCHIO : AREA := PI * SQR(RAGGIO);
      RETTANGOLO : AREA := LATO1 * LATO2;
      QUADRATO : AREA := SQR(LATO);
      TRIANGOLO : BEGIN
                    S := (A + B + C) / 2;
                    AREA := SQRT(S*(S-A)*(S-B)*(S-C))
                  END
    END
END; (* AREA *)

PROCEDURE SCRIVIFIGURA(UNAFIGURA : FIGURA);
BEGIN
  WITH UNAFIGURA DO
    CASE FORMA OF
      CERCHIO : WRITE('UN CERCHIO DI RAGGIO',RAGGIO:6:1);
      RETTANGOLO : WRITE('UN RETTANGOLO DI LATI',
                                    LATO1:6:1,LATO2:6:1);
      QUADRATO : WRITE('UN QUADRATO DI LATO',LATO:6:1);
      TRIANGOLO : WRITE('UN TRIANGOLO DI LATI',
                                    A:6:1,B:6:1,C:6:1)
    END
  END; (* SCRIVIFIGURA *)
```

```
BEGIN
   MAXAREA := 0;
   WRITELN('SEQUENZA FIGURE ....');
   WRITELN; WRITELN;
   WHILE NOT EOF(INPUT) DO
   BEGIN
      LEGGIFIGURA(QUESTAFIGURA);
      AREAFIGURA := AREA(QUESTAFIGURA);
      WRITE('AREA =',AREAFIGURA:10:2,' DI ');
      SCRIVIFIGURA(QUESTAFIGURA);
      IF AREAFIGURA > MAXAREA
         THEN BEGIN
                 MAXAREA := AREAFIGURA;
                 MAXFIGURA := QUESTAFIGURA
              END;
      READLN; WRITELN; WRITELN
   END;
   WRITE( LA FIGURA CON AREA MAGGIORE E'' ');
   SCRIVIFIGURA(MAXFIGURA);
   WRITELN(' E LA SUA AREA E''',MAXAREA:10:2)
END.
```

Esercizio 11.3

```
PROGRAM ESERCIZIO113(INPUT,OUTPUT);

(* IL PROGRAMMA LEGGE DUE FRASI, CIASCUNA TERMINANTE CON UN
   PUNTO E STAMPA UNA LISTA DI TUTTE LE LETTERE CHE
   COMPAIONO IN ENTRAMBE LE FRASI *)

TYPE CARATTERI = SET OF CHAR;

VAR CARATTERI1,CARATTERI2,LETTERECOMUNI : CARATTERI;
    LETTERA : CHAR;

   PROCEDURE SCANSIONEFRASE(VAR CARATTERIUSATI : CARATTERI);
   VAR CH : CHAR;
   BEGIN
      CARATTERIUSATI := [];
      REPEAT
         READ(CH);
         CARATTERIUSATI := CARATTERIUSATI + [CH]
      UNTIL CH = '.'
   END; (* SCANSIONEFRASE *)

BEGIN
   SCANSIONEFRASE(CARATTERI1);
   SCANSIONEFRASE(CARATTERI2);
   LETTERECOMUNI := CARATTERI1 * CARATTERI2 * ['A'..'Z'];
   WRITE('LE LETTERE COMUNI SONO :');
   FOR LETTERA := 'A' TO 'Z' DO
      IF LETTERA IN LETTERECOMUNI
         THEN WRITE(LETTERA :2);
   WRITELN
END.
```

```
Esercizio 11.4

PROGRAM ESERCIZIO114(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE LE SCHEDE DI ISCRIZIONE DEGLI
   STUDENTI E STAMPA UNA LISTA DEGLI STUDENTI PER OGNUNA
   DELLE DIECI MATERIE *)

CONST LUNGHEZZANOME = 20;
      MAXNUMSTUDENTI = 100;

TYPE MATERIA = (ARTE,BIOLOGIA,CHIMICA,FISICA,FRANCESE,
                GEOGRAFIA,INGLESE,MATEMATICA,STORIA,TEDESCO);
     DATISTUDENTE = RECORD
                      NOME : PACKED ARRAY [1..LUNGHEZZANOME]
                                                    OF CHAR;
                      CORSI : SET OF MATERIA
                    END;

VAR QUESTAMATERIA : MATERIA;
    N,NUMEROSTUDENTI : 0..MAXNUMSTUDENTI;
    STUDENTI : ARRAY [1..MAXNUMSTUDENTI] OF DATISTUDENTE;

   PROCEDURE LEGGISCHEDA(VAR S : DATISTUDENTE);
   VAR CH : CHAR;
       QUESTAMATERIA : MATERIA;
       I : 1..LUNGHEZZANOME;
   BEGIN
      WITH S DO
      BEGIN
         FOR I := 1 TO LUNGHEZZANOME DO READ(NOME[I]);
         CORSI := [];
         FOR QUESTAMATERIA := ARTE TO TEDESCO DO
         BEGIN
            READ(CH);
            IF CH = 'X'
               THEN CORSI := CORSI + [QUESTAMATERIA]
         END
      END;
      READLN
   END; (* LEGGISCHEDA *)

BEGIN
   N := 0;
   REPEAT
      N := N + 1;
      LEGGISCHEDA(STUDENTI[N])
   UNTIL EOF(INPUT);
   NUMEROSTUDENTI := N;
   FOR QUESTAMATERIA := ARTE TO TEDESCO DO
   BEGIN
      CASE QUESTAMATERIA OF
         ARTE       : WRITELN('***ARTE***');
         BIOLOGIA   : WRITELN('***BIOLOGIA***');
         CHIMICA    : WRITELN('***CHIMICA***');
```

```
            FISICA      : WRITELN('***FISICA***');
            FRANCESE    : WRITELN('***FRANCESE***');
            GEOGRAFIA   : WRITELN('***GEOGRAFIA***');
            INGLESE     : WRITELN('***INGLESE***');
            MATEMATICA  : WRITELN('***MATEMATICA***');
            STORIA      : WRITELN('***STORIA***');
            TEDESCO     : WRITELN('***TEDESCO***')
         END;
         WRITELN;
         FOR N := 1 TO NUMEROSTUDENTI DO
            WITH STUDENTI[N] DO
               IF QUESTAMATERIA IN CORSI THEN WRITELN(NOME);
         WRITELN; WRITELN
   END
END.
```

Esercizio 11.5

```
PROGRAM ESERCIZIO115(INPUT,OUTPUT);

(* QUESTO PROGRAMMA LEGGE LA LUNGHEZZA DI UNA BARRA ED UNA
   LISTA DI ORDINI DI PEZZI DELLA BARRA E DECIDE (CON IL
   METODO RICORSIVO "TRIAL AND ERROR") L'INSIEME ORDINATO
   DELLE LUNGHEZZE DEI PEZZI CHE DEVONO ESSERE TAGLIATI
   PER MINIMIZZARE GLI SPRECHI *)

CONST MAXLISTAORDINI = 20;

TYPE ORDINI = 1..MAXLISTAORDINI;


 VAR LISTAORDINI : ORDINI;
     INSIEMEMIGLIORE : INSIEMEORDINI;
     LUNGHEZZAMIGLIORE,L : 0..MAXINT;
     LUNGHEZZA : ARRAY [ORDINI] OF 1..MAXINT;
     I : ORDINI;

 (* GLI INSIEMI VENGONO USATI PER DESCRIVERE LA SOLUZIONE DI
    TENTATIVO, LA SOLUZIONE MIGLIORE ED I COMPONENTI
    POTENZIALI. LA LUNGHEZZA DEGLI ORDINI E' MEMORIZZATA
    NELL'ARRAY LUNGHEZZA. INSIEMEMIGLIORE CONTIENE LA MIGLIOR
    SOLUZIONE TROVATA AL PUNTO CORRENTE. LUNGHEZZAMIGLIORE
    DENOTA LA LUNGHEZZA TOTALE DEGLI ORDINI IN
    INSIEMEMIGLIORE. *)

    PROCEDURE CONSIDERA(LUNGHEZZATENTATIVO : INTEGER;
                        ORDINISCELTI,ORDINIRESTANTI :
                                          INSIEMEORDINI);

    (* LUNGHEZZATENTATIVO DENOTA LA LUNGHEZZA TOTALE DEGLI
       ORDINI CHE SI TROVANO CORRENTEMENTE NELL'INSIEME
       ORDINISCELTI. ORDINISCELTI E' UN INSIEME I CUI
       ELEMENTI SONO GLI INDICI DI LUNGHEZZA DEGLI ORDINI
       PER LA CORRENTE SOLUZIONE DI TENTATIVO.
```

```
    ORDINIRESTANTI E' UN INSIEME I CUI ELEMENTI SONO
    GLI INDICI DI LUNGHEZZA DEGLI ORDINI ANCORA DA
    CONSIDERARE NELLA SOLUZIONE DI TENTATIVO *)

VAR X : 0..MAXLISTAORDINI;

BEGIN
  IF LUNGHEZZATENTATIVO <= L
    THEN
      BEGIN
        IF LUNGHEZZATENTATIVO > LUNGHEZZAMIGLIORE
          THEN
            BEGIN
              LUNGHEZZAMIGLIORE := LUNGHEZZATENTATIVO;
              INSIEMEMIGLIORE := ORDINISCELTI
            END;
        X := 0;
        WHILE ORDINIRESTANTI <> [] DO
        BEGIN
          REPEAT X := X + 1 UNTIL X IN ORDINIRESTANTI;
          ORDINIRESTANTI := ORDINIRESTANTI - [X];
          CONSIDERA(LUNGHEZZATENTATIVO + LUNGHEZZA[X],
                    ORDINISCELTI + [X],ORDINIRESTANTI)
        END
      END
END; (* CONSIDERA *)

BEGIN
   READ(L);
   WRITELN('LA BARRA PRODOTTA E'' LUNGA',L:5,'MM');
   WRITELN;
   WRITELN('GLI ORDINI SONO');
   READ(LISTAORDINI);
   FOR I := 1 TO LISTAORDINI DO
   BEGIN
      READ(LUNGHEZZA[I]);
      WRITELN(LUNGHEZZA[I],' MM')
   END;
   LUNGHEZZAMIGLIORE := 0;
   INSIEMEMIGLIORE := [];

   (* INIZIALMENTE L'INSIEME DI TENTATIVO E' VUOTO E TUTTI
      GLI ORDINI POSSONO ESSERE CONSIDERATI DALLA SOLUZIONE
      DI TENTATIVO *)

   CONSIDERA(0,[],[1..LISTAORDINI]);
   WRITELN;
   WRITELN('INSIEME OTTIMO DI ORDINI');
   FOR I := 1 TO LISTAORDINI DO
      IF I IN INSIEMEMIGLIORE
         THEN WRITELN(LUNGHEZZA[I]:6,' MM');
   WRITELN;
   WRITELN('SCARTO =',L - LUNGHEZZAMIGLIORE:4,' MM')
END.
```

```
{esercizio 12.1

PROGRAM ESERCIZIO121(INPUT,CLASSE1);

(* CREAZIONE DEL FILE CLASSE1 CONTENENTE I VOTI DEGLI
   STUDENTI *)

TYPE DATISTUDENTE = RECORD
                       NOME : PACKED ARRAY [1..30] OF CHAR;
                       VOTO : ARRAY [1..5] OF 0..10;
                       TOTALE : 0..50
                    END;
     FILESTUDENTI = FILE OF DATISTUDENTE;

VAR CLASSE1 : FILESTUDENTI;

   PROCEDURE LEGGISCHEDA(VAR S : DATISTUDENTE);
   VAR LUNGHEZZANOME : 0..30;
       MATERIA : 1..5;
       CH : CHAR;
   BEGIN
        WITH S DO
        BEGIN
           LUNGHEZZANOME := 0;
           REPEAT
              LUNGHEZZANOME := LUNGHEZZANOME + 1;
              READ(CH);
              NOME[LUNGHEZZANOME] := CH
           UNTIL CH = '.';
           WHILE LUNGHEZZANOME < 30 DO
           BEGIN
              LUNGHEZZANOME := LUNGHEZZANOME + 1;
              NOME[LUNGHEZZANOME] := ' '
           END;
           TOTALE := 0;
           FOR MATERIA := 1 TO 5 DO
           BEGIN
              READ(VOTO[MATERIA]);
              TOTALE := TOTALE + VOTO[MATERIA]
           END
        END;
        READLN
   END; (* LEGGISCHEDA *)

BEGIN
   REWRITE(CLASSE1);
   WHILE NOT EOF(INPUT) DO
   BEGIN
      LEGGISCHEDA(CLASSE1^);
      PUT(CLASSE1)
   END
END.
```

```
Esercizio 12.3

PROGRAM ESERCIZIO123(OUTPUT,CLASSE1);

(* TABULAZIONE DEI RECORD DEL FILE CLASSE1 *)

TYPE DATISTUDENTE = RECORD
                       NOME : PACKED ARRAY [1..30] OF CHAR;
                       VOTO : ARRAY [1..5] OF 0..10;
                       TOTALE : 0..50
                    END;
     FILESTUDENTI = FILE OF DATISTUDENTE;

VAR CLASSE1 : FILESTUDENTI;

   PROCEDURE STAMPADATI(S : DATISTUDENTE);
   VAR STELLA : 1..20;
   BEGIN
         WITH S DO
         BEGIN
            WRITE(NOME,TOTALE:10,'  ');
            FOR STELLA := 1 TO ROUND(0.4 * TOTALE) DO
               WRITE('*')
         END;
         WRITELN
      END; (* STAMPADATI *)

   BEGIN
      RESET(CLASSE1);
      WHILE NOT EOF(CLASSE1) DO
      BEGIN
         STAMPADATI(CLASSE1^);
         GET(CLASSE1)
      END
   END.


   Esercizio 12.4

   PROGRAM ESERCIZIO124(CLASSE1,CLASSE2,CLASSI);

   (* FUSIONE DEI FILE CLASSE1 E CLASSE2 CONTENENTI I VOTI
      ORDINATI DEGLI STUDENTI PER OTTENERE UN SINGOLO FILE
      ORDINATO, CLASSI. *)

   TYPE DATISTUDENTE = RECORD
                          NOME : PACKED ARRAY [1..30] OF CHAR;
                          VOTO : ARRAY [1..5] OF 0..10;
                          TOTALE : 0..50
                       END;
        FILESTUDENTI = FILE OF DATISTUDENTE;

   VAR CLASSE1,CLASSE2,CLASSI : FILESTUDENTI;
       FINEDIUNODEIDUEFILE : BOOLEAN;
```

```
    BEGIN
       RESET(CLASSE1); RESET(CLASSE2); REWRITE(CLASSI);
       FINEDIUNODEIDUEFILE := EOF(CLASSE1) OR EOF(CLASSE2);
       WHILE NOT FINEDIUNODEIDUEFILE DO
       BEGIN
          IF CLASSE1^.NOME < CLASSE2^.NOME
             THEN BEGIN
                     CLASSI^ := CLASSE1^; GET(CLASSE1);
                     FINEDIUNODEIDUEFILE := EOF(CLASSE1)
                  END
             ELSE BEGIN
                     CLASSI^ := CLASSE2^; GET(CLASSE2);
                     FINEDIUNODEIDUEFILE := EOF(CLASSE2)
                  END;
          PUT(CLASSI)
       END;
       WHILE NOT EOF(CLASSE1) DO
       BEGIN
          CLASSI^ := CLASSE1^;
          PUT(CLASSI);
          GET(CLASSE1)
       END;
       WHILE NOT EOF(CLASSE2) DO
       BEGIN
          CLASSI^ := CLASSE2^;
          PUT(CLASSI);
          GET(CLASSE2)
       END
    END.

Esercizio 12.6

PROGRAM ESERCIZIO126(OUTPUT,PAGINA1,PAGINA2);

(* QUESTO PROGRAMMA LEGGE I FILE TESTO PAGINA1 E PAGINA2,
   E LI STAMPA AFFIANCATI *)

CONST MEZZAMAXLINEA = 30;

TYPE INTERVALLOTESTO = 0..MEZZAMAXLINEA;

VAR PAGINA1,PAGINA2 : TEXT;
    LUNGHEZZA : INTERVALLOTESTO;

   PROCEDURE STAMPALINEADA(VAR F : TEXT; VAR LUNGHEZZA :
                                         INTERVALLOTESTO);
   VAR CH : CHAR;
       CONT : INTERVALLOTESTO;
   BEGIN
      CONT := 0;
      WHILE NOT EOLN(F) DO
      BEGIN
         CONT := CONT + 1;
         READ(F,CH);
```

```
            WRITE(CH)
         END;
         READLN(F);
         LUNGHEZZA := CONT
      END; (* STAMPALINEADA *)

BEGIN
   RESET(PAGINA1); RESET(PAGINA2);
   WRITELN(' ':MEZZAMAXLINEA DIV 2 - 4,'PAGINA 1',
           ' ':MEZZAMAXLINEA DIV 2 - 8,'PAGINA 2');
   WHILE NOT (EOF(PAGINA1) OR EOF(PAGINA2)) DO
   BEGIN
      STAMPALINEADA(PAGINA1,LUNGHEZZA);
      IF LUNGHEZZA < MEZZAMAXLINEA
         THEN WRITE(' ':MEZZAMAXLINEA - LUNGHEZZA);
      STAMPALINEADA(PAGINA2,LUNGHEZZA);
      WRITELN
   END;
   WHILE NOT EOF(PAGINA1) DO
   BEGIN
      STAMPALINEADA(PAGINA1,LUNGHEZZA);
      WRITELN
   END;
   WHILE NOT EOF(PAGINA2) DO
   BEGIN
      WRITE(' ':MEZZAMAXLINEA);
      STAMPALINEADA(PAGINA2,LUNGHEZZA);
      WRITELN
   END
END.


Esercizio 13.2

PROGRAM ESERCIZIO132(OUTPUT,IMPIEGATI);

(* QUESTO PROGRAMMA LEGGE UN FILE CONTENENTE IL NUMERO DI
   TELEFONO DEGLI IMPIEGATI E STAMPA UNA LISTA DI RECORD
   CHE SPECIFICANO GLI UTENTI DI OGNI TELEFONO *)

CONST LUNGHEZZANOME = 16;

TYPE LETTERE = PACKED ARRAY [1..LUNGHEZZANOME] OF CHAR;
     NUMERO = 0..MAXINT;
     PUNTATORELISTA = ^OCCUPANTE;
     OCCUPANTE = RECORD
                    NOME : LETTERE;
                    SUCCESSIVO : PUNTATORELISTA
                 END;
     LISTAOCCUPANTI = RECORD
                         PRIMO,ULTIMO : PUNTATORELISTA
                      END;
     PUNTATORESTANZA = ^RECORDSTANZA;
     RECORDSTANZA = RECORD
                       TELEFONO : NUMERO;
```

```
                        OCCUPANTI : LISTAOCCUPANTI;
                        SUCCESSIVA : PUNTATORESTANZA
                    END;

VAR IMPIEGATI : FILE OF RECORD
                            NOME : LETTERE;
                            TELEFONO : NUMERO
                        END;
    LISTASTANZE,STANZA : PUNTATORESTANZA;

  PROCEDURE CERCASTANZA(NUMEROCERCATO : NUMERO;
                        VAR STANZATROVATA : PUNTATORESTANZA);
  VAR QUESTASTANZA,STANZAPRECEDENTE : PUNTATORESTANZA;
      POSIZIONETROVATA : BOOLEAN;

    PROCEDURE INSERISCISTANZA;
    VAR NUOVASTANZA : PUNTATORESTANZA;
    BEGIN
      NEW(NUOVASTANZA);
      WITH NUOVASTANZA DO
      BEGIN
        TELEFONO := NUMEROCERCATO;
        WITH OCCUPANTI DO
        BEGIN
          PRIMO := NIL;
          ULTIMO := NIL
        END;
        SUCCESSIVA := QUESTASTANZA
      END;
      IF STANZAPRECEDENTE = NIL
        THEN LISTASTANZE := NUOVASTANZA
        ELSE STANZAPRECEDENTE^.SUCCESSIVA := NUOVASTANZA;
      STANZATROVATA:= NUOVASTANZA
    END; (* INSERISCISTANZA *)

  BEGIN
    QUESTASTANZA := LISTASTANZE;
    STANZAPRECEDENTE := NIL;
    POSIZIONETROVATA := FALSE;
    WHILE NOT POSIZIONETROVATA AND (QUESTASTANZA <> NIL)DO
      IF QUESTASTANZA^.TELEFONO >= NUMEROCERCATO
        THEN POSIZIONETROVATA := TRUE
        ELSE BEGIN
               STANZAPRECEDENTE := QUESTASTANZA;
               QUESTASTANZA := QUESTASTANZA^.SUCCESSIVA
             END;
    IF POSIZIONETROVATA
      THEN IF QUESTASTANZA^.TELEFONO = NUMEROCERCATO
             THEN STANZATROVATA := QUESTASTANZA
             ELSE INSERISCISTANZA
      ELSE INSERISCISTANZA
  END; (* CERCASTANZA *)

  PROCEDURE AGGIUNGIIMPIEGATO(VAR LISTA : LISTAOCCUPANTI);
  VAR QUESTOOCCUPANTE : PUNTATORELISTA;
```

```
BEGIN
   NEW(QUESTOOCCUPANTE);
   WITH QUESTOOCCUPANTE^ DO
   BEGIN
      NOME := IMPIEGATI^.NOME;
      SUCCESSIVO := NIL
   END;
   WITH LISTA DO
   BEGIN
      IF PRIMO = NIL
         THEN PRIMO := QUESTOOCCUPANTE
         ELSE ULTIMO^.SUCCESSIVO := QUESTOOCCUPANTE;
      ULTIMO := QUESTOOCCUPANTE
   END
END; (* AGGIUNGIIMPIEGATO *)

PROCEDURE STAMPAELENCO;
VAR QUESTASTANZA,STANZASUCCESSIVA : PUNTATORESTANZA;

   PROCEDURE STAMPAOCCUPANTI(TELEFONO : NUMERO;
                             OCCUPANTI : LISTAOCCUPANTI);
   VAR PRIMALINEA : BOOLEAN;
       QUESTOOCCUPANTE,OCCUPANTESUCCESSIVO :
                                          PUNTATORELISTA;
   BEGIN (* STAMPAOCCUPANTI *)
      WRITELN;
      PRIMALINEA := TRUE;
      OCCUPANTESUCCESSIVO := OCCUPANTI.PRIMO;
      REPEAT
         QUESTOOCCUPANTE := OCCUPANTESUCCESSIVO;
         IF PRIMALINEA
            THEN BEGIN
                    WRITE(TELEFONO : 6,'  ');
                    PRIMALINEA := FALSE
                 END
            ELSE WRITE(' ' : 8);
         WRITELN(QUESTOOCCUPANTE^.NOME);
         OCCUPANTESUCCESSIVO :=
                           QUESTOOCCUPANTE^.SUCCESSIVO;
         DISPOSE(QUESTOOCCUPANTE)
      UNTIL OCCUPANTESUCCESSIVO = NIL
   END; (* STAMPAOCCUPANTI *)

BEGIN
   PAGE;
   WRITELN('LISTA DELLE STANZE');
   WRITELN('----------');
   WRITELN; WRITELN;
   STANZASUCCESSIVA := LISTASTANZE;
   WHILE STANZASUCCESSIVA <> NIL DO
   BEGIN
      QUESTASTANZA := STANZASUCCESSIVA;
      WITH QUESTASTANZA^ DO
         STAMPAOCCUPANTI(TELEFONO,OCCUPANTI);
      STANZASUCCESSIVA := QUESTASTANZA^.SUCCESSIVA;
```

```
            DISPOSE(QUESTASTANZA)
         END
      END; (* STAMPAELENCO *)

BEGIN (* PROGRAMMA PRINCIPALE *)
   RESET(IMPIEGATI);
   LISTASTANZE := NIL;
   WHILE NOT EOF(IMPIEGATI) DO
   BEGIN
      CERCASTANZA(IMPIEGATI^.TELEFONO,STANZA);
      AGGIUNGIIMPIEGATO(STANZA^.OCCUPANTI);
      GET(IMPIEGATI)
   END;
   STAMPAELENCO
END.
```

Indice Analitico

Questo indice fornisce una lista esauriente delle parole e dei termini usati nel libro. Quelli indicati in *corsivo* sono identificatori standard in PASCAL, o termini standard usati nella sua definizione.

Il numero di pagina dell'occorrenza che definisce una parola o un termine è indicato in **grassetto,** mentre le altre occorrente significative in carattere normale.

Procedure e funzioni predefinite in PASCAL

Finito di stampare ottobre 1985 — Tipografia E.S.A. Editrice - s.r.l.
(00184) Roma — Via della Polveriera, 13 - Tel. (06) 465197