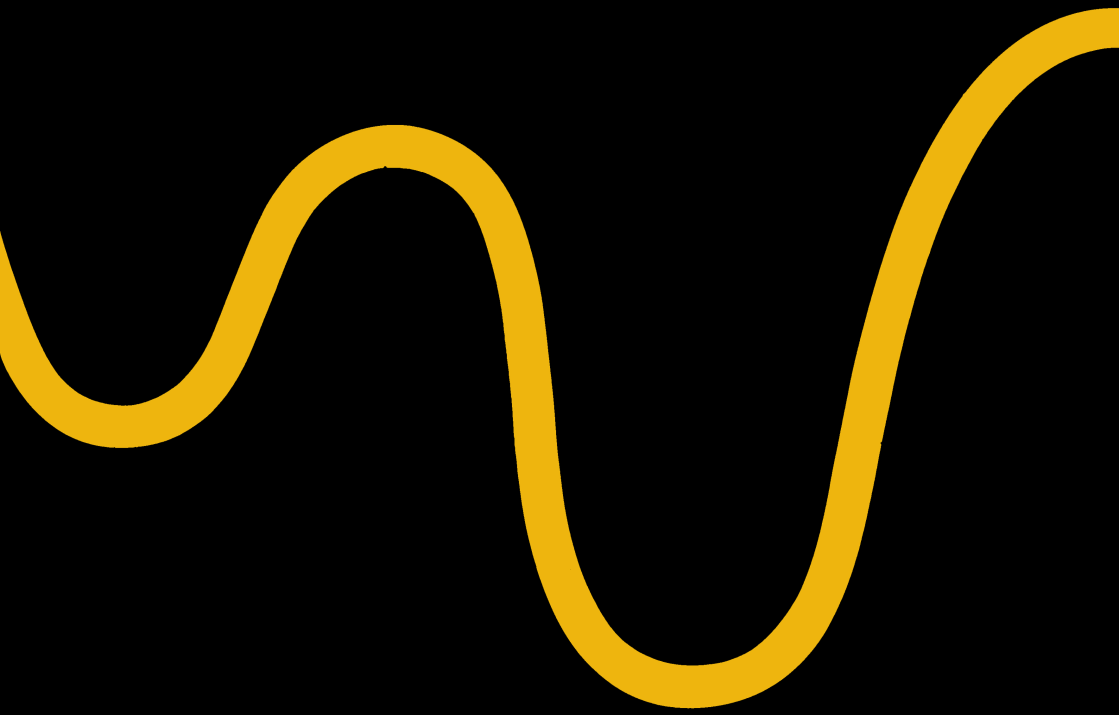
PASCAL

GUIDA PER PROGRAMMATORI

OLIVIER LECARME
JEAN-LOUIS NEBUT



McGraw-Hill

PASCAL
Guida per programmatori

PASCAL

GUIDA PER PROGRAMMATORI

OLIVIER LECARME
JEAN-LOUIS NEBUT

McGRAW-HILL Book Company GmbH

Amburgo New York St. Louis San Francisco Auckland
Bogotá Città del Guatemala Città del Messico Johannesburg
Lisbona Londra Madrid Montreal Nuova Delhi Panama
Parigi San Juan San Paolo Singapore Sydney Tokyo Toronto

Titolo originale: *Pascal for Programmers*

Realizzazione editoriale: EDIGEO srl, via del Lauro 3, 20121 Milano
Traduzione: Stefano Gatti e Cristina Leporati
Grafica di copertina: Valentina Boffa
Composizione e stampa: Litovelox, Trento

ISBN 88-7700-602-1

1ª edizione settembre 1985

Indice

Prefazione

L'incentivo a scrivere questo libro ci venne da una considerazione sorprendente, fatta un po' di tempo fa: mentre cresceva il numero di testi introduttivi sulla programmazione dei calcolatori che usavano il Pascal come linguaggio principale, c'era un assoluto bisogno di un libro che introducesse e descrivesse l'intero linguaggio a coloro che avevano già familiarità con la programmazione. Alcuni dei migliori testi sul Pascal disponibili si limitavano a descrivere la programmazione in sé e per sé, dando assai poco rilievo al linguaggio di programmazione usato. Molti altri descrivevano la programmazione soltanto in termini di Pascal, usando il linguaggio come un riferimento per discutere i concetti fondamentali dell'argomento. Tutti erano stati scritti per essere usati durante un corso introduttivo di programmazione.

La nostra considerazione risale alla metà del 1979. Ora, molti anni dopo, sono stati pubblicati molti altri testi che richiamano il Pascal nei titoli, ma la situazione generale rimane per la maggior parte immutata. Riteniamo che sia ancora necessario un libro che spieghi tutto sul Pascal a coloro che già conoscono la programmazione e che non devono imparare anche che cosa è un calcolatore, un programma, un'espressione o una procedura.

Si presume che il lettore abbia precedentemente usato alcuni linguaggi di programmazione ad alto livello come il Fortran, il Cobol, il BASIC o il PL/1, o anche linguaggi assemblatori o macchina. Si suppone che il lettore abbia dimestichezza con i concetti base della programmazione e in questo modo non si perderà tempo in ulteriori spiegazioni.

Conseguentemente, questo libro non è stato scritto per essere usato in un corso introduttivo sulla programmazione, poiché a questo scopo esistono già molti testi soddisfacenti. Il nostro libro può essere usato per un corso intermedio da coloro che scrivono programmi più avanzati, ma il suo fine principale è quello di servire come libro di testo per l'autoapprendimento, poiché esso offre una trattazione esauriente del Pascal e una descrizione fedele e precisa del linguaggio standardizzato dall'ISO e dagli istituti nazionali di standardizzazione; nostra intenzione è anche che esso venga usato come testo di riferimento. A que-

sto scopo abbiamo riservato un capitolo separato a ciascuna delle principali caratteristiche del linguaggio e abbiamo fornito numerose appendici per chi necessita di dettagli più specifici.
Rispettiamo scrupolosamente lo Standard ISO, anche quando non concordiamo completamente con alcune scelte minori fatte durante la sua preparazione. Riteniamo che il linguaggio definito da questo standard sia il solo che abbia il diritto di essere chiamato *Pascal Standard*. Qualsiasi modifica, restrizione o estensione fatta nel corso delle implementazioni specifiche del linguaggio, qualsiasi siano le sue qualità intrinseche, dovrebbe essere descritta e documentata dai realizzatori stessi, in un testo di riferimento allegato.
Pensiamo che tali modifiche non trovino spazio in un libro con un intento generale, per quanto comunemente adottate. Inoltre, crediamo che descrivere e spiegare le variazioni locali o le estensioni del Pascal significhi promuoverle e perpetuarle, ostacolando così gli scopi principali della standardizzazione internazionale, essenziale per il futuro del Pascal.
Noi, autori di questo libro, siamo stati impegnati col Pascal per parecchi anni. Siamo entrambi insegnanti universitari e abbiamo insegnato o usato il Pascal per molto tempo, come strumento per insegnare la programmazione ad ascoltatori differenti, con differenti background. Entrambi facciamo parte di gruppi di lavoro interessati al Pascal e alla metodologia di programmazione. Jean-Louis Nebut ha completato recentemente un libro sul Pascal che fa parte della categoria già ben rappresentata, ma in francese. Questo testo si è dimostrato molto utile, poiché esisteva assai poco sull'argomento in questa lingua. Olivier Lecarme era un sostenitore dell'uso del Pascal già nel 1972 (durante una conferenza di lavoro dell'IFIP). Ha contribuito a parecchie implementazioni del Pascal ed è un membro attivo del comitato di standardizzazione internazionale e francese sul Pascal.
La nostra assai varia esperienza ci ha fatto esaminare questo linguaggio molto attentamente. Noi lo consideriamo ancora oggi adattò ai suoi scopi originali, così come lo era quando fu progettato per la prima volta.
Abbiamo diviso il compito di scrivere questo libro nel seguente modo: Olivier Lecarme ha scritto tutti i capitoli e ha fornito la maggior parte degli esempi; Jean-Louis Nebut ha impostato tre capitoli e ha fornito i rimanenti esempi, tutti gli esercizi (cón la soluzione) e numerose appendici. Tutti gli esempi di programmazione e tutte le soluzioni agli esercizi di programmazione sono stati ampiamente verificati da Jean-Louis Nebut al centro di calcolo universitario di Rennes, con l'aiuto dell'implementazione del Pascal per Multics, fatta a Grenoble dal CRISS-IREP con la sovvenzione finanziaria della INRIA. Inoltre, i testi fatti da Nebut sono stati molto utili per il gruppo di Grenoble, specialmente come incentivo a mantenere le loro implementazioni il più possibile in armonia con lo Standard ISO.
Naturalmente questo libro risente dell'influenza dei libri precedenti. Riconosciamo con piacere un debito particolare allo Standard ISO (1983) e al Jensen e Wirth (1974) come libri di consultazione; a Wirth (1973, 1976) e Alagić e Arbib (1978) come fonte di esempi; al libro di Nebut sopra menzionato e al corso te-

nuto da Jean-Claude Boussard e Olivier Lecarme nella sessione estiva del 1977 a Montreal come modelli per l'organizzazione del libro e l'ordine di presentazione dei vari concetti e argomenti.

Desideriamo ringraziare anche le nostre famiglie e i colleghi per la loro tolleranza e partecipazione durante la realizzazione di questo progetto. Senza il loro costante aiuto e incoraggiamento, la realizzazione di questo libro non sarebbe stata possibile.

Olivier Lecarme
Jean-Louis Nebut

Introduzione

Se paragonato con il Fortran, il BASIC, il Cobol, o il PL/1, il Pascal non è semplicemente un altro linguaggio che permette la scrittura di programmi, bensì uno strumento che incoraggia la scrittura di programmi *migliori*; non sarebbe utile iniziare con un esempio di programma scritto in Pascal. Le qualità del linguaggio potrebbero non essere abbastanza chiare in un programma dimostrativo lungo una pagina. Perciò in questa introduzione si descriveranno soltanto i principi essenziali che governano il progetto del Pascal e le caratteristiche principali che ne derivano. Come in un processo di apprendimento per passi successivi, i lettori troveranno che la loro conoscenza del Pascal crescerà ad ogni capitolo e quindi non si dovranno sorprendere se occorrerà leggere l'intero libro per averne un'idea completa.

NOZIONI GENERALI SUL PASCAL

Il linguaggio di programmazione Pascal fu progettato da Niklaus Wirth, professore alla Scuola Politecnica Federale di Zurigo, Svizzera, nel 1969. La principale motivazione del suo autore fu di definire uno strumento per insegnare la programmazione in modo sistematico. I linguaggi allora usati per questo scopo erano, a suo avviso, o troppo primitivi, o troppo complicati, o basati su principi poco chiari, o tutte queste cose insieme. La necessità di un linguaggio basato su principi semplici, logici e naturali era particolarmente evidente, come testimoniava la crescente popolarità della nuova filosofia di programmazione, chiamata *programmazione strutturata*, iniziata da Edsger Dijkstra, C.A.R. Hoare e altri.

Naturalmente è possibile in qualsiasi linguaggio, perfino in linguaggio macchina, scrivere dei programmi ben strutturati che siano affidabili, manutenibili, portabili ed efficienti. Tuttavia, specialmente insegnando le prime nozioni sulla programmazione, sembrava veramente assurdo dover usare gli strumenti esistenti con così numerose precauzioni che perfino la scrittura di un programma semplice diventava un compito noioso. Conseguentemente, l'idea principale nel

progetto del Pascal era che un buon linguaggio di programmazione dovesse incoraggiare la costruzione di buoni programmi.
La seconda idea era che tale linguaggio dovesse essere implementato facilmente ed efficientemente sui calcolatori moderni, invece di aver bisogno di compilatori mastodontici con delle fasi di ottimizzazione poco affidabili, o di nuove architetture di macchine utopistiche, o ancora di drastiche definizioni di sottoinsiemi del linguaggio per essere ragionevolmente efficiente. Come conseguenza, quando fu pubblicata la definizione del linguaggio, si poteva già usare un compilatore completo, pratica assai poco frequente fra i progettisti di linguaggi.
Si sono usati numerosi principi fondamentali per raggiungere questi due scopi principali. Per esempio si è deciso che:

a) Un concetto deve essere introdotto nel linguaggio soltanto se è necessario e ben capito da tutti. Le nuove caratteristiche non verificate devono essere introdotte soltanto nei linguaggi sperimentali.
b) Le ridondanze utili vengono usate il più possibile, cioè quelle ridondanze che possono essere controllate dal compilatore o dal sistema di implementazione. Le dichiarazioni obbligatorie sono il miglior esempio di tali ridondanze.
c) Poiché sarebbe controproducente dare a chi usa il linguaggio differenti mezzi per fare la stessa cosa, il linguaggio deve usare soltanto i migliori mezzi disponibili ed eliminare gli altri.
d) Poiché la prevenzione degli errori implica sempre un certo costo, il controllo necessario deve essere fatto a tempo di compilazione, cosicché il controllo viene fatto una volta soltanto, invece che durante l'esecuzione, caso che richiederebbe la ripetizione del controllo ad ogni singola esecuzione del programma. Il progetto del linguaggio deve quindi facilitare questo tipo di controllo il più possibile.
e) Sebbene possano esistere nel linguaggio due concetti differenti che, qualche volta, si equivalgono, non sempre questa equivalenza è generalizzabile. Per esempio, se una funzione può essere definita usando un'espressione, ciò non significa che la sostituzione dell'espressione con il nome della funzione debba essere permessa ovunque: sarebbe meglio proibire una generalizzazione inutile, piuttosto che consentirla, aumentando il costo dei programmi che ne fanno uso.
f) I linguaggi universali tendono ad essere enormi e barocchi e nessuna persona comune che li utilizzi può capirli interamente. Un linguaggio che è progettato per essere semplice ed efficiente deve necessariamente avere delle limitazioni al suo stesso uso.

Come conseguenza di questi principi fondamentali di progettazione, il Pascal è veramente un linguaggio semplice e sistematico. È molto più semplice del PL/1 o del Cobol, e perfino più semplice del Fortran. Tuttavia, generalmente parlando, possiede una maggiore potenza espressiva del BASIC o del Fortran e, in molte situazioni, anche del Cobol. Non essendo universale, il Pascal non ha lo stesso campo di applicazione del PL/1 e non può essere usato, per esempio,

nella manipolazione dei file ad accesso diretto o nella gestione di eventi asincroni.
Il campo dell'elaborazione dati per uso gestionale è stato così a lungo identificato con il Cobol che può risultare difficile trasporre una particolare applicazione dal Cobol al Pascal senza riprogettarla interamente, ma può anche valere la pena di fare questa fatica. I lettori abituati al Cobol dovranno essere più pazienti degli altri, e dovranno aspettare fino al Capitolo 11 per gli esempi del tipo al quale sono abituati.
I lettori che hanno una certa familiarità con qualsiasi linguaggio di programmazione si sentiranno probabilmente frustrati, quando non riusciranno a trovare alcune delle loro caratteristiche favorite nel Pascal. Ogni volta che una mancanza è particolarmente evidente, si proverà a spiegare perché tale caratteristica è assente nel Pascal. La risposta, generalmente, sarà che questa caratteristica non è ben compresa da coloro che usano il linguaggio, non è molto utile, è facilmente sostituita da altre parti del linguaggio, o ha costi di implementazione troppo alti.
Un'altra differenza che dovrebbe colpire il lettore è la differenza di stile. La programmazione «difensiva», l'evitare i GOTO, la scelta di strutture di dati ben definite, una struttura di programma gerarchica, l'evitare trucchi di programmazione, tutte queste regole di buon stile sono sempre più accentuate nei libri di programmazione. Tali regole possono essere applicate ai programmi scritti in qualsiasi linguaggio, ma tendono ad essere completamente naturali nel Pascal. Per esempio, l'evitare i GOTO in BASIC o Fortran è semplicemente una diabolica fonte di frustrazione. Il giusto uso di quello strumento inestimabile che è il concetto di procedura, in Cobol e in PL/1, produce generalmente programmi pesanti, facilmente sbagliati e inefficienti. In Pascal, al contrario, dopo che si è acquisita una certa esperienza, l'uso di tali stili di programmazione sembrerà perfettamente naturale, come sarà dimostrato, ci auguriamo, dagli esempi e dagli esercizi in questo libro.

BREVE STORIA DEL PASCAL

Il Pascal fu progettato nel 1969 e chiamato in onore di Blaise Pascal, scienziato, filosofo e scrittore francese del XVII secolo («Pascal» non è un acronimo: quindi non deve essere scritto in lettere maiuscole). Il linguaggio stesso ha le sue radici nell'Algol 60, nell'Algol W (un derivato dell'Algol 60, progettato da Niklaus Wirth e da C.A.R. Hoare nel 1965) e nelle idee esposte nell'articolo *Notes on Data Structuring* di Hoare (Dahl, Dijkstra e Hoare, 1972), pubblicato molto tempo dopo che era stato scritto e presentato in seminari.
Il primo compilatore Pascal — scritto in Pascal stesso e caricato usandone una copia tradotta a mano in un linguaggio a basso livello — venne scritto da Wirth e dai suoi assistenti, specialmente da Urs Ammann, per un calcolatore CDC 6000. La prima definizione ufficiale del Pascal venne pubblicata nel gennaio del 1971 (Wirth, 1971) in un rapporto di 28 pagine estremamente conciso e

qualche volta difficile da leggere, in cui ne veniva formalizzata la sintassi, ma non la semantica, che era descritta in inglese in modo informale.
Un tentativo di formalizzare la definizione della semantica veniva fatto in un rapporto di Hoare e Wirth, pubblicato nel 1973, che descriveva quasi tutta la semantica in sole 13 pagine. Il rapporto suggeriva anche alcune leggere revisioni al linguaggio, che furono inserite in una relazione revisionata scritta nello stesso anno, ma mai pubblicata.
Il nuovo linguaggio aveva suscitato molto interesse e alcune critiche e controversie, derivanti dalla concisione e brevità del rapporto descrittivo, dall'apparente debolezza del linguaggio, o dalle ambiguità nella sua descrizione. Nel frattempo, l'implementazione sul CDC venne rifatta interamente da Urs Ammann ed ampiamente distribuita, specialmente nelle università di tutto il mondo. Questa implementazione servì come base e strumento per le implementazioni su altri calcolatori, specialmente per un'implementazione portabile costruita a Zurigo, che usava un interprete ed era conosciuta col nome di Pascal-P. Quest'ultima fu presto usata per implementare il Pascal su quasi tutti i calcolatori esistenti, dal più piccolo micro al più grande super-calcolatore.
Con Kathleen Jansen, Niklaus Wirth scrisse un manuale per l'utente, che fu pubblicato con la relazione revisionata (Jensen e Wirth, 1974) e divenne rapidamente un best-seller. Tuttavia, la definizione ufficiale del linguaggio rimase ambigua e incompleta, e c'era una fortissima pressione da varie fonti per estendere il linguaggio in parecchie direzioni incompatibili. Inoltre, la facilità di implementazione del linguaggio e il fatto che quasi tutti i compilatori fossero scritti nel linguaggio stesso, rese fin troppo facile l'implementazione di deviazioni mal progettate e di estensioni scarsamente motivate, specialmente da quelle persone che non avevano capito pienamente il progetto generale del linguaggio o alcune delle sue caratteristiche più originali. E il Pascal veniva sempre più largamente usato nella scrittura di software commerciale, perfino più di quanto non fosse usato come strumento didattico.
I tempi erano chiaramente maturi per una standardizzazione del linguaggio, se la comunità dei suoi utenti voleva assumerlo come linguaggio comune e non degradarlo ad un arcipelago di dialetti sempre meno omogenei, come era già accaduto al BASIC. L'impulso alla standardizzazione fu dato da molte persone, ma specialmente dal *Pascal Users Group*, un gruppo internazionale informale di utenti del Pascal, creato con lo scopo di scambiare idee e informazioni sul linguaggio per mezzo di un bollettino trimestrale. L'iniziativa ufficiale fu presa dal *British Standards Institute*, membro dell'Organizzazione Internazionale per la Standardizzazione (ISO).
Sotto la direzione di Tony Addyman, furono scritte successivamente numerose proposte preliminari di rapporti sul Pascal che furono sottoposte alla comunità internazionale degli utenti, quindi ai comitati ufficiali dell'ISO e dei suoi membri nazionali. L'ANSI e l'IEEE per gli Stati Uniti, il DIN per la Germania, l'AFNOR per la Francia e l'associazione olandese furono i maggiori contribuenti. Il difficile e pesante processo di standardizzazione internazionale è ora completo e lo standard internazionale può essere considerato congelato.

Il *Pascal Standard* (ISO, 1983) è principalmente una chiarificazione del rapporto revisionato scritto da Niklaus Wirth. Ne elimina le ambiguità, risolve le contraddizioni, riempie gli spazi vuoti non lasciando più campo a diverse interpretazioni, fatte da persone diverse. Inoltre, attua una leggera modifica e un'importante estensione al linguaggio, entrambe giudicate assolutamente necessarie dalla comunità internazionale e approvate dall'autore originale del linguaggio. Una convalida ulteriore fu sviluppata durante il processo di standardizzazione e costituisce uno strumento inestimabile per determinare se una data implementazione è conforme allo Standard e, se ciò non avviene, per stabilire dove se ne distacca.

Il presente libro si basa sullo Standard ISO del Pascal e non su una particolare implementazione; usa, per quanto è possibile, la terminologia dello Standard, ma non usa né la sua metodologia descrittiva, né la sua sequenza di presentazione, che sono sistematiche ma non pedagogiche. Infatti, come tutti gli altri Standard, sebbene sia più breve e leggibile della maggior parte di questi, lo Standard del Pascal è difficile e noioso sia da leggere che da capire e non può servire come documento descrittivo. È soltanto il riferimento finale, inteso a dare la risposta ufficiale a tutte le domande possibili (una possibile risposta tutt'altro che rara è che non c'è risposta alla domanda).

Ad una visione retrospettiva, ci si può chiedere che cosa ha causato il successo del Pascal, un successo che non era stato affatto prevedibile quando fu pubblicato il suo primo rapporto descrittivo. Le ragioni per delle aspettative così limitate erano evidenti: il rapporto era stato pubblicato nel primo numero di un giornale poco noto; nessuna organizzazione di sostegno, o agenzia, o venditore aveva svolto alcuna campagna promozionale; la prima implementazione era stata fatta su una macchina poco usata ed era deliberatamente un linguaggio privo di ambizione.

A nostro avviso, il successo del linguaggio risulta dalla congiunzione di parecchi fattori differenti:

a) L'implementazione iniziale fu sorprendentemente buona ed efficiente, scritta nel linguaggio stesso, leggibile, comprensibile e disponibile senza spesa.
b) L'implementazione portabile Pascal-P rese possibile usare il linguaggio sui microcalcolatori, che iniziavano proprio allora a comparire sul mercato.
c) Il *Pascal Users Group* ha dato un sostegno inestimabile, economico e amichevole, alle numerose persone interessate al linguaggio.
d) Non vi furono conflitti politici o economici, proprio a causa dell'assenza di qualsiasi sostegno ufficiale.

Inoltre il linguaggio appariva proprio nel momento in cui era necessario, in mezzo al disordine della crisi del software, come lo strumento naturale per implementare i principi della programmazione strutturata, che erano così dibattuti quando venivano applicati a linguaggi come il Fortran o il Cobol. Sebbene il Pascal fosse ancora completamente sconosciuto al di fuori dei circoli accademici cinque anni dopo la sua definizione iniziale, altri cinque anni lo resero abba-

stanza noto da essere considerato un temuto sfidante, sia per il Fortran che per il BASIC, per il titolo di linguaggio di programmazione più popolare. Dopo tutto, può essere che il successo del linguaggio sia semplicemente il risultato delle sue qualità intrinseche.

ORGANIZZAZIONE DEL TESTO

Questo libro è destinato a coloro che già possiedono delle nozioni generali sulla programmazione, perché hanno frequentato un corso introduttivo, o perché hanno affrontato la materia come autodidatti. Questo è un libro sul Pascal e non sulla programmazione; tuttavia, si è provato nel testo a incoraggiare la progettazione e la costruzione di buoni programmi. Abbiamo scelto di presentare queste idee con degli esempi e commenti pragmatici, non con l'esposizione e lo sviluppo di grandi principi generali.
Conseguentemente, una persona senza alcuna esperienza precedente di programmazione non ricaverà alcun profitto dalla lettura di questo libro e sarebbe consigliabile che si accostasse prima a uno dei numerosi libri di introduzione alla programmazione presenti sul mercato. Coloro che possiedono già alcune nozioni di programmazione e hanno anche una conoscenza superficiale del Pascal possono leggere i primi capitoli di questo libro assai rapidamente, controllando ciò che già conoscono con l'aiuto degli esempi e studiando il testo più attentamente soltanto quando inizieranno gli esempi al Capitolo 5. Pensiamo che coloro che già conoscono bene il Pascal possano trovare questo libro degno di nota e che esso possa approfondire la loro conoscenza del Pascal e perfino presentare alcuni aspetti del linguaggio che sono sconosciuti e nuovi modi di usarlo.
La sequenza di presentazione usata è poco comune a causa dello scopo insolito che questo libro si propone. Una descrizione formale del linguaggio finirebbe per presentare tutte le strutture dei dati in successione, quindi tutte le strutture di controllo, con lo sfortunato inconveniente di imporre una doppia lettura. Un libro introduttivo sulla programmazione cercherebbe di raggruppare il maggior numero possibile di concetti semplici all'inizio per consentire la presentazione immediata di esempi semplici; quindi presenterebbe i concetti rimasti in progressione dal più semplice al più avanzato (cioè complicato), mischiando così le cose e nascondendo la struttura logica del linguaggio. In questo libro, invece, si è cercato di usare una successione logica e sistematica per descrivere un linguaggio logico e sistematico.
Nei primi quattro capitoli vengono presentati i mattoni fondamentali che costituiscono i programmi. I programmi sono fatti di *azioni* che elaborano *oggetti*, conseguentemente i blocchi costituenti sono *istruzioni* per azioni elementari e *tipi* per oggetti elementari. I rimanenti capitoli presentano, una dopo l'altra, le strutture differenti che rendono possibile costruire i programmi a partire da questi blocchi costituenti. Ogni capitolo tratta una particolare struttura, sia per le azioni che per gli oggetti. La successione di questi capitoli è progressiva, cioè i concetti più difficili sono collocati alla fine: a parte ciò, i capitoli sono indi-

pendenti tra loro; tuttavia essi dovrebbero essere letti in ordine, perché gli esempi e gli esercizi usano generalmente tutti i concetti che sono stati presentati precedentemente.

Gli esempi sono parte integrante del testo e non dovrebbero mai essere tralasciati: essi sono l'unico mezzo per capire pienamente l'intento e la funzione dei concetti. Nella maggior parte dei casi essi sono seguiti da commenti che aggiungono informazioni sull'argomento di natura pratica e pragmatica, che non possono essere spiegate *in abstracto*. Andando avanti, gli esempi e i commenti diventano progressivamente più lunghi, poiché usano sempre di più la potenza espressiva del linguaggio.

Gli esercizi presentati sono reali: non esercitazioni, ma veri problemi di programmazione. Le soluzioni degli esercizi scelti (uno per capitolo, identificato da un asterisco) appaiono nell'Appendice A. Queste soluzioni seguono la logica degli esempi. Naturalmente, i lettori seri sono fortemente incoraggiati a trovare loro soluzioni prima di ricorrere a quelle suggerite dal testo — che sono anzi incoraggiati ad analizzare e criticare — che non vengono assolutamente intese come le uniche «giuste» possibili.

Come già detto, il linguaggio Pascal descritto nel testo è esattamente quello descritto dallo Standard dell'ISO; come tale differisce dai linguaggi accettati da implementazioni non standardizzate. Poiché lo Standard è nuovo, le implementazioni non standardizzate costituiscono attualmente la regola, ma si auspica che tutti gli implementatori seri adegueranno i loro prodotti allo Standard il più presto possibile. Come modesto incentivo, si cercherà di non descrivere le deviazioni dallo Standard di uso corrente, qualunque possa essere la loro popolarità.

Per quanto concerne l'adeguamento allo Standard di una data implementazione, si devono precisare alcune nozioni. Nella maggior parte dei casi, quando lo Standard stabilisce qualcosa, un programma che se ne distacca è semplicemente illegale e deve essere rifiutato da un'implementazione conforme allo Standard. Per esempio, la dichiarazione di tutte le variabili è obbligatoria nel Pascal, e un programma che dimentichi di dichiarare alcune variabili è semplicemente errato: nessuna implementazione seria del Pascal potrà accettarlo.

In altre situazioni, il problema può essere più complesso e non verrà individuato senza mandare in esecuzione il programma. Questo avviene, per esempio, se si ha un divisore uguale a zero, o se il valore di un indice di array cade al di fuori dei limiti definiti per quell'array. Tale situazione è chiamata *errore* e, sebbene gli implementatori debbano fare del loro meglio per mantenere un buon controllo in questi casi, ve ne sono alcuni particolarmente difficili oppure c'è qualche macchina particolarmente ostile che può rendere questo controllo impossibile o molto costoso. Il linguaggio Pascal è stato progettato per minimizzare il numero di tali situazioni e le rimanenti difficoltà sono messe in evidenza in questo libro.

In certe situazioni lo Standard non è in grado di specificare che cosa si deve fare e lascia alcune decisioni all'implementatore. Questo porta alle due nozioni di concetto *definito nell'implementazione* e *dipendente dall'implementazione*. Un

concetto definito nell'implementazione deve essere definito, ma in modo che possa differire da un'implementazione all'altra. Esempi semplici sono la precisione dell'aritmetica dei reali, l'intervallo dei numeri interi e l'insieme dei caratteri. Un programma portabile non deve contare su di una particolare specifica di un concetto definito nell'implementazione.

Un concetto è dipendente dall'implementazione se non è necessariamente definito in una data implementazione, il che implica che la sua definizione, se ne esiste una, può differire da un'implementazione all'altra. Un esempio semplice è l'ordine di valutazione dei termini di un'espressione. Un programma portabile non deve fare affidamento su di un concetto dipendente dall'implementazione, qualunque possa essere la sua definizione; per esempio, esso non deve assumere alcun particolare ordine di valutazione delle espressioni, e non dovrebbe neppure assumere l'esistenza di qualsiasi ordine. Una lista di tutti i concetti definiti nelle implementazioni e dipendenti dalle implementazioni si può trovare nell'Appendice E.

Infine, lo Standard dell'ISO distingue due classi di implementazioni conformi allo Standard, a seconda che accettino o meno una particolare estensione fatta al linguaggio originale dal processo di standardizzazione. Le implementazioni conformi al *livello 1* implementano il concetto di conformant-array corrispondente, come descritto nel paragrafo 9.7, mentre quelle conformi al *livello 0* non lo implementano. (L'ANSI Standard definisce solo il livello 0).

1

Elementi di base

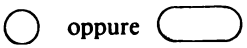
Un programma è una sequenza di istruzioni eseguibili che elaborano e producono informazioni. La sequenza di istruzioni costituisce la struttura di controllo dell'algoritmo descritto dal programma; le informazioni, numeriche e non, sono organizzate in strutture dati adatte ad affrontare il problema che si vuole risolvere. Ogni linguaggio di programmazione ha delle regole che stabiliscono come scrivere le istruzioni in modo corretto e come costruire le strutture dati. Il modo in cui viene presentata la grammatica differisce da linguaggio a linguaggio ed il metodo scelto per questo libro si scosta abbastanza dai formalismi più tradizionali che derivano dalla Forma Normale di Backus (o dalla Backus-Naur Form). Speriamo che la nostra presentazione possa essere compresa più facilmente da chi si accosta al Pascal per la prima volta.

Questo capitolo illustra sia il modo in cui verranno descritti i vari costrutti linguistici nel resto del libro, sia le regole per la costruzione dei nomi. Sono quindi presentate le nozioni elementari che servono per la comprensione di un programma Pascal, la definizione sintattica di costanti e variabili, e vengono introdotte le istruzioni semplici del linguaggio, di quegli elementi cioè che consentono di scrivere istruzioni composte, procedure e, in ultima analisi, l'intero programma.

Dal momento che questo libro vuole anche essere un manuale di riferimento del linguaggio e non solo un'introduzione al Pascal, sarà necessario anticipare sia in questo, che nei tre successivi capitoli, alcuni punti che devono essere menzionati subito, ma che non potranno essere spiegati fino a che non saranno stati introdotti i concetti necessari. Questo inconveniente sarà sempre meno avvertito nei capitoli successivi al secondo ed i riferimenti in avanti potranno essere saltati in una prima lettura.

1.1 METODO DI PRESENTAZIONE

Nei capitoli successivi, la presentazione di ciascuna istruzione, semplice o composta, sarà preceduta da una discussione sulle sue funzioni e da un breve confronto con istruzioni simili, nel caso ve ne siano, in quattro linguaggi di programmazione ampiamente usati: Fortran, Cobol, BASIC e PL/1. La sintassi è descritta formalmente mediante un metalinguaggio grafico, seguita da opportuni commenti in italiano ed illustrata con esempi. La semantica è descritta formalmente con regole di verifica e quindi riformulata in italiano. Ambedue i modi di presentare la grammatica hanno lo stesso obiettivo: fornire una descrizione chiara, precisa e non ambigua. Probabilmente le spiegazioni in italiano saranno più utili durante l'uso del libro come testo introduttivo al Pascal, mentre si ricorrerà alle descrizioni formali, sintattiche e semantiche, quando lo si consulterà come manuale di riferimento.
Per ciascun modo possibile di strutturare i dati viene fornita dapprima una descrizione formale dei concetti sottostanti, quindi viene data una definizione delle corrispondenti strutture dati in Pascal. E così possibile mantenere nettamente separati i concetti astratti dalle corrispondenti implementazioni e confrontare più facilmente il Pascal con gli altri linguaggi. Un altro vantaggio che si ottiene operando in questa maniera è di poter definire in modo preciso gli operatori applicabili alle varie strutture dati.
Dopo che ne è stata definita la sintassi e la semantica, ciascun costrutto linguistico — sia che si tratti di istruzioni, che di strutture dati — viene riproposto in un piccolo esempio illustrativo, seguito da chiarimenti e osservazioni pragmatiche, su possibili varianti e miglioramenti in vista di una maggior tolleranza ai guasti, portabilità, ecc... Non appena sarà possibile (dal Capitolo 7 in avanti), queste spiegazioni saranno seguite da esempi più generali e di maggior utilità, che fanno uso delle nozioni appena introdotte. Per quanto possibile, sono assenti affermazioni dogmatiche, evitando, ad esempio, di presentare descrizioni formali tutte le volte che queste non hanno un risvolto pratico, come nel caso in cui la descrizione formale sia più complicata della nozione stessa da descrivere, come avverrebbe per l'istruzione **goto** nel paragrafo 1.8.
Il metalinguaggio usato per la descrizione della sintassi impiega tre simboli grafici. Un cerchio (o una figura costituita da due semicerchi collegati da due tratti orizzontali), come



racchiude un simbolo del linguaggio: un carattere speciale, composto da segni di interpunzione, oppure una parola chiave o simbolo base, costituito da lettere. Un rettangolo racchiude il nome di un'altra regola sintattica, che è descritta altrove. Cerchi e rettangoli sono congiunti da frecce, che definiscono l'ordine nel quale si susseguono.
Una regola sintattica è definita in un diagramma che contiene almeno due frec-

ce, una che ne indica il punto di ingresso e un'altra che ne indica l'uscita. Il titolo del diagramma è il nome della regola sintattica ivi definita. Una «frase», costruita concatenando fra loro i simboli che si incontrano percorrendo il diagramma in uno dei modi possibili (seguendo le frecce e scegliendo una delle direzioni possibili quando si incontra un bivio) è una frase conforme alla regola. Il diagramma che definisce un elemento del linguaggio può sostituire tale elemento quando compaia in un rettangolo. Una frase che non contenga più nomi di regole (nella quale cioè siano stati tutti sostituiti da particolari frasi conformi alla regola) è un frammento di un programma Pascal o un programma sintatticamente corretto.
Tutti i nomi di regole in questo libro sono la traduzione più corretta ed usata dei corrispondenti nomi dello Standard ISO, ma il metalinguaggio usato riduce enormemente il numero dei nomi necessari.
Lo scopo dei diagrammi sintattici che compaiono nel testo è di descrivere localmente i concetti presentati. La raccolta di diagrammi sintattici che compare al termine del volume (Appendice B) dà una descrizione sintattica, concisa ma esaustiva, dell'intero linguaggio. A causa di questi scopi contrastanti, i diagrammi non sono identici nelle due situazioni: parecchi diagrammi, sparsi su più capitoli, possono essere ricomposti in un solo diagramma nell'Appendice B.

ESEMPIO 1.1

Le seguenti frasi sono conformi alle regole sintattiche illustrate nella Figura 1.1:

proc *ident*
proc *ident* (*nome*, *nome*, *nome*)
proc type (*nome*)

ident e *nome* sono nomi di regole che, dal momento che sono racchiusi in rettangoli, saranno definite altrove.

Le regole assiomatiche che specificano la semantica sono scritte sotto forma di asserzioni o regole di verifica. L'effetto di un'istruzione, o lo stato di una strut-

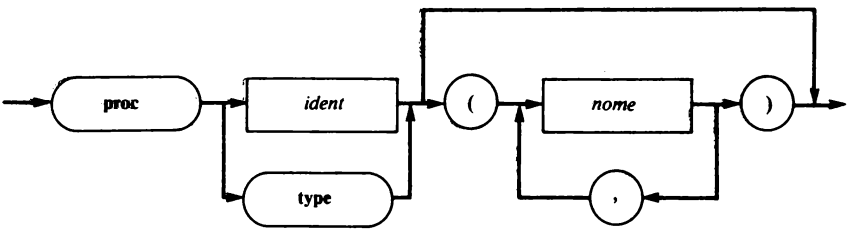


Figura 1.1 Esempio di diagramma sintattico

tura dati, è descritta da condizioni iniziali (l'*antecedente*) che precedono l'istruzione o l'operatore definito su una struttura dati, e condizioni finali (il *conseguente*), che devono essere verificate dopo avere eseguito l'istruzione o dopo avere applicato l'operatore alla struttura dati.
Un'asserzione della forma $\{P\}\, S\, \{Q\}$, dove P e Q sono formule logiche e S è un'istruzione Pascal, significa che se P è vera prima dell'esecuzione di S, allora Q sarà vera dopo la sua esecuzione. In altre parole, l'effetto di S, quando P è vera, è di rendere vera Q: S trasforma P in Q.
La notazione speciale $\{P_y^x\}$ rappresenta la formula che si ottiene sostituendo sistematicamente ogni libera occorrenza di x (non legata a P dal quantificatore universale) in y.

ESEMPIO 1.2

La semantica dell'istruzione di assegnamento, in Fortran o in PL/1, è definita dalla seguente asserzione:

$$\{P_Y^X\}\ X = Y\ \{P\}$$

Questo significa che, se conosciamo un particolare conseguente di un'istruzione di assegnamento, possiamo dedurne l'antecedente usando questa regola. Se, dopo l'istruzione di assegnamento

```
A(I) = B(J) * C(K)
```

sappiamo che A(I) = 1, questo significa che, prima dell'istruzione,

```
B(J) * C(K) = 1
```

Le regole di verifica hanno la seguente forma:

se <una o più asserzioni collegano i componenti di una struttura dati o di una struttura di controllo ad alcuni predicati>
allora <un'asserzione collega una struttura dati o una struttura di controllo agli stessi predicati>

Questo significa che, se sono verificate alcune condizioni su una struttura, allora si può definire un'asserzione sulla struttura stessa.

ESEMPIO 1.3

La semantica dell'istruzione logica IF del Fortran è definita dalla regola di verifica:

se $\{P \wedge B\}\ S\ \{Q\}$ e $\{P \wedge \neg B \supset Q\}$
allora $\{P\}$ IF (B) S $\{Q\}$

Si supponga, per esempio, che l'asserzione P sia X = COS(Y) e che si voglia rendere vera l'asserzione Q, cioè X = ABS(COS(Y)). Si può scrivere la seguente istruzione

```
IF (X.LT.0.0) X = -X
```

La condizione B è che X sia minore di zero.

1.2 ELEMENTI DI UN PROGRAMMA

Un programma Pascal è costituito da oggetti alfanumerici che possono essere sia simboli del linguaggio che entità elementari definite dal programmatore: identificatori, numeri, stringhe, label, direttive e commenti. Simboli ed entità elementari sono composti da caratteri: lettere, cifre e segni di interpunzione.
Le numerose differenze esistenti fra gli insiemi di caratteri disponibili sui vari calcolatori, o addirittura sui diversi dispositivi di input/output di uno stesso calcolatore, rappresentano certamente un problema irritante. L'insieme di caratteri che ci si può ragionevolmente aspettare essere comune a tutti i dispositivi è estremamente ristretto ed è quasi impossibile da usare come unico mezzo espressivo in un moderno linguaggio di programmazione. Di conseguenza, per la scrittura di programmi in Pascal si presuppone che esista un insieme adeguato di caratteri; esistono poi delle regole per l'uso di caratteri supplementari, disponibili su una certa classe di macchine, e per la sostituzione di eventuali caratteri mancanti con rappresentazioni alternative.
Lo Standard ISO descrive una rappresentazione preferenziale, che diventa obbligatoria quando si pensa di scrivere programmi che dovranno girare su calcolatori diversi. La potenza espressiva dell'insieme di caratteri disponibili può invece essere usata in pieno quando si scrivono programmi per altri scopi, ad esempio solo per essere letti. La descrizione che segue fa una distinzione fra la rappresentazione standard, usata per lo scambio di programmi ed accettata da qualsiasi compilatore conforme allo Standard, e la forma, più libera, usata in questo volume.
La rappresentazione particolare di un dato carattere (maiuscolo, minuscolo, corsivo, grassetto, ecc...) non ha alcun significato al di fuori della stringa di caratteri cui appartiene. Questa prima regola afferma quindi che una X maiuscola ed una x minuscola sono lo stesso carattere e Begin, begin, BEGIN e bEgIn rappresentano tutti la stessa parola. Così le lettere differenti sono solo:

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

Le cifre sono:

0 1 2 3 4 5 6 7 8 9

Nei diagrammi sintattici che definiscono i simboli lessicali, lettere e cifre sono considerate parole del vocabolario del linguaggio, sono racchiuse in cerchi e non sono ulteriormente definite in altri diagrammi.
I simboli usati nel linguaggio si dividono in caratteri speciali e parole chiave. Nel presente volume sono utilizzati i seguenti caratteri speciali:

+ − * / < ≤ ≠ ≥ > = . , .. : ; [ ] ( ) := ↑

(Si noti che solo i due simboli := e .. richiedono l'uso di due caratteri). Le parole chiave, che sono sempre stampate in grassetto, sono:

array	**begin**	**case**	**const**	**div**	**do**	**downto**
else	**end**	**file**	**for**	**function**	**goto**	**if**
in	**label**	**mod**	**nil**	**of**	**packed**	**procedure**
program	**record**	**repeat**	**set**	**then**	**to**	**type**
until	**var**	**while**	**with**	**not**	**and**	**or**

Nella rappresentazione standard usata nell'applicazione di programmi su macchine diverse, non si ricorre al grassetto e le parole chiave sono scritte nel solito modo.
I seguenti caratteri speciali sono normalmente rappresentati in un modo diverso che utilizza caratteri più facilmente disponibili.

Rappresentazione utilizzata	≠	**and**	**or**	**not**	≤	≥	[	]	↑
Rappresentazione Standard	< >	**and**	**or**	**not**	< =	> =	[	]	^
Alternativa Standard							(.	.)	@
Simbologia alternativa		∧	∨	¬					

Come mostrato in tabella, esistono alcune alternative standard per quelle rappresentazioni per cui non sono disponibili i caratteri suggeriti dallo Standard. Una particolare implementazione su cui, ad esempio, non sono presenti le parentesi quadre, dovrebbe definire un'alternativa non standard per tutti quegli operatori di confronto che fanno uso di questi caratteri. Esistono inoltre due

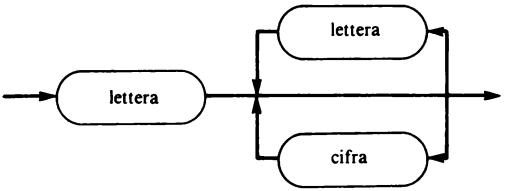


Figura 1.2 Diagramma sintattico di un identificatore

altre alternative, per delimitare i commenti, che non sono simboli del linguaggio; esse verranno descritte fra breve.
Gli identificatori sono definiti per mezzo della regola illustrata in Figura 1.2. Gli identificatori, chiamati «nomi» in molti altri linguaggi, sono una sequenza di lettere e/o di cifre che cominci con una lettera. In un identificatore non si può usare alcun altro carattere (come lo spazio) o caratteri speciali (come #, @, o $). Un identificatore può essere di lunghezza arbitraria (l'unico vincolo è il numero massimo di caratteri per riga di programma) e tutti i suoi caratteri sono significativi. Le parole chiave sono simboli riservati; un identificatore non può essere omonimo di alcuna parola chiave. Tutte queste regole e convenzioni sono sostanzialmente diverse da quelle valide per Fortran, BASIC, Cobol e PL/1 che, a loro volta, sono tutte diverse fra loro.
Gli identificatori sono scelti liberamente dal programmatore — che deve sottostare all'unica restrizione imposta dalle parole chiave — e sono usati per dare un nome ai vari oggetti definiti nel programma (costanti, tipi, variabili, nomi di campi, procedure, funzioni, parametri e limiti di array). Occorre mettere in evidenza che la scelta di identificatori significativi ha un impatto importante sulla leggibilità del programma.
Disgraziatamente molti compilatori impongono un limite, non standard, sul massimo numero di caratteri significativi di un identificatore. Il limite più comune è di otto caratteri, cosicché questi compilatori non sono in grado di distinguere *elementouno* da *elementodue*, pur essendo questi due identificatori diversi. Quel che è peggio è che la parola chiave **procedure** viene riconosciuta qualche volta all'interno di un identificatore come *procedur* o *procedurale*.

ESEMPIO 1.4

Le seguenti sequenze di caratteri rappresentano degli identificatori validi:

i indici numero1 EndOfFile OrdinamentoVeloce OUT

Le seguenti sequenze di caratteri non rappresentano degli identificatori:

Begin *3volte Uscita–Stampante* **IN**

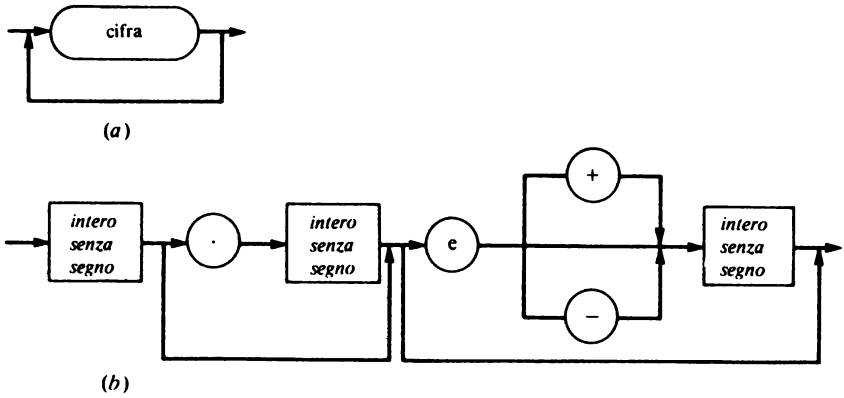


Figura 1.3 Diagramma sintattico di (*a*) un intero senza segno e (*b*) un numero senza segno

Il primo infatti è una parola chiave, il secondo comincia con una cifra, il terzo contiene un carattere diverso da una lettera o da un numero e l'ultimo è ancora una parola chiave.

Nei diagrammi sintattici e nelle rispettive descrizioni presentate in questo volume, la parola «identificatore» apparirà raramente da sola; molto spesso sarà infatti specificato a che identificatore ci si riferisce: di una costante, di una funzione, ecc... Questi diversi identificatori hanno sempre lo stesso significato; si tratta cioè di identificatori cui è stata precedentemente associata una particolare qualifica.
I numeri sono rappresentati solo in notazione decimale e possono riferirsi sia a valori interi che a valori reali (si veda il paragrafo 2.4). Un numero che contiene una parte decimale — separata da un punto — e/o la lettera «e» (che significa «dieci elevato alla») è un numero reale; in tutti gli altri casi è un numero intero. Le regole per numeri ed interi senza segno sono riportate in Figura 1.3.

ESEMPIO 1.5

Interi senza segno 0 2 83 9210
Numeri senza segno 0 0.0 3.14159 0.3e−08 3E2 1e+4

Le seguenti sequenze di caratteri non corrispondono a numeri

.0 0. 1. e+4 E−10 .e5

Si noti che questa definizione è più restrittiva di quanto si trovi in altri linguag-

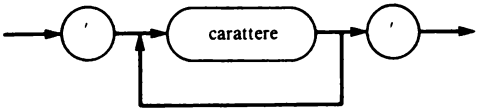


Figura 1.4 Diagramma sintattico di una stringa di caratteri

gi di programmazione, nei quali alcune, se non addirittura tutte le sequenze proposte, rappresentano numeri validi. Questa definizione più restrittiva evita molti problèmi lessicali.
Le label sono interi senza segno nell'intervallo chiuso 0-9999 e vengono usate nei casi in cui occorre identificare un'istruzione. Si riconoscono dal loro valore intero ed il loro uso, definito nel paragrafo 1.8, è molto limitato dalla grammatica del linguaggio.
Le stringhe sono costanti il cui valore è rappresentato da una sequenza di caratteri racchiusi fra apici. La regola per le stringhe di caratteri è illustrata in Figura 1.4. L'insieme di tutti i caratteri disponibili è definito dalla particolare implementazione. Una stringa che contiene un solo carattere denota una costante di tipo *char* (si veda il paragrafo 2.4). Se occorre rappresentare delle virgolette nella stringa, gli apici sono raddoppiati.

ESEMPIO 1.6

Le seguenti sequenze di caratteri rappresentano costanti di tipo *char*:

'a' 'A' ''''

La prima è la prima lettera dell'alfabeto in minuscolo, la seconda è l'equivalente maiuscolo, mentre l'ultima rappresenta le virgolette (').
Le seguenti sequenze di caratteri rappresentano delle stringhe:

'Pascal per programmatori'
'E'' sbagliato, riprova'

Le direttive hanno la stessa sintassi degli identificatori e possono apparire al posto del corpo di un sottoprogramma. Il loro uso verrà spiegato nei paragrafi 1.3 e 4.1. L'unica direttiva standard è *forward*.
I commenti sono sequenze di caratteri che compaiono al di fuori delle stringhe. Sono racchiusi fra parentesi graffe e non possono contenere, al loro interno, una parentesi graffa aperta. I simboli (* e *) sono rappresentazioni standard alternative rispettivamente della parentesi graffa aperta e chiusa. Nei programmi ben scritti si fa largo uso di commenti, per spiegare il funzionamento interno di procedure e funzioni o l'uso di variabili, parametri e costanti; per esprimere asserzioni prima o dopo istruzioni composte o sottoprogrammi; per collegare la

fine di un sottoprogramma o di una istruzione composta con il suo inizio, e così via.
Spazi, commenti ed il simbolo di fine linea sono considerati separatori. Fra due oggetti (simboli, identificatori, numeri, stringhe, label e direttive) può comparire un numero arbitrario di separatori e ve ne deve essere almeno uno fra due identificatori, numeri, label, direttive o parole chiave. Tuttavia, all'interno di un singolo oggetto non può comparire alcun separatore.
Tutte le regole precedenti sono abbastanza simili a quelle del PL/1, ma sono molto diverse da quelle del Fortran e del BASIC. Da un punto di vista generale il formato di un programma Pascal è libero. Tuttavia, una corretta rappresentazione del testo del programma è essenziale per la sua leggibilità e, a questo scopo, più di un produttore di compilatori di Pascal fornisce anche speciali programmi di paragrafatura (detti «prettyprinter») che formattano i programmi, in modo automatico, seguendo alcune regole logiche.
Tutti i programmi, o frammenti di programma, presentati in questo volume ubbidiscono implicitamente ad una serie di regole di stampa che comportano la rientranza dal margine di ciascuna linea e l'uso sistematico di commenti nelle istruzioni composte. Dato per scontato che la rappresentazione di un dato carattere sia univoca, indipendentemente dal suo modo (maiuscolo o minuscolo), dal suo corpo, stile o font (a, A, **A**, a, *a*... indicano tutte la prima lettera dell'alfabeto), si farà uso di corpi differenti nelle diverse parti del programma: le parole chiave saranno in **grassetto**, gli identificatori in *corsivo* ed i commenti avranno lo stesso carattere del testo, in un corpo leggermente più piccolo, fra parentesi graffe. In questo volume inoltre, all'interno dei commenti così come nelle regole di verifica, gli operatori logici **not**, **and** e **or** sono sostituiti, per facilità di lettura, rispettivamente dai simboli $\neg$, $\wedge$, $\vee$.
Le prime due convenzioni costituiscono la regola seguita in molti libri sull'Algol 60, Algol 68 e Pascal.

1.3 STRUTTURA DEL PROGRAMMA

In molti linguaggi di programmazione tradizionali un programma è l'unità di compilazione (rappresenta cioè quanto può essere processato da una singola attivazione del compilatore), allo stesso modo in cui lo è un sottoprogramma qualsiasi. Ciò è vero per Fortran e Cobol (laddove esistono i sottoprogrammi) ed è parzialmente vero per il PL/1. Un programma eseguibile (cioè un testo completo che può essere mandato in esecuzione) si ottiene collegando il programma principale con i sottoprogrammi di cui fa uso. Le convenzioni di collegamento sono definite, almeno parzialmente, nel linguaggio stesso; per esempio il Fortran introduce l'istruzione COMMON e ne riserva l'uso per la condivisione dei dati.
Il Pascal non definisce cosa sia l'unità di compilazione che dipende, di solito, dalla particolare implementazione del linguaggio. Lo Standard ISO lascia aperta la possibilità di chiamare e descrivere un sottoprogramma esterno ad un pro-

gramma (tramite opportune direttive non standard descritte nel Capitolo 4), ma non fornisce alcuno strumento per mettere in comune dati fra più unità di compilazione. Tutti gli oggetti usati in un programma devono essere definiti al suo interno.
Per tutti gli esempi completi presentati in questo volume, un programma Pascal è sia una unità di compilazione che una entità autosufficiente, in grado di essere mandata in esecuzione. La fase di collegamento deve solo cercare nella libreria ed incorporare i sottoprogrammi che costituiscono il supporto Pascal a tempo di esecuzione e che sono circa una trentina.
Permettere la compilazione indipendente di un sottoprogramma vorrebbe dire non potere verificare l'uso dei parametri e ciò sarebbe in contrasto con uno dei cardini della filosofia del Pascal: il controllo statico di tipo per migliorare sicurezza ed affidabilità dei programmi.
Un programma Cobol è identificato con molti dettagli nell'IDENTIFICATION DIVISION. Il Fortran non richiede un'identificazione del programma ed in PL/1 solo l'opzione MAIN specifica che una delle procedure è in realtà il programma principale. In Pascal un programma è composto da un'intestazione di programma (che ne costituisce l'identificazione) seguita da un blocco (definizione e dichiarazione di tutti gli oggetti usati nel programma e istruzioni) e termina con un punto (si veda la Figura 1.5).
L'identificatore che segue il simbolo **program** è il nome del programma, che non ha alcun significato all'interno del programma (naturalmente può averne uno all'esterno). L'identificatore opzionale che segue è una lista di parametri di programma, costituita da identificatori tutti diversi tra loro, che devono essere dichiarati nel blocco del programma (con un'unica eccezione spiegata nel paragrafo 8.2). L'uso più frequente di questi parametri è quello di identificatori dei file che saranno usati nel programma; la loro presenza nell'intestazione del programma consente di collegarli a file fisici, o ad insiemi di dati, nell'ambiente del programma. Tuttavia il metodo e il significato di questo collegamento dipendono dall'implementazione. Un solo punto è specificato nello Standard ISO: un programma che legge dati da un dispositivo standard di input, rappresentato dall'identificatore di file predefinito *input*, deve specificare questo identificatore come un parametro del programma; analogamente deve specificare l'identificatore di file predefinito *output* se effettua delle operazioni di scrittura sul dispositivo standard di output. I dettagli saranno trattati nel paragrafo 8.2.

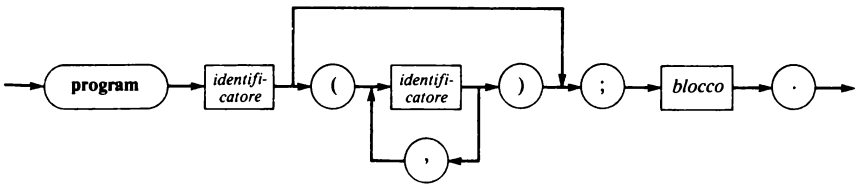


Figura 1.5 Diagramma sintattico per un programma

Altri identificatori possono essere usati come parametri di programma, ma il loro significato e la loro relazione con le corrispondenti entità esterne non sono standard; in alcune implementazioni l'uso di alcuni tipi di identificatori può essere illegale, mentre in altri è permesso. Di fatto l'intestazione di un programma ne rappresenta l'interfaccia con il mondo esterno, motivo per cui questa parte del linguaggio dipende fortemente da tale ambiente. In alcune implementazioni i parametri di programma possono essere delle costanti: in questo modo è possibile parametrizzare il programma, assegnandone dei valori solo al momento della sua esecuzione: inoltre possono essere degli identificatori di sottoprogrammi per procedure o funzioni esterne al programma (nel programma apparirà la sola intestazione seguita dalla direttiva non standard *external*). Qualche volta questo metodo è utilizzato anche per le variabili ordinarie.
A grandi linee si può affermare che un blocco equivale alla coppia DATA DIVISION – PROCEDURE DIVISION del Cobol. La corrispondenza con il corpo di una procedura in PL/1 o con una subroutine in Fortran è più tenue, dal momento che ambedue questi linguaggi consentono l'uso di molti tipi di dichiarazioni implicite e non stabiliscono una chiara linea di demarcazione fra le dichiarazioni e le istruzioni eseguibili. In Pascal tutti gli oggetti usati in un programma devono essere definiti o dichiarati prima del loro uso (ad eccezione di quegli oggetti che sono predefiniti o predichiarati nel linguaggio). Questi oggetti

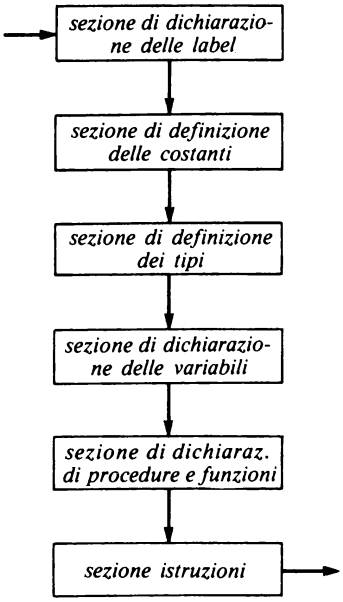


Figura 1.6 Diagramma sintattico di un blocco

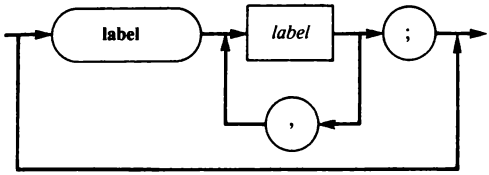


Figura 1.7 Diagramma sintattico della sezione di dichiarazione delle label

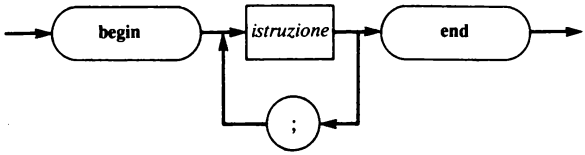


Figura 1.8 Diagramma sintattico della sezione istruzioni

sono normalmente file e variabili, come in Cobol, ma possono anche essere molte altre cose.
L'ordine in cui compaiono le differenti sezioni è fissato rigidamente (si veda la Figura 1.6) e corrisponde alla logica di costruzione progressiva del programma (ad eccezione delle label): ogni parte dichiarativa o di definizione ha bisogno di quella che la precede per poter essere costruita. Tutte le sezioni, ad eccezione dell'ultima, possono essere vuote, come si vede, ad esempio, per la sezione di dichiarazione delle label, in Figura 1.7.
La definizione di costanti e la dichiarazione di variabili sarà trattata nel paragrafo 1.4. La definizione di tipo comincia nel Capitolo 2 e continua nei Capitoli 7, 9, 11, 12 e 13. La dichiarazione di procedure e funzioni è spiegata nel Capitolo 4.
La sezione istruzioni (Figura 1.8) è chiamata anche *corpo* del programma; ne rappresenta la parte eseguibile ed è costituita da una sequenza di istruzioni separate da punti e virgola e racchiuse dai simboli **begin** e **end**. Questo costrutto ha anche il nome di istruzione composta; si veda il paragrafo 1.7.
Il simbolo **begin** nel blocco del programma costituisce il suo punto di ingresso (entry-point). All'attivazione del programma vengono fatte automaticamente alcune operazioni (si veda il paragrafo 8.2).

ESEMPIO 1.7: DUE PROGRAMMI PASCAL MOLTO SEMPLICI

```
program Zero {minimo programma vuoto};
begin
end {Zero}.
```

```
program Primo (output) {Questo è un programma Pascal completo};
begin
   writeln ('Questo e'' un programma Pascal molto semplice')
end {Primo}.
```

COMMENTI

1. Il programma *Zero* è un buon test per misurare il tempo speso da un compilatore Pascal e dal suo supporto a tempo di esecuzione per non fare niente (l'overhead del Pascal). Questo programma non legge dati, non esegue calcoli, non stampa alcun risultato.
2. L'intestazione del programma *Primo* contiene l'identificatore del file standard *output* come parametro, perché il programma vi stamperà qualche cosa (senza neppure nominarlo); la stringa di caratteri racchiusa fra parentesi è stampata dalla procedura predefinita *writeln*, chiamata nell'unica istruzione del programma.
3. In ambedue i programmi le sezioni di definizione e di dichiarazione sono vuote; *output* e *writeln* sono identificatori predefiniti e non devono essere dichiarati. In ambedue i programmi la sezione istruzione contiene una sola istruzione che, nel programma *Zero*, è l'istruzione vuota.

1.4 COSTANTI E VARIABILI

Una costante rappresenta un valore, nell'insieme dei valori universali: *1* rappresenta il valore intero «1»; *true* il valore logico «vero». In Fortran, Cobol, BASIC, PL/1 le costanti sono solo identificatori di valori universali. Nella maggior parte dei linguaggi di programmazione moderni, il programmatore può anche stabilire una corrispondenza fra un identificatore ed un valore universale: le costanti possono cioè avere un nome.
In Pascal questa corrispondenza viene stabilita per mezzo della definizione di costanti nella relativa sezione del blocco (si veda la Figura 1.9).
I valori designati possono essere interi, reali, caratteri o stringhe (si veda la Figura 1.10), ma nel paragrafo 2.3 verrà visto un ulteriore modo per introdurre costanti non universali. Solo le costanti che denotano valori interi o reali possono essere precedute dal segno. Occorre, a questo punto, mettere in evidenza un'importante e spiacevole lacuna del Pascal: non si può definire una costante in funzione di altre costanti, sia universali che dichiarate (ad eccezione dell'operatore di identità e di conversione di segno). Per esempio, potrebbe essere estremamente utile definire una costante *area* come il prodotto delle due costanti *lunghezza* e *larghezza*. Queste operazioni possono solo essere descritte in commenti, senza alcuna garanzia di validità.

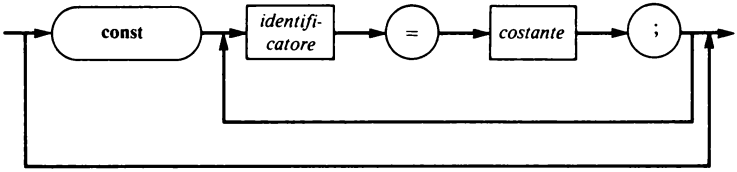


Figura 1.9 Diagramma sintattico della sezione di definizione delle costanti

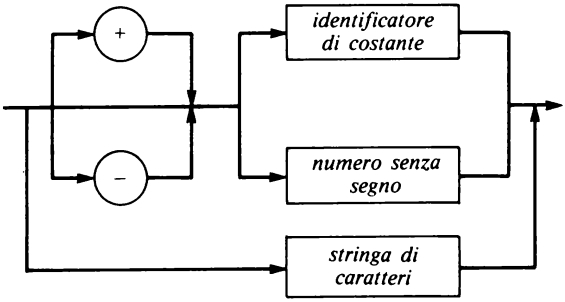


Figura 1.10 Diagramma sintattico di una costante

ESEMPIO 1.8: SEZIONE DI DEFINIZIONE DI COSTANTI

```
const
  pi = 3.1415926536 {valore universale; un reale};
  fine = '***Fine del programma***';
  nposti = 4; ncolori = 6;
  npioli = 24 {npioli = nposti * ncolori};
  finecarattere = '.';
```

COMMENTI

1. Il tipo di una costante è implicito nella sua rappresentazione: *pi* è una costante reale, *fine* è una costante di tipo stringa, *finecarattere* è una costante di tipo *char* e le rimanenti tre costanti sono di tipo intero.
2. Nulla impedisce ad un commento di essere falso; se *npioli* fosse stato posto uguale a 25, la precedente sezione di definizione di costanti avrebbe continuato a mantenere la sua validità.

Le variabili sono oggetti che possono cambiare valore durante l'esecuzione del programma. In Pascal è obbligatoria la dichiarazione di tutte le variabili, al

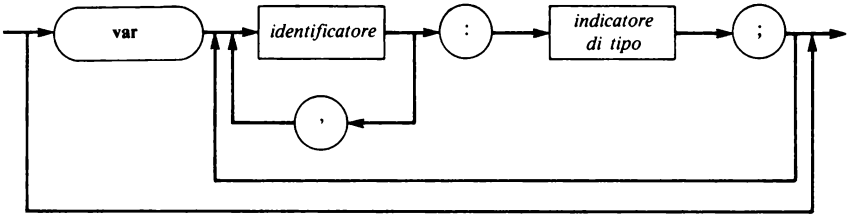


Figura 1.11 Diagramma sintattico della sezione di dichiarazione delle variabili

contrario di quanto avviene in Fortran, BASIC o PL/1 (tale obbligo vale anche per il Cobol, ma in Pascal devono essere definiti o dichiarati tutti gli oggetti e non soltanto le variabili). Non esiste alcuna relazione, implicita o esplicita, fra come si scrive l'identificatore di una variabile ed il tipo del valore che può esserle assegnato: tale associazione viene fatta nel momento in cui si dichiara la variabile (Figura 1.11). In funzione del tipo, una variabile può essere semplice (le variabili semplici sono descritte nel Capitolo 2) o composta (i tipi strutturati sono descritti nei Capitoli 7, 9, 11, 12 e 13). Fra il punto in cui una variabile viene dichiarata e il punto in cui le viene assegnato un valore qualsiasi, il valore di una variabile è indefinito: è un errore cercare di usarla.
Gli identificatori di variabile, ad eccezione del caso in cui compaiono nella sezione di dichiarazione delle variabili (o nella lista di parametri del programma), sono tutti confinati nella sezione istruzioni, e costituiscono dei riferimenti alla variabile: accessi in lettura, se l'identificatore compare in un'espressione; accessi in scrittura, se compare nella parte sinistra di un'istruzione di assegnamento o in qualche chiamata di sottoprogramma (parametro attuale di tipo variabile, alcuni parametri di molte procedure predefinite).

ESEMPIO 1.9

```
program Secondo (input, output);
   var x,y: integer {due variabili intere};
begin {qui x e y hanno valori indefiniti}
   read(x,y) {due valori sono letti dal dispositivo standard di input ed assegnati
              rispettivamente a x e y};
   writeln ('la somma di', x, 'e', y, 'e", x+y)
end {Secondo}.
```

COMMENTI

L'identificatore predefinito *integer* è il nome del tipo che comprende l'insieme di tutti i valori interi; *x* e *y* sono due variabili dichiarate nella sezione di dichia-

razione delle variabili del blocco del programma e referenziate dall'istruzione *read* (che rappresenta un accesso in scrittura alle variabili) e dall'istruzione *writeln* (accesso in lettura).

1.5 CHIAMATE A PROCEDURE STANDARD

Le procedure standard sono sottoprogrammi predefiniti nel linguaggio e di conseguenza sono note ad ogni compilatore conforme allo Standard. Questi sottoprogrammi esistono in tutti i linguaggi; l'esempio più noto è costituito dai sottoprogrammi di input-output: READ e WRITE in Fortran e Cobol, GET e PUT in PL/1, READ e PRINT in BASIC. Tuttavia, in questi linguaggi i corrispondenti sottoprogrammi sono generalmente chiamati usando speciali costrutti sintattici invece delle solite istruzioni di chiamata di procedura. In Pascal il costrutto usato ha l'aspetto generale di una chiamata a procedura, ad eccezione di alcuni dettagli secondari.
Verranno ora presentati brevemente i sottoprogrammi Pascal per la gestione dell'input-output, anche se le procedure saranno trattate in modo completo nel Capitolo 4 e l'input-output nei Capitoli 7 e 8. Negli Esempi 1.7 e 1.9 si è già fatto uso di operazioni di input-output e non possiamo aspettare fino al Capitolo 7 per scrivere programmi che siano effettivamente utilizzabili. Tuttavia la presentazione che segue fornisce solo uno schema di riferimento.
La chiamata di una procedura standard è un'istruzione semplice che comanda l'esecuzione della procedura stessa. È costituita dall'identificatore di procedura seguito da una lista di parametri racchiusi fra parentesi (ad eccezione di *writeln*, che può essere chiamata senza parametri).

1.5.1 INPUT: *read*

La chiamata si effettua nella forma *read*(*v1,v2,..., vn*). I parametri *vi* sono indirizzi di variabili. La chiamata a questa procedura provoca la lettura di *n* valori, *val1, val2,..., valn* dal dispositivo standard di input ed assegna questi *n* valori alle *n* variabili di nome *v1, v2,..., vn*. Il tipo di *vali* deve essere lo stesso di *vi*: intero, reale o carattere. Quando *vi* è una variabile di tipo intero o reale, il processo di lettura è il seguente: vengono saltati tutti gli spazi ed i terminatori di linea (ritorno carrello, fine scheda); non appena si trova un carattere diverso da quelli appena descritti, si verifica che sia un segno o una cifra e, nel caso lo sia, si comincia a costruire *vali* (altrimenti si è verificato un errore e il programma termina con un opportuno messaggio); la lettura termina non appena si incontra un carattere illegale in un numero senza segno (in accordo con il diagramma sintattico del paragrafo 1.2). Se *vi* è una variabile di tipo carattere, *vali* è il valore del primo carattere non ancora letto nelle precedenti fasi di lettura, qualunque esso sia.

1.5.2 OUTPUT SU BUFFER: *write*

La chiamata è *write(e1,e2,..., en)*. I parametri *ei* sono espressioni (una variabile è il più semplice tipo di espressione) i cui valori possono essere un intero, un reale, un carattere, un booleano o una stringa. I valori dei parametri sono appesi, secondo un formato standard di output. La scrittura fisica del buffer non è fatta dalla procedura predefinita *write*.

1.5.3 OUTPUT IMMEDIATO: *writeln*

La chiamata è *writeln(e1,e2,... en)*, dove la lista dei parametri può essere vuota; se ciò accade, *writeln* provoca l'invio, sul dispositivo standard di uscita, del buffer ad esso associato (stampato in linea o fuori linea) e lo svuota rendendolo di nuovo disponibile. Se invece c'è una lista di parametri, *writeln* funziona dapprima come una *write* (appendendo i parametri al contenuto del buffer) e quindi come se non avesse alcun parametro (svuotando d'un colpo il buffer). Scrivere un buffer vuoto equivale ad un salto riga.

ESEMPIO 1.10

```
program Terzo (input, output)
   {questo programma dimostra come funzionano read e write sui dispositivi
        standard};
   var a,c,f: integer;
      b,e: real; d: char {tipo standard per i caratteri};
begin {Terzo}
   read(a,b,c,d,e); read(f)
   {del tutto equivalente a read(a); read(b);...; read(f) oppure a read(a,b,c,d,e,f,)};
   write('I tre valori interi sono'); writeln(a,c,f);
   writeln('La somma dei due valori reali', b, 'e', e, 'e"', b+e);
   writeln('Il valore del carattere d e" stampato fra due asterischi: *',
        d,'*')
end {Terzo}.
```

Se il dispositivo standard di input contiene la sequenza di caratteri:

```
Prima linea:   1  1.0  -1  -0.1e-2
Seconda linea: 4
```

e se il formato implicito previsto dall'implementazione prevede per gli interi otto caratteri e per i reali in virgola mobile sei cifre dopo la virgola, sul dispositivo standard di output verranno stampate le seguenti righe:

```
I tre valori interi sono 1  -1  4
La somma dei due valori reali 0.100000e+01 e 0.100000e-02 e' 0.101000e+01
Il valore del carattere d è stampato fra due asterischi: * - *
```

Invece con i seguenti dati sul dispositivo standard di input:

```
Prima linea:    1  1.0
Seconda linea:  -1
Terza linea:    -0.1E-2  4
```

sul dispositivo standard di output verranno stampate le seguenti linee:

```
I tre valori interi sono 1  -1  4
La somma dei due valori reali 0.100000e+01 e -0.100000e-02 e' 0.999000e+00
Il valore del carattere d è stampato fra due asterischi: *  *
```

COMMENTI

1. La successione delle due chiamate di procedura

   write (*'I tre valori interi sono'*); *writeln*(*a, c, f*)

   è del tutto equivalente alla singola istruzione

   writeln(*'I tre valori interi sono', a, c, f*)

2. Quando non è possibile scrivere una stringa di caratteri su un'unica linea di programma, come nell'ultima chiamata di *writeln*, può essere divisa su due linee, che sono giustapposte nel buffer di output.
3. Con il primo insieme di dati la lettura del valore *-1* in *c* si ferma al segno meno subito dopo *1*; *d*, che è di tipo *char*, avrà quindi il valore «-». Con il secondo insieme di valori la lettura di *-1* termina alla fine della linea. Quindi a *d* viene assegnato il valore del carattere corrispondente, uno spazio.
4. Non c'è alcuna relazione fra il formato della chiamata a *read* e quello dei dati di input: la lettura viene guidata esclusivamente dal tipo delle variabili cui i dati devono essere assegnati.
5. Naturalmente l'output può essere presentato in altri modi, sotto il completo controllo del programmatore, come sarà spiegato nel paragrafo 8.4.

1.6 ISTRUZIONI DI ASSEGNAMENTO

L'istruzione di assegnamento fa sì che una variabile (o un elemento di una variabile composta) o un identificatore di funzione — all'interno del suo corpo — assuma un (nuovo) valore. Si esaminerà ora solo il primo di questi casi.

Sintassi

La regola sintattica per l'istruzione di assegnamento è presentata in Figura 1.12. Nel rettangolo di nome «variabile» può esserci l'identificatore di una variabile semplice o di una variabile strutturata considerata però come un unico oggetto. Altrimenti può rappresentare un componente di una variabile composta che dipende dalla struttura della variabile e che verrà descritto nei capitoli corrispondenti. L'istruzione di assegnamento in Pascal, occorre sottolinearlo, non fa uso del segno di uguale. Questo evita ogni confusione con l'operatore relazionale di uguaglianza (una notazione come I = I + 1, benché sia legale in Fortran o in PL/1, va contro il senso comune). Il simbolo := è comunque ormai comune in molti linguaggi fin dai tempi dell'Algol 60.

Semantica

Se x è una variabile di tipo T e y è un'espressione il cui valore è compatibile con T (vedere poco oltre),
allora $\{P_y^x\}\ x := y\ \{P\}$.

Questo significa che l'antecedente dell'istruzione di assegnamento può essere dedotto dal suo conseguente sostituendo la parte destra dell'espressione in tutte le occorrenze della variabile presente nella parte sinistra come, per esempio nell'istruzione:

$$\{x+1 = 10\}\ x := x+1\ \{x = 10\}$$

COMMENTI

1. L'ordine in cui vengono valutate le due parti dell'istruzione dipende dall'implementazione; un programma corretto non deve perciò fare affidamento su

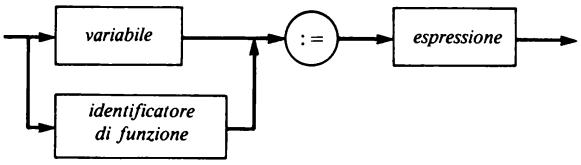


Figura 1.12 Diagramma sintattico di un'istruzione di assegnamento

quest'ordine. Questa osservazione è importante solo nel caso in cui la variabile contenga un'espressione il cui valore può essere modificato durante la valutazione della parte destra dell'espressione, o viceversa. Questa situazione, molto pericolosa, si verifica solo se una funzione modifica i propri parametri o alcune variabili globali, secondo un meccanismo noto come *side effect* (effetto collaterale).

2. Una variabile, immediatamente dopo la sua dichiarazione, ha un valore indefinito che rimane tale fino a che non le venga assegnato un valore di tipo compatibile, in un'istruzione di assegnamento, o in seguito ad una chiamata di una procedura di input.
3. In Pascal, come in Fortran e BASIC, non sono possibili assegnamenti multipli, che sono invece leciti in Cobol e PL/1.

Benché non si sia ancora spiegato bene il concetto di tipo in Pascal, e poiché i vari tipi disponibili nel linguaggio verranno introdotti in capitoli successivi, è tuttavia necessario, per completezza di esposizione, dare un significato preciso al termine «compatibilità di tipo» in un'istruzione di assegnamento.

Un oggetto di tipo *T2* è compatibile con un tipo *T1*, in un'istruzione di assegnamento, se:

1. *T1* e *T2* sono lo stesso tipo e non sono di tipo file (o non contengono campi di tipo file); oppure
2. *T1* è di tipo *real* e *T2* è di tipo *integer*; oppure
3. *T1* e *T2* sono di tipo ordinale ed il valore del tipo *T2* cade nell'intervallo chiuso definito da *T1* (vedere paragrafo 2.4); oppure
4. *T1* e *T2* sono insiemi equivalenti e tutti gli elementi dell'oggetto di tipo *T2* appartengono all'intervallo chiuso definito dal tipo base di *T1* (si veda il Capitolo 12); oppure
5. *T1* e *T2* sono tipi stringa equivalenti (si veda il paragrafo 9.5).

Per il momento è sufficiente ricordare che se x è dichiarata di tipo *integer* e y di tipo *real*, l'assegnamento $y := x$ è legale, mentre l'assegnamento $x := y$ non lo è, poiché il tipo *real* non è compatibile con il tipo *integer*. Questa incompatibilità è giustificata dalla perdita di informazione nella conversione di un reale in un intero e perché esistono almeno due modi più sensati per effettuare questa conversione: per arrotondamento o per troncamento. Di conseguenza quest'operazione deve essere esplicitamente richiesta, nel programma, per mezzo di una delle funzioni di conversione di tipo (si veda il paragrafo 3.4).

1.7 ISTRUZIONI COMPOSTE

Un'istruzione composta è l'insieme di più istruzioni che devono essere eseguite in sequenza e devono essere trattate come un'unica istruzione. Si tratta del primo e del più semplice modo di strutturare le istruzioni che presentiamo per il Pascal. In Fortran e in BASIC, se si devono eseguire più istruzioni in sequenza, occorre separarle con un simbolo di fine istruzione posto alla fine di ogni linea di programma; in PL/1 si ricorre ad un punto e virgola, che è il terminatore di ogni istruzione. Non è invece possibile raggruppare più istruzioni in Fortran ed in BASIC (almeno nelle versioni standard di questi linguaggi), che non hanno alcuna struttura di controllo. In Cobol, d'altra parte, è possibile raggruppare più istruzioni, anche se in un modo piuttosto limitato, in un'istruzione composta che è chiusa da un punto; tuttavia un gruppo di istruzioni composte non può contenerne un altro. In PL/1 l'istruzione composta più semplice è data dal gruppo di DO (senza il controllo sul ciclo) che ha all'incirca, le stesse proprietà dell'istruzione equivalente in Pascal. In questi due linguaggi una tale struttura è necessaria dal momento che la struttura non è completamente parentesizzata.

Sintassi

Si noti, dalla Figura 1.13, che in Pascal il punto e virgola è un separatore di istruzioni — come negli altri linguaggi derivati dall'Algol — e non un terminatore, come in PL/1; inoltre **begin** e **end** non sono istruzioni, ma solo parentesi di un'istruzione.

Semantica

Data l'istruzione composta

begin *S1*; *S2*;...; *Sn* **end**
se per ogni *i* da *1* a *n*, {Pi−1} *Si* {Pi},
allora {Po} istruzione composta {Pn}.

Questo significa che ogni istruzione *Si* può essere considerata un trasformatore di asserzioni (*Si* trasforma *Pi−1* in *Pi*) e che in un'esecuzione sequenziale, il conseguente di un'istruzione rappresenta l'antecedente dell'istruzione successiva. Si può allora dedurre l'antecedente ed il conseguente dell'istruzione compo-

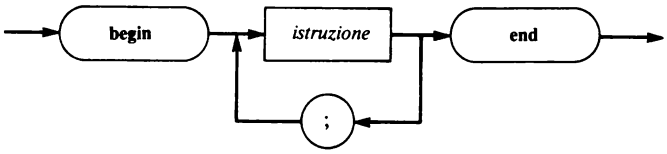


Figura 1.13 Diagramma sintattico per un'istruzione composta

sta stessa e trattarla come una «scatola nera», descritta solo in termini di un trasformatore di asserzioni. Il metodo di procedere appena descritto si può applicare a tutte le istruzioni del Pascal, ad eccezione dell'istruzione **goto**, che interrompe l'ordine di esecuzione sequenziale di un insieme di istruzioni.

1.8 ALTRE ISTRUZIONI SEMPLICI

Le istruzioni semplici sono i costituenti elementari della sezione eseguibile di un programma e possono essere combinate in molti modi in blocchi strutturati, di dimensioni maggiori. Nel paragrafo 1.6 abbiamo già introdotto un primo esempio di istruzione semplice: l'assegnamento. Un'altra istruzione semplice, di importanza fondamentale, è l'istruzione di chiamata di procedura che verrà descritta nel Capitolo 4. In questo capitolo descriveremo invece due altre istruzioni semplici.

L'*istruzione vuota* ha una sintassi estremamente austera (si veda la Figura 1.14). La sua semantica non le è da meno.

{P} istruzione vuota {P}

Un'istruzione vuota non contiene niente e non fa niente. Il suo scopo è: di semplificare alcune descrizioni sintattiche; di permettere la presenza di un punto e virgola davanti ad un terminatore di istruzioni come nel caso del simbolo **end**; di descrivere in modo esplicito alcune situazioni in cui non si deve fare niente (si veda il paragrafo 5.1); di permettere la definizione delle procedure vuote, che sono molto utili nella fase di costruzione e di debugging di un programma.

L'istruzione **goto** (Figura 1.15), tanto necessaria e così usata in Fortran e BASIC (e perfino in Cobol), ha un uso molto limitato in Pascal — a causa della ricchezza di istruzioni di controllo — tanto che, di norma, un programma in Pascal non ne contiene affatto.
L'istruzione **goto** indica che l'elaborazione deve continuare dal punto specificato dalla label, cioè a partire dall'istruzione che è identificata da quella particolare label (si veda il paragrafo 4.1). L'uso dell'istruzione **goto**, considerato una fonte di anarchia nei programmi, è scoraggiato dall'esistenza di pesanti restrizioni che devono essere scrupolosamente osservate. La label che è specificata in un'istruzione **goto** può indirizzare solo:

1. Una delle istruzioni che si trovano nella stessa istruzione composta che contiene il **goto**.

Figura 1.14 Diagramma sintattico di un'istruzione vuota

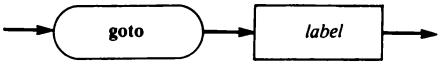


Figura 1.15 Diagramma sintattico di un'istruzione **goto**

2. Un'istruzione composta che contiene il **goto**.
3. Un'istruzione al livello più esterno nella sezione istruzione di un blocco che racchiude il blocco che contiene il **goto**. È così consentito uscire da una procedura ad una procedura più esterna o al programma principale, ma non saltare all'interno di un'istruzione composta.

Nel caso descritto al punto 3, tutti i blocchi da cui si è usciti terminano la loro attivazione (questo argomento verrà trattato ancora nel Capitolo 4). Di conseguenza non si può usare un'istruzione **goto** per saltare dall'esterno all'interno di un'istruzione composta o di una procedura e neppure da un ramo ad un altro in un'istruzione condizionale. Il suo uso normale si limita al trattamento di condizioni di errore all'interno di cicli o di sottoprogrammi, quando l'elaborazione non può proseguire e deve essere interrotta prematuramente e definitivamente. Le label sono state rese uno strumento poco attraente e questo è un ulteriore fattore per scoraggiarne l'uso da parte del programmatore: non sono infatti né variabili, né costanti, né valori, non sono di alcun ausilio mnemonico e devono essere dichiarate proprio all'inizio del blocco in cui sono usate, come se il programmatore dovesse vergognarsene. È lasciato al lettore, come esercizio, il compito di contare le label e le istruzioni **goto** che compaiono in tutti gli esempi (ad eccezione del prossimo!) presentati in questo volume.

ESEMPIO 1.11

```
{questo programma non ha altro significato se non ripetere tutti i costrutti già visti}
program Quarto (input, output);
    label 13;
    var x, y: integer;
begin {la sezione istruzioni è di fatto un'istruzione composta}
    read(y);
    x := y {un'istruzione di assegnamento};
    writeln('valore di x =', x);
    goto 13 {un'istruzione goto, all'interno di un'istruzione composta};
{l'istruzione successiva non verrà mai eseguita};
    y := x;
13: {questa è un'istruzione vuota (fra i due punti e la parentesi graffa sinistra)}
end {Quarto}.
```

2

Tipi semplici

I programmi per calcolatori fanno uso di tre classi differenti di valori universali:

a) numeri per effettuare calcoli;
b) valori logici per test;
c) caratteri stampabili, per comunicare con gli utenti.

Queste tre classi dovrebbero essere presenti in tutti i linguaggi di programmazione ma, a causa di alcuni problemi che sorgono per la rappresentazione interna al calcolatore di questi valori, la situazione è tutt'altro che chiara. Benché esistano due soli valori logici, non sempre questi sono esplicitamente presenti: il Fortran ha un tipo LOGICAL con i valori .TRUE. e .FALSE., ma il BASIC non ha nulla di simile, il PL/1 usa stringhe di bit di lunghezza unitaria e il Cobol offre solo un surrogato di valori logici con le sue variabili di tipo condizione. Nel caso poi dei caratteri, la situazione è ancora più complicata, per l'esistenza di molti insiemi di caratteri diversi. In Cobol, il modo DISPLAY indica la forma stampabile di una variabile, ma si suppone che l'insieme di caratteri scelto sia dipendente dall'implementazione. In Fortran non esiste alcuna notazione per una variabile che contenga uno o più caratteri e si può specificare un particolare formato di rappresentazione per le variabili che contengono stringhe di caratteri solo nelle apposite istruzioni di formattazione dell'input-output.
L'architettura delle unità aritmetiche dei singoli calcolatori impone diverse rappresentazioni per i numeri: gli interi sono vincolati ad appartenere ad un sottoinsieme di N; i reali sono un sottoinsieme finito e limitato di R; i numeri in virgola fissa possono avere anche una rappresentazione decimale compattata o meno. Il Fortran ha interi, reali e numeri in doppia precisione; il BASIC ha interi e reali; il Cobol consente tutte le specificazioni possibili, sebbene non riconosca, almeno nella definizione standard, i reali; il PL/1 consente una miscellanea fra Fortran e Cobol, insieme ad altre possibilità per numeri binari a virgola fissa. Per aumentare la precisione nei calcoli esistono, in Fortran e PL/1, nu-

meri reali a lunghezza multipla, che vengono implementati dall'hardware, se disponibile, o dal software (appositi package o microprogrammi).
Così, nei linguaggi classici di programmazione, i numeri possono essere specificati in molti modi diversi, ma questi stessi linguaggi non offrono praticamente alcun modo per classificare i dati in categorie diverse. In contrasto, il Pascal semplifica il modo di specificare i valori universali ed inoltre fornisce strumenti sia per definire altri insiemi di valori elementari, sia per costruire strutture complesse, a partire da dati elementari, usando dei descrittori di tipo. In questo capitolo presenteremo la definizione e le proprietà dei tipi in generale, quindi esamineremo in dettaglio i tipi semplici.

2.1 DEFINIZIONI DI TIPO

Un tipo specifica il dominio dei valori che possono essere assegnati a variabili di tale tipo e, implicitamente, determina l'insieme di operatori che possono essere applicati a queste variabili. Come regola generale, ambedue gli operandi di un operatore (diadico) devono essere dello stesso tipo, o di tipi compatibili (con l'eccezione descritta nel Capitolo 12). Il compilatore ha così a disposizione molti strumenti per verificare la coerenza delle espressioni e per determinare il significato esatto di molti operatori ambigui. La suddivisione di tutti i valori in classi distinte, disponibile grazie al concetto di tipo, è uno dei contributi più essenziali ai linguaggi di programmazione fatto dal Pascal. Dichiarare una variabile di un dato tipo equivale a fare un'asserzione, sempre valida, su questa variabile, asserzione che il compilatore può verificare automaticamente, solitamente durante la compilazione stessa e, solo in alcuni casi, durante l'esecuzione del programma, per mezzo di controlli poco costosi da realizzare. In questo modo si fornisce al programmatore uno degli strumenti più potenti per aumentarne la fiducia nei confronti dei propri programmi.
Un tipo viene descritto tramite un *descrittore di tipo*, che può apparire sia nella sezione di dichiarazione dei tipi di un blocco — nel qual caso gli viene assegnato un nome che ne consente una successiva identificazione — sia nella dichiarazione di una variabile — nel qual caso il tipo è anonimo e non potrà più essere referenziato. Ogni descrittore di tipo introduce un tipo completamente nuovo e distinto da tutti gli altri, anche nel caso in cui questo nuovo tipo abbia un dominio di valori identico a quello di un altro tipo già definito. Due tipi sono identici se e solo se sono lo stesso tipo o se uno è stato definito con il nome dell'altro, che diventa così noto con due nomi distinti. Occorre osservare esplicitamente che questo meccanismo di assegnamento di nomi ai tipi non esiste assolutamente nei quattro linguaggi di programmazione presi come termine di confronto.
Per completezza verrà ora definito il concetto di *compatibilità di tipo*, benché manchino alcune nozioni che verranno trattate solo in capitoli successivi. Due tipi *T1* e *T2* si dicono compatibili se:

1. *T1* e *T2* sono lo stesso tipo, oppure
2. *T1* è un sottocampo di *T2* o viceversa (si veda il paragrafo 2.5), oppure
3. *T1* e *T2* sono due sottocampi dello stesso tipo, oppure
4. *T1* e *T2* sono ambedue insiemi con lo stesso tipo base (si veda il Capitolo 12) e ambedue devono contemporaneamente essere, o non essere, compattati, oppure
5. *T1* e *T2* sono stringhe con lo stesso numero di componenti (per una discussione sulle stringhe si rimanda al paragrafo 9.5).

La sintassi della sezione di definizione di tipo all'interno di un blocco è illustrata in Figura 2.1. La sezione stessa è poi demarcata da una nuova sezione di dichiarazione o dalla sezione istruzioni del blocco stesso. Nella sezione di definizione dei tipi vengono definiti tutti i tipi locali ad un blocco.
I tipi in Pascal sono distribuiti su tre classi, come si vede in Figura 2.2. Ciascuna delle tre classi può specificare una o più forme di descrizione dei tipi o può ridursi ad un identificatore di tipo. I *tipi semplici*, con i rispettivi descrittori, sono esaminati nei rimanenti paragrafi di questo capitolo. I *tipi strutturati* (o *aggregati*) che sono di fatto dei costruttori di tipo, vengono trattati nei Capitoli 7, 9, 11 e 12. Il *tipo puntatore*, descritto nel Capitolo 13, è ancora un costruttore di tipo, ma ha in più alcune proprietà che lo rendono diverso dai tipi strutturati.

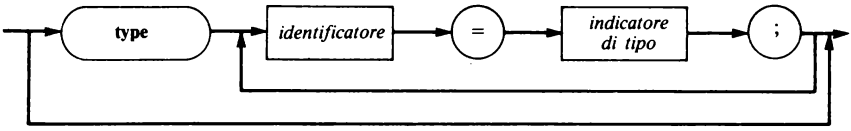


Figura 2.1 Diagramma sintattico della sezione di definizione di tipo

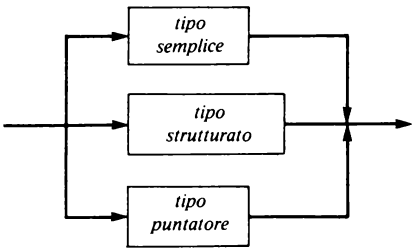


Figura 2.2 Diagramma sintattico di un indicatore di tipo

2.2 GENERALITÀ SUI TIPI SEMPLICI

Esistono parecchi tipi elementari, predefiniti nel linguaggio, che coprono l'insieme dei valori universali, ed esiste anche un costruttore di tipo che consente la definizione, per enumerazione, di nuovi valori non universali: questi sono i tipi semplici disponibili in Pascal. Il valore di un tipo semplice è per definizione semplice, cioè atomico o scalare e non può essere ulteriormente suddiviso in componenti. I tipi semplici possono essere classificati come mostrato in Figura 2.3. La prima linea del diagramma consente di definire un nuovo tipo già definito. I due tipi saranno perciò identici.
I tipi ordinali (Figura 2.4) sono tipi i cui valori possono essere enumerati: essi costituiscono insiemi ordinati e limitati ed il dominio dei loro valori è finito. Per ciascun tipo ordinale esistono tre operatori universali, definiti nel linguaggio come *funzioni predefinite*:

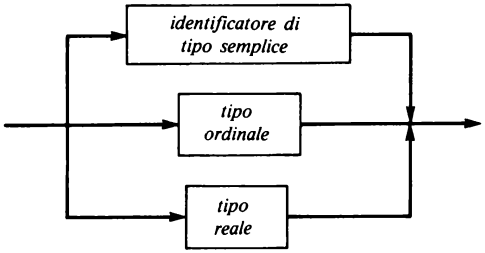


Figura 2.3 Diagramma sintattico di un tipo semplice

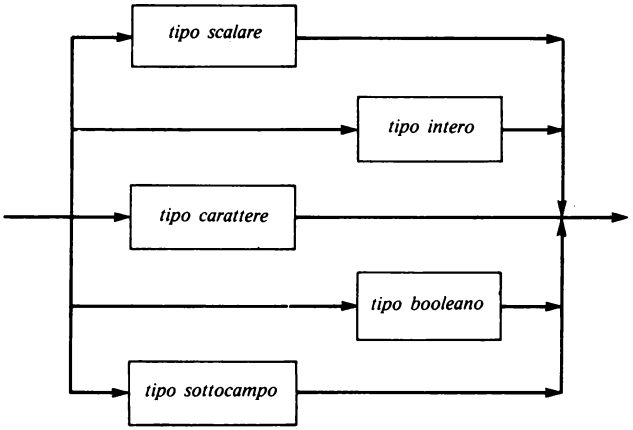


Figura 2.4 Diagramma sintattico di un tipo ordinale

1. *succ*(x): se x è un'espressione di un tipo ordinale T, *succ*(x), se è definita, è un'espressione dello stesso tipo che rappresenta quel valore che segue immediatamente x nel dominio dei valori di T (il successore di x), secondo la relazione di ordinamento definita in T. Se questo valore non esiste, ad esempio nel caso che x sia il massimo valore in T, viene segnalato un errore.
2. *pred*(x): è definita, in modo simile a *succ*(x), come il predecessore di x nell'insieme di valori di T. Non sempre *pred*(x) esiste (quando ad esempio x è il minimo valore in T) ed in tal caso viene segnalato un errore.
3. *ord*(x): è un'espressione di tipo *integer* che indica il numero ordinale associato a x. Nei paragrafi successivi vengono definiti i numeri ordinali per ciascun tipo ordinale.

2.3 TIPI SCALARI

Un tipo scalare (Figura 2.5) definisce un insieme ordinato di valori per elencazione degli identificatori che costituiscono i valori stessi. Questi identificatori diventano identificatori di costanti il cui punto di definizione è il descrittore di tipo stesso (si veda il paragrafo 1.4). L'ordine in cui gli identificatori compaiono nella definizione determina l'ordinamento dei rispettivi valori. Gli operatori definiti sui tipi scalari sono le tre funzioni predefinite *succ*, *pred* e *ord*, che valgono per tutti i tipi ordinali, ed i sei operatori relazionali. Il numero ordinale del primo valore è 0 e il numero ordinale di ciascun elemento si ottiene sommando uno al numero ordinale del suo predecessore.
I tipi scalari, introdotti dal Pascal, sono usati soprattutto per evitare rappresentazioni esoteriche delle informazioni locali ad un programma che non siano valori universali. La rappresentazione esplicita che ne consegue è molto più chiara di quella tradizionale dei numeri interi e rappresenta un importante passo in avanti verso una maggior leggibilità dei programmi, paragonabile all'uso di costanti cui è stato assegnato un nome. Il lettore scettico è invitato a catalogare, negli esempi che compaiono nel presente volume, tutti gli usi di rappresentazioni con numeri interi o con costanti anonime (ad eccezione di 0, 1, *true* e *false*).

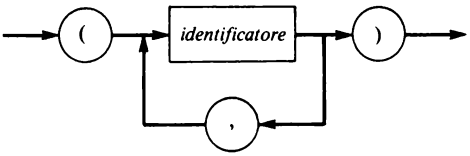


Figura 2.5 Diagramma sintattico di un tipo scalare

ESEMPIO 2.1

```
{sezione di definizione di tipo}
type
   statocivile = (celibe, coniugato, libero, vedovo);
   arcobaleno = (violetto, indaco, blu, verde, giallo, arancio, rosso);
   statoio = (online, offline, timeout, busy, erroreparita, finecarta);
```

COMMENTI

1. Il dominio dei valori del tipo scalare *arcobaleno* è costituito solo dai sette valori indicati come *violetto, indaco, blu, verde, giallo, arancio* e *rosso*. Questi valori sono ordinati: *violetto* < *indaco* < *blu* < *verde* < *giallo* < *arancio* < *rosso* e, per di più, *violetto* = *pred* (*indaco*) = *pred*(*pred*(*blu*)) = ...; e anche *giallo* = *succ*(*verde*) = *succ*(*succ* (*succ*(*succ*(*violetto*)))) e così via.
2. Gli identificatori di costante che indicano i valori di un tipo scalare sono locali al blocco in cui è definito il descrittore di tipo. Di conseguenza:
   a) non è possibile ridefinire questi identificatori all'interno dello stesso blocco;
   b) al di fuori del blocco in cui sono definiti non sono noti, né lo sono, a maggior ragione, al di fuori del programma; i loro valori non possono quindi essere né letti né scritti.

   Se, nell'esempio precedente, si aggiunge la seguente definizione di tipo

   coloreprimario = (*blu, giallo, rosso*)

   si commette un errore, perché i primi tre identificatori di costante, *blu*, *giallo* e *rosso* sono già stati definiti nello stesso contesto nel tipo *arcobaleno*.
3. I tipi scalari sono completamente incompatibili; due tipi differenti non hanno niente in comune e due valori di tipi differenti non possono neppure essere confrontati.

2.4 TIPI ORDINALI PREDEFINITI

I tipi predefiniti sono tipi semplici, derivati dai valori universali. Nello Standard ISO, vengono chiamati *tipi richiesti* e non useremo il termine «tipi standard» a causa della possibile confusione fra ciò che è standard e ciò che non lo è. Naturalmente tutto quello che compare nello Standard del linguaggio è standard. Si suppone che gli identificatori corrispondenti siano definiti nell'ambiente del programma.

2.4.1 IL TIPO *integer*

L'identificatore di tipo predefinito *integer* si riferisce al tipo ordinale il cui dominio di valori è un sottoinsieme dei numeri interi limitato dai valori $-maxint$ e $+maxint$; *maxint* è una costante intera predefinita il cui valore è definito dall'implementazione. Normalmente *maxint* è il più grande intero che si può rappresentare nel calcolatore. I valori di tipo *integer* sono definiti dai diagrammi sintattici del paragrafo 1.2. Gli operatori validi su oggetti di tipo *integer* comprendono tutti gli operatori relazionali, gli operatori aritmetici previsti (si veda il Capitolo 3) e, naturalmente, tutte le funzioni predefinite applicabili ai tipi ordinali. L'ordinale corrispondente ad un intero è l'intero stesso.
Il tipo *integer* può essere visto come un particolare tipo scalare i cui valori non sono rappresentati con identificatori di costanti, ma con la più usuale notazione adoperata per i numeri naturali. Una ragione per cui lo si deve considerare un tipo predefinito è che, altrimenti, non sarebbe possibile definirlo nel linguaggio; un'altra ragione è data dall'esistenza di operatori che richiedono operandi di tipo *integer*.

2.4.2 IL TIPO *Boolean*

L'identificatore di tipo predefinito *Boolean* si riferisce al tipo ordinale il cui dominio di valori è costituito dai soli due valori *false* e *true*. Gli operatori applicabili ad oggetti booleani sono gli operatori logici (si veda il paragrafo 3.2), tutti gli operatori relazionali e le tre funzioni predefinite *pred*, *succ* e *ord*. Tutto funziona come se il tipo *Boolean* fosse definito come

Boolean = (*false*, *true*);

cioè $false < true$, $ord(false) = 0$ e $ord(true) = 1$.
Tuttavia il tipo *Boolean* deve essere predefinito, poiché tutti gli operatori relazionali, qualunque sia il tipo dei loro operandi, danno un risultato di tipo *Boolean* ed anche perché alcune strutture di controllo del Pascal richiedono un'espressione di tipo booleano (si vedano i Capitoli 5 e 6). Queste due situazioni rappresentano la maggioranza dei casi in cui si usano valori booleani esplicitamente o implicitamente in espressioni. Tuttavia, in Pascal, la dichiarazione esplicita di oggetti di tipo *Boolean* è meno frequente che negli altri linguaggi, perché i tipi scalari forniscono uno strumento più generale. Come è dimostrato dai molti esempi presentati in questo volume, si preferisce generalmente l'uso di un indicatore di stato a più valori ai molti flag booleani equivalenti.

2.4.3 IL TIPO *char*

L'identificatore di tipo predefinito *char* si riferisce al tipo ordinale i cui valori sono costituiti dall'insieme di caratteri definiti dall'implementazione. Alcuni di questi caratteri possono non avere una forma rappresentabile. Si ricorda che una costante di tipo *char* viene rappresentata da un carattere racchiuso tra apici. Dal momento che il Pascal non definisce alcun particolare insieme di caratteri (né ipotizza l'esistenza di un insieme di caratteri comune o standardizzato), il particolare insieme di caratteri che si suppone essere disponibile — ed il relativo ordinamento — dipendono dall'implementazione. Tuttavia un programma Pascal che sia indipendente dall'implementazione può contare sulle seguenti relazioni, che devono essere soddisfatte in tutte le implementazioni:

a) le cifre sono sempre presenti, ordinate e contigue, cosicché

$$'0' = pred('1') = pred\ (pred('2')) = \ldots$$

b) le lettere esistono, ma possono anche essere solo di un tipo, maiuscole o minuscole; le minuscole, se presenti, sono ordinate, ma non necessariamente contigue ($'a' < 'b' < 'c' < \ldots$); in modo simile, le maiuscole, se presenti, sono ordinate, ma non necessariamente contigue ($'A' < 'B' < 'C' < \ldots$).

Queste relazioni sono valide per tutti gli insiemi di caratteri di uso corrente, però non si può dire nulla sull'ordinamento reciproco fra maiuscole e minuscole, o sulla posizione relativa di lettere e cifre, o sulla posizione dello spazio nell'insieme dei caratteri. La relazione di ordinamento debole, stabilita per le lettere, è un'eredità dell'insieme di caratteri EBCDIC dove, per esempio *succ* $('I') \neq 'J'$. Benché lo Standard Pascal sia stato deciso dall'ISO, non fa uso, per i caratteri dello standard, della ISO stessa, la cui variante americana, nota come ASCII, è probabilmente l'insieme di caratteri più largamente usato e che gode di molte auspicabili proprietà.
Gli operatori applicabili al tipo *char* sono le tre funzioni predefinite *succ*, *pred* e *ord*. L'ordinale corrispondente ad un carattere dipende dall'implementazione ed è tale che l'ordinamento di due caratteri è lo stesso dei rispettivi ordinali. Un'ulteriore funzione predefinita, $chr(x)$, fornisce il carattere il cui ordinale è x, naturalmente se tale carattere esiste. Il tipo *char*, al pari del tipo *integer*, può essere considerato alla stregua di un particolare tipo scalare i cui valori non sono degli identificatori, ma delle notazioni speciali.

2.5 TIPI SOTTOCAMPO

Un tipo sottocampo definisce un nuovo tipo il cui dominio di valori è un sottoinsieme del dominio dei valori di un altro tipo ordinale, detto *tipo ospite*. Il descrittore di tipo specifica i limiti inferiore e superiore del nuovo tipo come

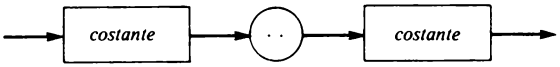


Figura 2.6 Diagramma sintattico di un tipo sottocampo

delle costanti, che sono contemporaneamente sia di questo tipo che del tipo ospite (si veda la Figura 2.6). Perciò queste costanti fanno parte del dominio di valori del tipo sottocampo. Il limite inferiore non può, inoltre, essere maggiore del tipo superiore.
Le variabili di un tipo sottocampo ereditano tutte le proprietà del relativo tipo ospite, fra cui ordinamento, operatori e funzioni. La definizione di un tipo sottocampo è un caso particolare molto importante di asserzione su una variabile, perché:

1. Si autodocumenta ed è manifesto.
2. È automaticamente verificabile: il compilatore può infatti controllare, durante la compilazione, che per ciascun assegnamento — la cui parte sinistra sia di un sottocampo — la corrispondente parte destra non ecceda i limiti dichiarati e può inoltre generare un codice (generalmente semplice e poco costoso) che faccia questi controlli durante l'esecuzione del programma.

I tipi sottocampo sono molto utili soprattutto quando il tipo ospite è il tipo predefinito *integer*, perché costituiscono uno dei metodi più semplici e sicuri per ridurre gli errori nei programmi. È infatti estremamente raro che un programma faccia uso del tipo *integer* in tutta la sua estensione e trovare una dichiarazione con questo tipo è spesso sintomo di programmatori pigri o addirittura trasandati nel loro modo di lavorare. Naturalmente, definire in modo esatto il sottoinsieme richiesto da una data variabile richiede del tempo, ma ne vale realmente la pena.
La dichiarazione di una variabile di un tipo sottocampo suggerisce inoltre al compilatore che è possibile risparmiare memoria per la sua rappresentazione. Questo risparmio può essere molto importante per tipi strutturati di grandi dimensioni i cui elementi siano di un tipo sottocampo.

ESEMPIO 2.2

```
{sezione di definizione di tipo con sottocampi}
type
   indice = -10..+10 {tipo ospite = integer};
   letteraesa = 'A'..'F' {tipo ospite = char};
   arcobaleno = (violetto, indaco, blu, verde, giallo, arancio, rosso);
   tintecalde = giallo..rosso {tipo ospite = arcobaleno};
```

COMMENTI

1. Il tipo ospite di ciascun tipo sottocampo è definito implicitamente dal tipo dei limiti del tipo sottocampo.
2. Sarebbe molto meglio definire, per il tipo *indice*, una costante intera di valore *10*, da usare nel descrittore di tipo. Qual è, altrimenti, il significato di quel *10*? Lo si deve interpretare in relazione alla base dei numeri decimali, o al numero di caratteri di un nome, o a qualche cosa d'altro?
3. Se un tipo sottocampo è definito come *estremoinferiore..estremosuperiore*, le due funzioni *pred*(*estremoinferiore*) e *succ*(*estremosuperiore*) possono anche essere definite, ma sicuramente il loro valore non appartiene al sottocampo. Infatti *pred* e *succ* forniscono un valore che, se esiste, appartiene al tipo ospite. Con una variabile *c* di tipo *tintecalde*, per esempio, il cui valore sia *giallo*, la funzione *pred*(*c*) esiste, ma non è del tipo *tintecalde*; è invece di tipo *arcobaleno* e non può essere assegnata ad alcuna variabile di tipo *tintecalde*. Se *c* = *rosso*, *succ*(*c*) non esiste e la sua valutazione porterebbe ad un errore.

2.6 IL TIPO *real*

L'identificatore di tipo predefinito *real* si riferisce al tipo semplice il cui dominio di valori è il sottoinsieme, finito, limitato e definito dall'implementazione, dei numeri reali. I valori di questo tipo sono definiti in base al diagramma sintattico del paragrafo 1.2. Questo tipo è atomico ma non è ordinale, per cui le funzioni *pred*, *succ* e *ord* non possono essere applicate.
Sul tipo *real* sono definiti gli operatori relazionali e aritmetici. Esistono due funzioni di conversione per trasformare un valore di tipo *real* in un valore di tipo *integer* (si veda il paragrafo 3.4). Inoltre esistono altre funzioni predefinite per le operazioni aritmetiche.
Si noti che il Pascal, che non ha l'ambizione di essere un linguaggio di programmazione universale, non fornisce modi diversi di rappresentazione di un numero reale, che diano una precisione crescente, poiché non si potrebbe garantirne un'implementazione compatibile su calcolatori diversi. Poiché il tipo *real* non è ordinale, non può essere utilizzato come tipo ospite per tipi sottocampo perché, al contrario di quanto accade per gli interi, una tale realizzazione sarebbe troppo costosa.

3
Espressioni

Un'espressione è un valore o un insieme di valori ed operatori su di essi definiti che forniscono un unico risultato. I valori possono essere costanti, variabili, o campi di variabili, e anche il valore calcolato di una funzione. Quando si devono calcolare più valori in un'espressione, questi devono essere tutti o di tipo semplice o di tipo *set*. Le espressioni e gli operatori applicabili ai tipi set saranno discussi nel Capitolo 12.

3.1 REGOLE DI PRIORITÀ E COMPOSIZIONE DEGLI OPERATORI

Un'espressione viene valutata seguendo le regole definite nel linguaggio (*priorità degli operatori*) e in matematica (*composizione degli operatori*). Nei diagrammi sintattici illustrati in Figura 3.1 sono mostrati quattro livelli di priorità. Le costanti senza segno sono state descritte nel paragrafo 1.4. I designatori di funzione compaiono nel paragrafo 3.3, i costruttori di set nel Capitolo 12 e gli identificatori di limiti di un array nel paragrafo 9.7; le variabili sono infine descritte nei Capitoli 7, 9, 11 e 13.
Qui di seguito sono elencati gli operatori in ordine di priorità decrescente:

negazione logica	**not**	priorità più elevata
operatori di moltiplicazione	*, /, **div**, **mod**, **and**	
operatori di somma	+, −, **or**	
operatori relazionali	=, ≠, ≤, <, >, ≥	priorità più bassa

Si ricorda che la rappresentazione raccomandata dallo Standard ISO fa uso di una combinazione di caratteri diversi per ≠, ≤ e ≥ (si veda il paragrafo 1.2). In Pascal, contrariamente a quanto avviene in molti linguaggi di programmazione, gli operatori logici sono allo stesso livello degli operatori aritmetici, e

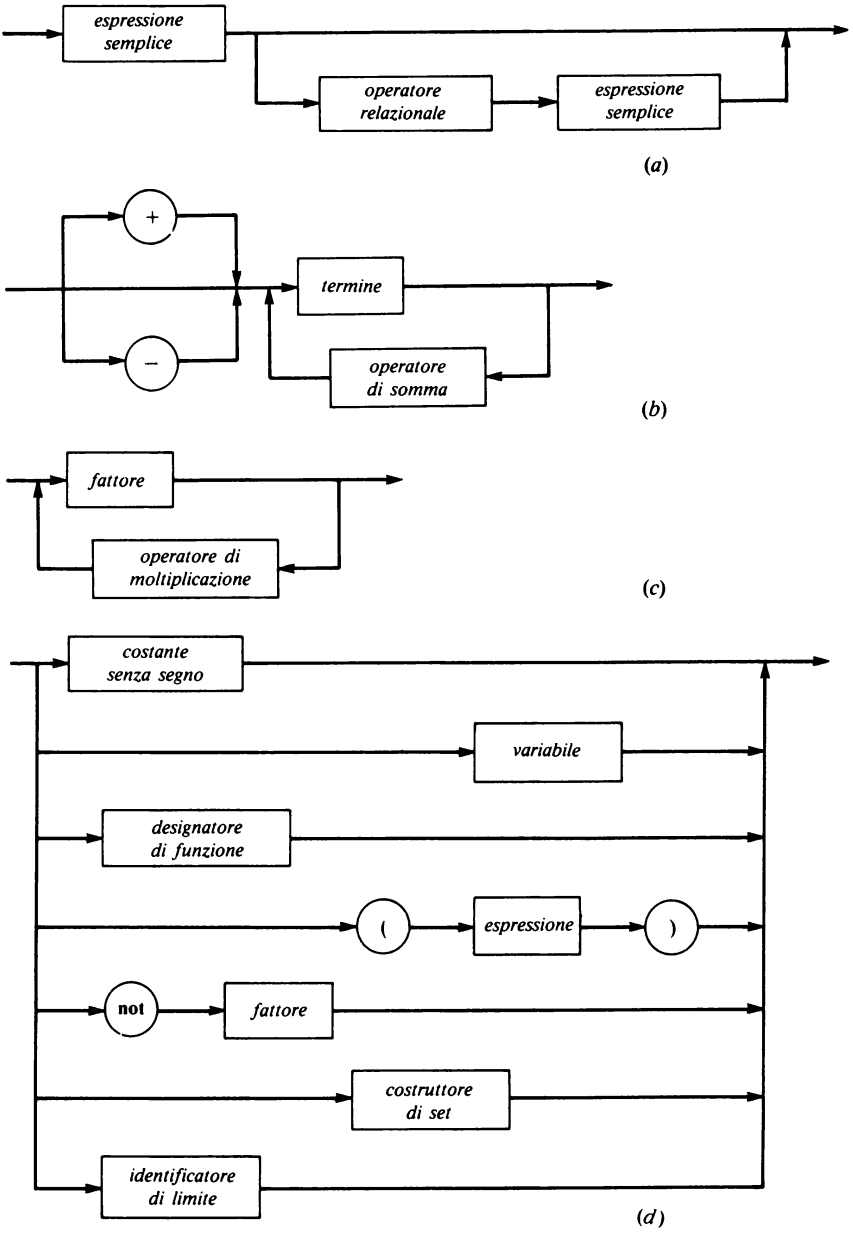


Figura 3.1 I quattro livelli di priorità: (*a*) espressioni; (*b*) espressioni semplici; (*c*) termine; (*d*) fattore

precisamente **and** ha la stessa priorità degli operatori di moltiplicazione e **or** degli operatori di somma. Si è fatto questo per consentire l'uso degli operatori di confronto con valori booleani, specialmente = nell'uguaglianza logica. Una sfortunata conseguenza di tutto ciò è che in Pascal, in certe operazioni di confronto, è necessario fare uso delle parentesi che in altri linguaggi sarebbero inutili. Tuttavia, dal momento che un'espressione è valida solo se ogni operatore ha tutti gli operandi compatibili con il tipo richiesto (si veda il paragrafo 2.1), la maggior parte degli errori che è possibile commettere viene rilevata direttamente dal compilatore.

ESEMPIO 3.1

Siano *a*, *b*, e *c* variabili di tipo *integer* (o di un suo sottocampo):

$a+1$ è un'espressione semplice (di tipo *integer*);

$a \leq b$ è un'espressione di tipo *Boolean*;

$-b/2*a$ è un'espressione semplice (di tipo *real*: si veda il paragrafo 3.2), valutata come $(((-b)/2)*a)$ e non come $(-b)/(2*a)$;

$(a+b)$ **or** c è un'espressione semplice non corretta (anche se sintatticamente valida) a causa dell'incompatibilità di tipo fra operatore ed operandi;

$(a-b)$ è un fattore (di tipo *integer*).

COMMENTI

Ogni espressione racchiusa tra parentesi diventa un fattore: ciò consente al programmatore di scrivere espressioni di qualsiasi tipo, arbitrariamente complesse. Così l'espressione Fortran

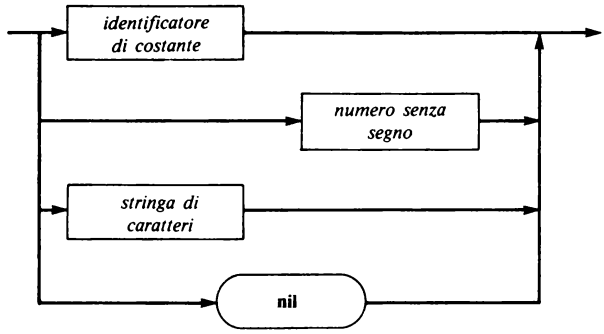
```
A .EQ. B .AND. A .GE. C
```

in Pascal deve essere scritta nel modo seguente:

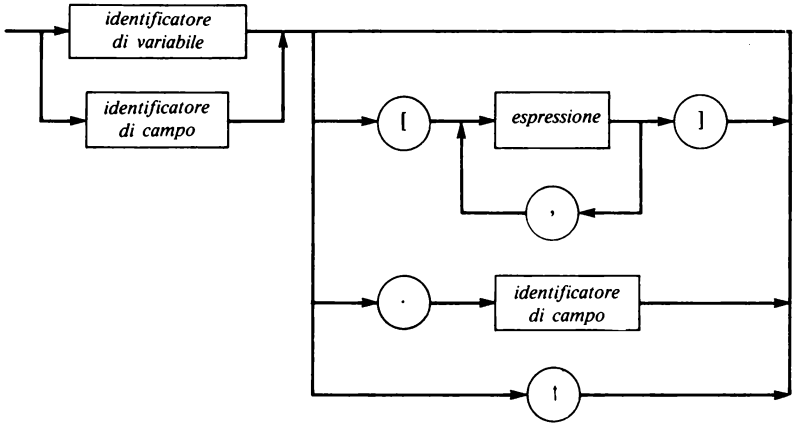
$(a = b)$ **and** $(a \geq c)$

Tuttavia le parentesi non costano nulla e nei programmi normali è abbastanza raro trovare espressioni complesse, così ci sentiamo di suggerire al programmatore di usare liberamente le parentesi per evitare qualsiasi ambiguità di lettura del programma e per ridurre il numero di errori potenziali.

Le componenti elementari di un'espressione sono descritte nelle regole sintattiche date per i fattori. Le scelte possibili riguardano le costanti senza segno (secondo la regola sintattica del paragrafo 1.4) o il simbolo **nil**, che rappresenta



(*a*)



(*b*)

Figura 3.2 Alcuni componenti elementari di un'espressione: (*a*) costante senza segno e (*b*) variabile

un valore associabile ad una variabile di tipo puntatore (si veda il Capitolo 13). Un'altra scelta possibile riguarda le variabili (nel qual caso si usa il valore della variabile referenziata), e più precisamente un identificatore di variabile (se l'espressione non contiene alcun identificatore), un identificatore di variabile semplice, un identificatore di limiti, o una catena di riferimenti che denotano un elemento di una variabile strutturata. La sintassi relativa ad una catena di questo genere è illustrata in Figura 3.2, ma gli elementi che costituiscono la regola sintattica sono trattati in dettaglio nei capitoli relativi alle corrispondenti strutture dati. I designatori di funzione e i costruttori di set sono trattati rispettivamente nel paragrafo 3.3 e nel Capitolo 12.

Il tipo di un'espressione dipende dal tipo dei fattori di cui è composta, dalla natura degli operatori utilizzati (per esempio un operatore relazionale dà sempre un risultato di tipo *Boolean*) e dall'applicazione delle regole di compatibilità di tipo, definite nel paragrafo 2.1. Così, qualsiasi fattore il cui tipo sia un sottocampo di qualsiasi tipo ospite T è considerato di tipo T. L'«espansione» di un tipo sottocampo al suo tipo ospite viene fatta in modo automatico (e senza alcun lavoro aggiuntivo), mentre la «contrazione» inversa, al tipo sottocampo, viene fatta immediatamente prima dell'assegnamento ad una variabile di tipo sottocampo (e ciò implica, nella maggioranza dei casi, un controllo a tempo di esecuzione).
Il risultato di un'espressione di tipo *integer* è matematicamente corretto se cade nell'intervallo $[-maxint..\ +maxint]$; in caso contrario non può essere calcolato e viene rilevata una condizione di errore. Il risultato di un'espressione di tipo *real* è sempre solo un'approssimazione (le operazioni sui reali sono operazioni esatte su argomenti approssimati e quindi restituiscono valori approssimati). La precisione delle operazioni sui reali dipende dalle particolari proprietà dell'unità aritmetica che esegue il programma e anche dalla particolare implementazione del Pascal, dal momento che è possibile implementare i reali sia in semplice che in doppia precisione, se sono ambedue disponibili sulla macchina. Il linguaggio, in sé, non fornisce alcun modo per specificare la precisione voluta, quando si opera su numeri reali, perché una tale specificazione sarebbe troppo dipendente dalla macchina.
Ogniqualvolta viene richiesta un'espressione o un fattore di tipo *real*, si può fornire un valore di tipo *integer*; la conversione viene fatta in modo automatico. D'altro canto non c'è alcun modo per convertire, in modo automatico, un valore reale in un intero, e bisogna chiamare esplicitamente una funzione di conversione di tipo (si veda il paragrafo 3.4).
La priorità degli operatori specifica parzialmente l'ordine in cui le operazioni devono essere svolte; per esempio, in $a*b+c$ la moltiplicazione viene eseguita prima della somma. Tuttavia, l'ordine in cui vengono valutati gli operandi di un operatore dipende dall'implementazione: l'ordine può infatti essere quello in cui gli operandi compaiono nel testo, o l'ordine inverso, oppure gli operandi possono essere valutati in parallelo e, nel caso di operatori booleani, può essere sufficiente valutare un solo operando. Per esempio in $a*b+c/d$ la moltiplicazione può essere eseguita prima, dopo o contemporaneamente alla divisione. Ciò significa che nessun programma può basarsi su quest'ordine. In un'espressione booleana come *a* **and** *b* (o *a* **or** *b*), in alcune implementazioni si valuta solo uno degli operandi, se ciò è sufficiente per determinare il risultato, mentre in altre vengono sistematicamente valutati ambedue gli operandi. Questa regola non consente la composizione di alcuni tipi di condizioni, il cui uso è naturale, e che sono accettate su alcune implementazioni, mentre su altre non funzionano assolutamente (si veda l'Esempio 11.6).

3.2 OPERATORI ARITMETICI, LOGICI E RELAZIONALI

Gli operatori aritmetici si dividono in due categorie

a) operatori monadici +, −
b) operatori diadici +, −, *, **div**, **mod**, /

Le tabelle seguenti ne chiariscono il significato e specificano il tipo dei loro operandi.

Operatore	Operazione	Tipo dell'operando	Tipo del risultato
+	identità	intero o reale	lo stesso dell'operando
−	conversione di segno		

Operatore	Operazione	Tipo degli operandi	Tipo del risultato
+	addizione	intero o reale	intero se ambedue gli operandi sono interi, altrimenti reale
−	sottrazione		
*	moltiplicazione		
/	divisione	intero o reale	reale
div	divisione fra interi	intero	intero
mod	modulo		

Come si è visto nel paragrafo precedente, un valore di tipo sottocampo del tipo *integer* viene automaticamente considerato di tipo *integer*, per cui non compare in queste tabelle. Occorre inoltre sottolineare che in Pascal non esiste un operatore dedicato all'elevamento a potenza e ciò per due ragioni: in primo luogo non è possibile sapere, durante la compilazione, il tipo del risultato dell'elevamento a potenza di un intero con un intero, che dipende dal segno dell'esponente (il risultato è intero con esponente positivo, reale in caso contrario). In secondo luogo l'elevamento a potenza di interi viene usato troppo spesso in modo sbagliato dai programmatori: in realtà sono poco frequenti i casi in cui l'esponente è maggiore di due. Per questo caso particolare, il Pascal mette a disposizione la funzione predefinita *sqr*; per l'elevamento di un numero reale x alla potenza reale y, si può usare l'espressione $exp(ln(x)*y)$, come riportato nell'Esempio 3.2. L'operatore / indica una divisione fra reali con arrotondamento. Il risultato è reale indipendentemente dal tipo degli operandi, se il divisore è zero viene segnalato un errore. L'operatore **div** calcola la divisione fra interi con arrotondamento per difetto: i **div** j è la parte intera del quoziente $i : j$ e vale zero se $abs(i) < abs(j)$.

Vale la seguente relazione:

$$abs(i) - abs(j) < abs((i \textbf{ div } j) * j) < abs(i)$$

Ambedue gli operatori devono essere di tipo *integer* e, se il divisore è zero, viene segnalato un errore. Il risultato è definito anche con operandi negativi, anche se lo si calcola più per ragioni di completezza che per altro, essendo questo un caso abbastanza insolito (inoltre il suo significato matematico ha due definizioni fra di loro incompatibili). Il valore di i **mod** j, per operandi positivi, è il resto della divisione intera di i per j. Più precisamente j deve essere strettamente maggiore di zero, altrimenti si incorre in un errore; i **mod** $j = i - k * j$ con k tale che $0 < i$ **mod** $j < j$; se $i > 0$ vale la solita relazione:

$$(i \textbf{ div } j) * j + i \textbf{ mod } j = i$$

Si noti che i tre operatori $+$, $-$ e $*$ sono anche definiti, con un significato diverso, su operandi di tipo set (si veda il Capitolo 12).
Gli operatori booleani sono **not**, **and** e **or**: il primo è monadico, gli altri due sono diadici. In Pascal non esistono gli operatori logici di equivalenza ($\equiv$) e di implicazione ($\supset$). Tuttavia, dal momento che il tipo *Boolean* è un tipo scalare con *false* < *true*, l'operatore relazionale = coincide con l'operatore di equivalenza; in modo analogo < denota anche l'implicazione logica. Per esempio, $(a < b) = (c < d)$ è vera se entrambi i confronti sono contemporaneamente o veri o falsi; altrimenti è falsa. In ogni modo questi operatori vengono usati raramente nei programmi ordinari.

Operatore	Operazione	Tipo degli operandi	Tipo del risultato
not	negazione	booleano	booleano
or	somma logica		
and	prodotto logico		

Nella tabella che segue sono elencati gli operatori relazionali. I quattro operatori $=$, $\neq$, $\leq$ e $\geq$, insieme all'operatore **in** (che non compare in tabella) sono definiti anche per operandi di tipo set (si veda il Capitolo 12). Gli operandi di operatori relazionali devono essere di tipo compatibile (si veda il paragrafo 2.1) o tali che un operando sia di tipo *real* e l'altro di tipo *integer* (o un suo sottocampo). Per i tipi stringa si rimanda al paragrafo 9.5, per i tipi set e puntatore, rispettivamente ai Capitoli 12 e 13.
Non è possibile comporre, nel senso matematico del termine, gli operatori relazionali. Per verificare se un intero x cade nell'intervallo intero $[a..b]$ si deve

scrivere l'espressione composta

$$(a \leq x) \textbf{ and } (x \leq b)$$

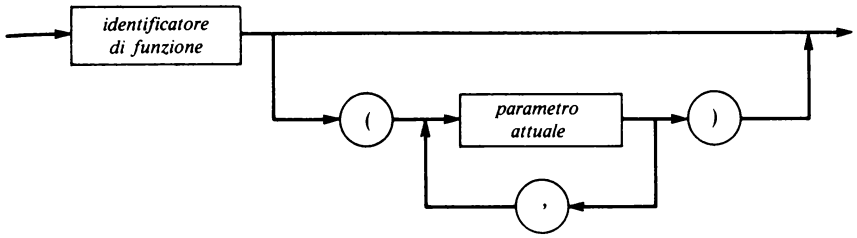
dal momento che $a \leq x \leq b$ è un'espressione illegale. Sarebbe infatti interpretata come $(a \leq x) \leq b$ e la prima espressione è di tipo *Boolean*, che non può quindi essere confrontata con l'intero *b*. Il problema è analogo in Fortran e PL/1 (per di più il PL/1 accetterebbe A<=X <= B attribuendole però un significato assurdo). Solo il Cobol consente di utilizzare la notazione comunemente usata.

Operatore	Operazione	Tipo degli operandi	Tipo del risultato
=	uguale a	qualsiasi tipo semplice, puntatori, stringhe o set	booleano
≠	diverso da		
<	minore di	qualsiasi tipo semplice o stringhe	
>	maggiore di		
≤	minore o uguale a	qualsiasi tipo semplice, stringhe o set	
≥	maggiore o uguale a		

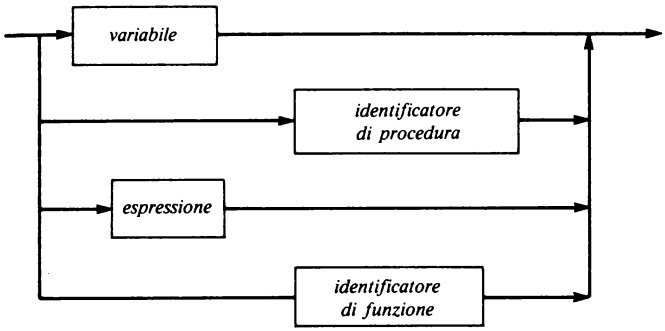
3.3 DESIGNATORI DI FUNZIONE

Un fattore può essere anche il valore calcolato da una funzione. Un designatore di funzione specifica l'attivazione di una funzione con quel nome. In questo paragrafo verranno presentati i designatori di funzione anche se i concetti di sottoprogramma e di attivazione verranno trattati solo nel prossimo capitolo (essi sono già comparsi nel paragrafo 2.2 a proposito delle funzioni predefinite *succ*, *pred* e *ord*).

Un designatore di funzione specifica l'attivazione di una funzione che può essere predefinita (come *succ* o *sin*) o dichiarata nella sezione del blocco riservata alla dichiarazione di procedure e funzioni. Se la funzione ha parametri formali, il designatore di funzione deve avere una lista di parametri attuali, che sono i sostituti dei corrispondenti parametri formali presenti nella dichiarazione della funzione (si veda la Figura 3.3). La corrispondenza fra parametri formali e parametri attuali è posizionale, come avviene nella maggior parte dei linguaggi di programmazione. Tutti i parametri attuali devono essere presenti (non esiste alcun meccanismo di passaggio dei parametri implicito) e devono essere dello stesso tipo del corrispondente parametro formale. Questa corrispondenza verrà



(*a*)



(*b*)

Figura 3.3 (*a*) Designatore di funzione e (*b*) parametro attuale

spiegata in dettaglio nel paragrafo 4.4. Ogni parametro attuale deve essere valutato o referenziato e quindi trasmesso alla funzione, ma l'ordine in cui queste operazioni vengono fatte, per ogni parametro, dipende dall'implementazione; un programma non può quindi basarsi su un ordine particolare. Nel prossimo paragrafo e, soprattutto, nell'Esempio 3.2, verranno introdotti parecchi designatori di funzione.

3.4 FUNZIONI PREDEFINITE

Le funzioni predefinite sono dichiarate, per definizione, nell'ambiente del programma. Esse comprendono alcuni convertitori di tipo (*trunc*, *round*, *ord*, *char*) e le comuni funzioni matematiche (*odd*, *abs*, *sqr*, *sqrt*, *sin*, *cos*, *arctan*, *ln*, *exp*). Nessuna di queste funzioni può essere scritta direttamente in Pascal senza modificare il linguaggio stesso, perché il tipo dei loro risultati dipende dal tipo dei loro parametri attuali. Molte di queste funzioni si possono considerare come operatori monadici che non seguono la sintassi definita sugli operatori.

3.4.1 FUNZIONI DI CONVERSIONE DI TIPO

trunc (x) restituisce il valore intero ottenuto troncando il valore reale x (cioè la parte intera di x). Se il risultato non è di tipo *integer*, se non cade cioè nell'intervallo chiuso $[-maxint..+maxint]$, viene segnalato un errore.

round(x) restituisce il valore intero ottenuto per arrotondamento del valore reale x secondo la seguente regola: *trunc*$(x+0.5)$ se $x > 0$ e *trunc*$(x-0.5)$ se $x < 0$.
Esempi: *trunc*(2.6) vale 2; *trunc*(-2.6) vale -2; *round*(2.6) vale 3; *round*(-2.6) vale -3.

ord(x) è definita con x di tipo ordinale qualsiasi. Se x è intero, *ord*(x) vale x. Se x è di tipo *char*, *ord*(x) è l'intero corrispondente al rango di x nell'enumerazione dei valori di tipo *char*; il primo valore ha rango zero. Se x è di tipo scalare, *ord*(x) è l'intero corrispondente al rango di x nell'enumerazione dei valori di questo tipo; il primo valore ha rango zero.

chr(x) restituisce un valore di tipo *char*, corrispondente all'ordinale x; se questo carattere non esiste, viene segnalato un errore. La seguente relazione vale per qualsiasi carattere *ch:*

$$ch(ord(ch)) = ch$$

3.4.2 FUNZIONI MATEMATICHE

Per tutte le seguenti funzioni, il parametro attuale può essere sia di tipo *real* che di tipo *integer*. Il risultato è di tipo *real* ad eccezione delle funzioni *abs* e *sqr*, nelle quali il risultato è dello stesso tipo del parametro attuale.

abs(x) restituisce il valore assoluto di x
sqr(x) restituisce il valore di x elevato alla potenza di *2*
sqrt(x) restituisce il valore della radice quadrata di x (se $x < 0$ dà errore)
ln(x) restituisce il logaritmo naturale di x (se $x < 0$ dà errore)
exp(x) restituisce l'esponenziale di x in base naturale
sin(x) restituisce il seno di x con x espresso in radianti
cos(x) restituisce il coseno di x con x espresso in radianti
arctan(x) restituisce il valore, in radianti, dell'arcotangente di x nell'intervallo $[-\pi/2..\pi/2]$

Il seguente predicato (funzione con risultato di tipo *Boolean*) richiede un parametro di tipo *integer*:

odd(x) vale *true* se x è dispari, *false* altrimenti; è equivalente all'espressione *abs*(x) **mod** *2* = *1*.

ESEMPIO 3.2

```
{date le seguenti dichiarazioni}
const pi = 3,1415926536;
type giorno = (lun,mar,mer,gio,ven,sab);
var ch: char; i: integer;
    tgx, x, gradopieno, y, z: real;
    xgradi, xprimi: integer;
begin
    ...
    {quindi, dopo aver letto una cifra:}
    read(ch);
    {si calcola il valore dell'intero corrispondente alla cifra appena letta:}
    i := ord(ch)-ord('0');
    ...
    {ord(succ(mer)) è un'espressione con valore 3}
    ...
    read(x) {x è un angolo in radianti};
    tgx := sin(x)/cos(x) {calcola il valore della tangente di x};
    {siano xgradi e xprimi il valore di x in gradi e primi}
    gradopieno := x*360/pi;
    xgradi := trunc(gradopieno);
    xprimi := round((gradopieno-xgradi)*360);
    write(exp(z*ln(y))) {scrive il valore di y elevato alla potenza z}
    ...
end
```

4

Procedure e funzioni

Il concetto di sottoprogramma è probabilmente il più potente e utile fra quelli a disposizione di un programmatore. Tuttavia, prima di quell'invenzione rivoluzionaria nella filosofia della programmazione, nota come «programmazione strutturata», la sua importanza non era stata compresa a fondo. Nei primi linguaggi di programmazione, un sottoprogramma era considerato solamente un artificio per ridurre la lunghezza del programma quando si doveva eseguire un'azione complessa in molti punti diversi. Scrivere una procedura, per poi chiamarla una sola volta, era spesso considerata una pura perdita di tempo e si scoraggiava l'uso dei parametri a causa della complessità e dell'inefficienza che essi comportavano.
L'aumento dei costi legati allo sviluppo e al mantenimento del software hanno provocato uno spostamento dell'enfasi dalla pura efficienza dei programmi alla loro affidabilità, manutenibilità e portabilità e i sottoprogrammi vengono ora considerati un importante strumento per strutturare i programmi. I principi generali della progettazione top-down richiedono, per risolvere un problema, di scomporlo in un insieme di sottoproblemi. Se si risolve ciascuno di questi sottoproblemi, si sarà anche risolto il problema originario con una semplice operazione di composizione delle soluzioni parziali. In modo simile, ciascun sottoproblema può essere scomposto, se necessario, in un insieme di sottosottoproblemi e così via. Il sottoprogramma diventa così l'equivalente linguistico del sottoproblema.
Presumiamo che il lettore sappia cosa sia un sottoprogramma e il solo scopo delle precedenti osservazioni è di spiegare perché un linguaggio come il Pascal, progettato alla fine degli anni '60, tratti i sottoprogrammi in una maniera per certi aspetti diversa da quella di linguaggi progettati 10 o 15 anni prima. In Fortran un sottoprogramma, chiamato SUBROUTINE o FUNCTION, è soprattutto un modo per ridurre le dimensioni dell'unità di compilazione e non uno strumento per strutturare i programmi. I diversi sottoprogrammi che costituiscono un programma sono tutti allo stesso livello — non essendo possibile inserire sottoprogrammi in altri sottoprogrammi — e la comunicazione fra que-

ste diverse parti del programma viene quasi sempre fatta attraverso aree dati comuni e non per passaggio di parametri. In BASIC e in Cobol, le procedure sono state aggiunte molto tempo dopo la loro prima definizione, risentono di molte limitazioni e di diverse restrizioni e presentano indubbi svantaggi. Il PL/1, che in parte fa uso di idee prese dall'Algol 60, dà al concetto di sottoprogramma (chiamato procedura) molto più risalto, ma eredita dal Fortran uno scadente progetto dei parametri che rende difficile la comunicazione fra i sottoprogrammi.
Il Pascal eredita direttamente dall'Algol 60 l'idea del sottoprogramma, migliorandone la sintassi e la dichiarazione dei parametri, allo scopo di consentirne un'implementazione più efficiente sui calcolatori di oggi e di garantirne una più elevata sicurezza d'uso.
Questo capitolo è diviso in quattro parti. Nella prima viene esaminata la sintassi della dichiarazione di un sottoprogramma e le conseguenze più importanti sui nomi con validità locale alle procedure. Nella seconda vengono trattati i parametri, la terza introduce e sviluppa i meccanismi di chiamata a un sottoprogramma, mentre l'ultima presenta i sottoprogrammi predefiniti in Pascal.
Il nome «sottoprogramma» è solo un termine generico e non compare in Pascal, che fa una distinzione fra funzioni (che restituiscono sempre un valore che va a sostituirsi alla funzione nell'espressione in cui viene chiamata) e procedure, (che, normalmente, comunicano con il loro ambiente esterno solo attraverso i parametri). Tuttavia, procedure e funzioni hanno molte proprietà in comune ed il termine generico di sottoprogramma verrà usato tutte le volte che ci si riferirà a concetti applicabili sia alle procedure che alle funzioni.

4.1 DICHIARAZIONE DI PROCEDURE E FUNZIONI

4.1.1 SINTASSI

La dichiarazione di un sottoprogramma è costituita da due parti legate fra di loro: l'intestazione del sottoprogramma, che ne specifica l'interfaccia con l'ambiente esterno, ed il corpo del sottoprogramma, che ne descrive il funzionamento interno. Queste due parti possono essere fisicamente separate, nel qual caso l'intestazione è seguita da una direttiva, che sostituisce il corpo del sottoprogramma; questo compare successivamente da qualche altra parte nel programma, preceduto da un identificatore di sottoprogramma, che ne specifica il nome e ne completa così la dichiarazione. Nel caso in cui intestazione e corpo compaiano insieme, direttiva ed identificatore possono essere omessi. Anche se questa è la situazione più frequente nei programmi di tutti i giorni, è da considerare estremamente auspicabile una netta separazione fra interfaccia con il mondo esterno e descrizione interna.
L'unica direttiva standard, presente in Pascal, è *forward*, che serve esattamente allo scopo di cui si è appena parlato, e cioè specificare l'intestazione del sottoprogramma, dichiarandola in anticipo, in modo che la sua interfaccia possa es-

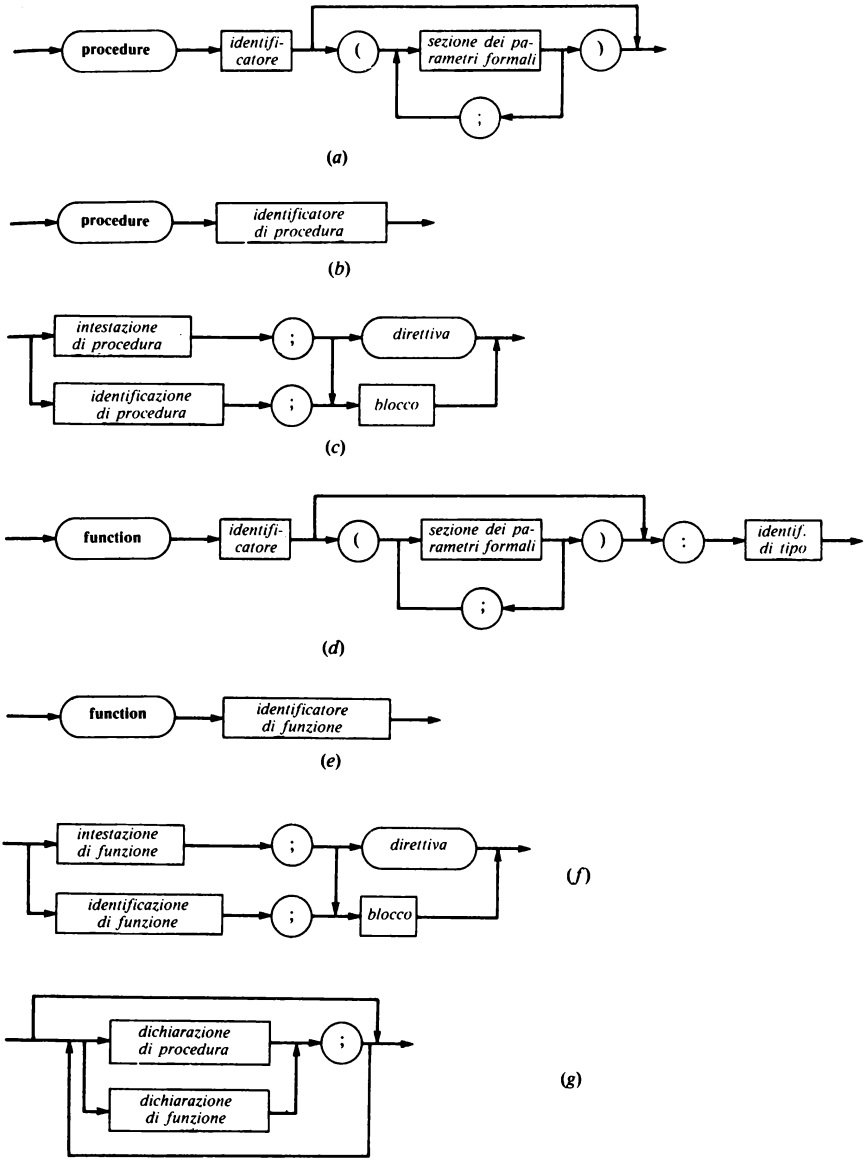


Figura 4.1 Sintassi della dichiarazione di procedure e funzioni: (*a*) intestazione di procedura; (*b*) identificazione di procedura; (*c*) dichiarazione di procedura; (*d*) intestazione di funzione; (*e*) identificazione di funzione; (*f*) dichiarazione di funzione; (*g*) sezione di dichiarazione di procedure e funzioni

sere nota subito, mentre il suo corpo potrà comparire in seguito. Diverse versioni del Pascal introducono nuove direttive, non standard, che vengono usate quando il corpo del sottoprogramma non fa parte del programma che si sta compilando. Per esempio la direttiva *external* serve per specificare che il corpo del sottoprogramma verrà compilato separatamente e che dovrà essere incluso nel modulo eseguibile prima dell'esecuzione.
L'intestazione del sottoprogramma consente così ad un particolare editor di collegamento (un programma che collega moduli compilati e che prepara il programma nella versione eseguibile) di verificare che il sottoprogramma compilato separatamente abbia la stessa specifica di interfaccia. In alcuni casi, può comparire una direttiva *Fortran*, che specifica che il corpo corrispondente è stato scritto in Fortran e che probabilmente fa uso di convenzioni di collegamento diverse da quelle del Pascal. Dal momento che, nel Pascal standard, non è prevista la possibilità di compilazioni separate, non si discuterà più a lungo questo argomento, rimandando il lettore alla documentazione tecnica fornita insieme ai compilatori.
Nel paragrafo 4.2 verranno trattati i parametri formali. La dichiarazione di una procedura e di una funzione sono molto simili (si veda la Figura 4.1); le uniche differenze sono: la parola chiave introduttiva, la dichiarazione del tipo della funzione e la necessità di assegnare un valore a questa durante la sua esecuzione. Contrariamente a Fortran e PL/1 che prevedono, per l'assegnamento di un valore alla funzione, un'istruzione speciale (RETURN), il Pascal usa l'istruzione di assegnamento ordinaria, specificando — nella parte sinistra — il nome della funzione. Il significato speciale attribuito al nome della funzione ha validità solamente all'interno del corpo della funzione stessa.
Si possono definire sottoprogrammi senza parametri; in questo caso la lista dei parametri è vuota e mancano anche le parentesi previste dalla sintassi.

4.1.2 STRUTTURA DEL CORPO DI UN SOTTOPROGRAMMA

Il corpo di un sottoprogramma è un blocco (definito nel paragrafo 1.3), ed è costituito da più parti dichiarative o di definizione, disposte in un ordine rigidamente fissato e seguite da un'istruzione composta che ne costituisce la parte eseguibile. Dal momento che i sottoprogrammi sono racchiusi nel corpo del programma e poiché possono contenere a loro volta altre dichiarazioni di procedure, parecchi blocchi possono essere racchiusi in una struttura a livelli, nota, sin dai tempi dall'Algol 60, come struttura a blocchi. Il Pascal è infatti un linguaggio con una struttura a blocchi, analogamente al PL/1 e diversamente dal Fortran e dal Cobol. In un sottoprogramma Fortran, per esempio, tutti i nomi che appartengono al sottoprogramma sono visibili ma, fra i nomi esterni, sono visibili solo quelli che si riferiscono ad altri sottoprogrammi o a dati che siano stati dichiarati in blocchi comuni. In PL/1 e in Pascal, invece, i nomi appartenenti a blocchi esterni — che contengono cioè il blocco in esame — sono visibi-

li, sempre che non siano omonimi di nomi dichiarati localmente al sottoprogramma.
Una descrizione più approfondita di questi concetti richiede alcune definizioni:

1. Un nome è un identificatore o una label; gli identificatori possono essere usati per le costanti, i tipi, le variabili, le procedure, le funzioni, i campi di record e per i parametri.

2. Ciascun nome, in un programma Pascal, ha un punto in cui viene definito, diverso per ciascuna categoria. Label, costanti, tipi, variabili, procedure e funzioni sono definite rispettivamente nelle sezioni di dichiarazione di label, definizione di costanti e così via. Le costanti che indicano i valori di un tipo scalare sono definite nel corrispondente identificatore di tipo. I campi di un record sono definiti nell'identificatore di tipo del record di cui fanno parte. I parametri vengono definiti nelle corrispondenti liste parametri. Gli identificatori predefiniti sono definiti (si suppone) nell'ambiente del programma.

3. Per omogeneità dei termini, un nome viene *definito* nel suo punto di definizione e *usato* in ogni altro punto del programma: tali punti vengono chiamati *punti di utilizzo* (*applied occurrence*).

4. La parte del programma dove un nome viene definito è detta *regione*; la parte in cui esso è visibile (cioè dove può essere usato) è detta *campo di influenza* (*scope*) o *ambito*.

In base a queste definizioni, alcune semplici regole spiegano il meccanismo di funzionamento di una struttura a blocchi:

1. Ciascun nome usato nel programma deve essere definito e ciascun uso di un nome deve avvenire all'interno dell'ambito della definizione. Non esiste alcun modo di definire o di dichiarare implicitamente un nome.

2. Un nome deve avere uno ed un solo punto di definizione nel suo campo di influenza; non può cioè essere definito due volte, all'interno dello stesso ambiente, con significati diversi (per esempio prima come una costante e quindi come una funzione) e neppure con lo stesso significato.

3. A meno di due sole eccezioni, il punto di definizione di un nome deve precedere sempre, nel programma, tutti i punti di utilizzo. Questo permette al compilatore di gestire tutto il processo di identificazione in una sola passata del programma. Le eccezioni riguardano le variabili usate come parametri del programma, che si trovano nella sua intestazione e quindi prima che siano state definite (si veda il paragrafo 1.3) ed i tipi puntatore (si veda il paragrafo 13.2).

4. La regione di un nome viene definita, per ciascuna categoria, nel modo seguente: per label, costanti, tipi, variabili, procedure e funzioni è il blocco in cui si trova la relativa dichiarazione; per le costanti che definiscono i valori di un tipo scalare è il blocco più esterno contenente l'identificatore di tipo;

per i parametri di sottoprogrammi è la lista dei parametri formali in cui sono specificati e il corpo della corrispondente procedura; per gli identificatori di campi è l'identificatore del tipo di record in cui sono specificati (gli identificatori di record sono visibili anche in due altri contesti molto ristretti: si veda il paragrafo 11.2); infine, per gli identificatori predefiniti, è una regione fittizia che comprende tutto il programma.

5. Il campo di influenza di un nome è costituito dalla sua regione, con l'eccezione di tutte quelle regioni in cui è definito un nome omonimo.

6. Una variabile esiste per tutto il tempo in cui è attivato il blocco che ne rappresenta la regione. Non esiste, invece, quando questo blocco non è attivato. Questo comportamento è analogo a quello delle variabili AUTOMATIC del PL/1 e diverso sia da quello delle variabili STATIC dello stesso PL/1 che dalla maggior parte delle implementazioni Fortran. La conseguenza più importante per il programmatore è che tutte le variabili locali ad un sottoprogramma, all'attivazione del sottoprogramma stesso hanno un valore indefinito. Un'altra importante conseguenza è che lo spazio di memoria richiesto dalle variabili locali può essere rilasciato al termine dell'attivazione del blocco.

Occorre mettere in rilievo alcune conseguenze di queste regole nel caso in cui un blocco A ne includa un secondo B (B può essere il blocco relativo ad un sottoprogramma la cui dichiarazione compare in A, o può essere racchiuso, ad un qualsiasi livello, in un blocco di questo tipo):

1 Un nome definito in A è visibile in B se e solo se non esistono variabili omonime definite in B; tale nome è *globale* in B.

2 Un nome definito in B non è mai visibile in A; esso è *locale* a B.

3 Se in B si vuol fare uso di un nome definito in A, non si deve definirne un omonimo in B; in caso contrario, lo stesso nome avrebbe due significati diversi nello stesso contesto, il che è sbagliato. In Pascal la situazione è più complicata che non in linguaggi come il PL/1, dal momento che un nome può denotare non solo una variabile o un sottoprogramma, ma anche una costante o un tipo.

ESEMPIO 4.1: CAMPO DI INFLUENZA E REGIONI

```
program P(infile, outfile);
   const a = 5;
   type b = (c, d, e);
   var f: b; g: integer; infile, outfile: file of real;
   procedure P1;
      const h = c;
      var g: b; i: integer;
```

```
    begin ... end {P1};
    procedure P2;
        type a = (h, j, k);
        procedure P3;
            var b, j: a;
        begin ... end {P3};
    begin ... end {P2};
begin ... end {programma P}.
```

La struttura statica di questo esempio può essere rappresentata dallo schema di Figura 4.2: le frecce delimitano la regione di ciascun nome, mentre il campo di influenza del nome stesso è evidenziato a tratto continuo.

COMMENTI

1. *infile* e *outfile* sono utilizzati nell'intestazione del programma prima di essere stati definiti. Devono essere definiti nel blocco del programma e non in un blocco in esso contenuto.
2. *integer* e *real* sono definiti in un ambiente fittizio, dal momento che sono tipi predefiniti. Potrebbero essere ridefiniti da un nome omonimo dichiarato all'interno del programma, benché questa sia una pessima idea.
3. L'intero *g*, dichiarato in *P*, è nascosto in *P1* dalla variabile con lo stesso nome di tipo *b*. Il tipo *b* è visibile in *P1*, dal momento che non è stato ridefinito.
4. In *P1* la costante *c*, di tipo *b*, è usata nella definizione della costante *h*. Di

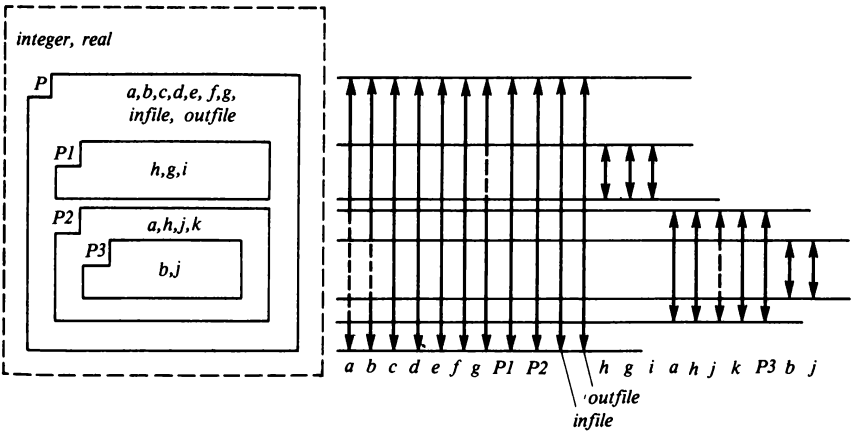


Figura 4.2 Struttura statica dell'Esempio 4.1

conseguenza non può essere ridefinita nello stesso blocco, né come costante, né come qualsiasi altra cosa.

5. La costante *h*, nella procedura *P1*, non ha alcuna relazione con l'omonima costante nella procedura *P2*, dal momento che compaiono in blocchi disgiunti.
6. La variabile *j* nella procedura *P3* nasconde la costante *j* della procedura *P2*. Di conseguenza, il tipo *a* nella procedura *P2* e le sue costanti scalari sono parzialmente nascosti nella procedura *P3*. Sarebbe possibile usare *a*, *h* e *k* in *P3*, ma ciò è fortemente sconsigliato, perché uno dei valori di questo tipo non sarebbe visibile e sarebbe accessibile solo mediante espressioni del tipo *succ*(*h*) o *pred*(*k*).

ESEMPIO 4.2: ARCO DI VITA DELLE VARIABILI

```
program P;
   var a: ...;
   procedure P1;
      var a1: ...;
   begin ... end {P1};
   procedure P2;
      var a2: ...;
      procedure P3;
         var a3: ...;
      begin ... P1 ... end {P3};
   begin ... P3 ... end {P2};
begin ... P1; ... P2 ... end {programma P}.
```

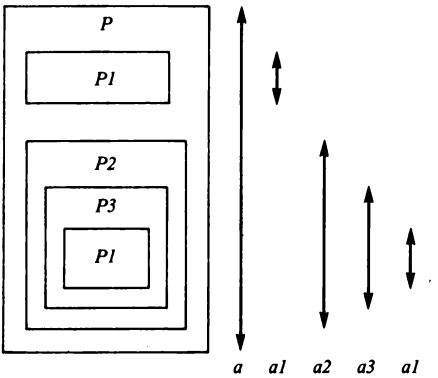


Figura 4.3 Struttura dinamica dell'Esempio 4.2

La struttura dinamica di questo esempio è rappresentata nello schema di Figura 4.3, in cui ciascun riquadro rappresenta un'attivazione del blocco di un sottoprogramma e le frecce delimitano l'arco di vita di ciascuna variabile.

COMMENTI

Si noti che *a1* ha due esistenze distinte che non hanno niente in comune. Al termine della prima attivazione di *P1* il valore corrente di *a1* viene perso. Se si vuole salvare un valore intermedio di *a1* fra due chiamate successive di *P1*, si deve usare una variabile globale come *a*.

Le regole di esistenza e visibilità dei nomi verranno ulteriormente illustrate con molti esempi in altre parti di questo libro. Queste regole sono state definite in modo tale che un sottoprogramma possa essere inserito in un programma indipendentemente dai nomi che usa al suo interno. Con l'uso di nomi globali, al contrario, più sottoprogrammi disgiunti possono fare riferimento agli stessi nomi. Tuttavia, dal momento che queste regole sono un po' complicate, occorre procedere con cautela. Quando si usano nomi globali si può andare incontro a due tipi di pericoli, che è meglio sottolineare:

a) se ci si dimentica di definire un nome locale in un sottoprogramma e se sfortunatamente esiste un nome uguale e dello stesso tipo in un sottoprogramma più esterno, l'errore può passare inosservato e provocare strani comportamenti;
b) in un sottoprogramma facente parte di una serie di sottoprogrammi annidati l'uno nell'altro, un'omonimia può nascondere un nome globale in un sottoprogramma che si trovi ad un livello più profondo. Anche in questo caso, in qualche rara circostanza l'errore può passare inosservato ed avere strane conseguenze.

Come regola generale è consigliabile definire sempre i nomi al più basso livello di annidamento possibile. Così si possono evitare, per quanto possibile, i due pericoli descritti prima e, nel caso di variabili, si ottimizza l'uso della memoria, poiché lo spazio per la memorizzazione dei loro valori è utilizzato solo durante la loro esistenza, cioè per il solo tempo di attivazione del sottoprogramma in cui sono dichiarate. Lo svantaggio di questa possibile ottimizzazione è che occorre salvare il valore di una variabile locale, fra due attivazioni successive di un sottoprogramma, in una variabile globale ad esso. Una volta ancora è importante sottolineare il fatto che il comportamento delle variabili in Pascal — e specialmente il loro arco di vita — è completamente diverso da quello della maggior parte delle implementazioni Fortran. Si noti, tuttavia, che lo standard Fortran non specifica che il valore di una variabile debba essere salvato fra due chiamate successive del sottoprogramma cui appartiene.

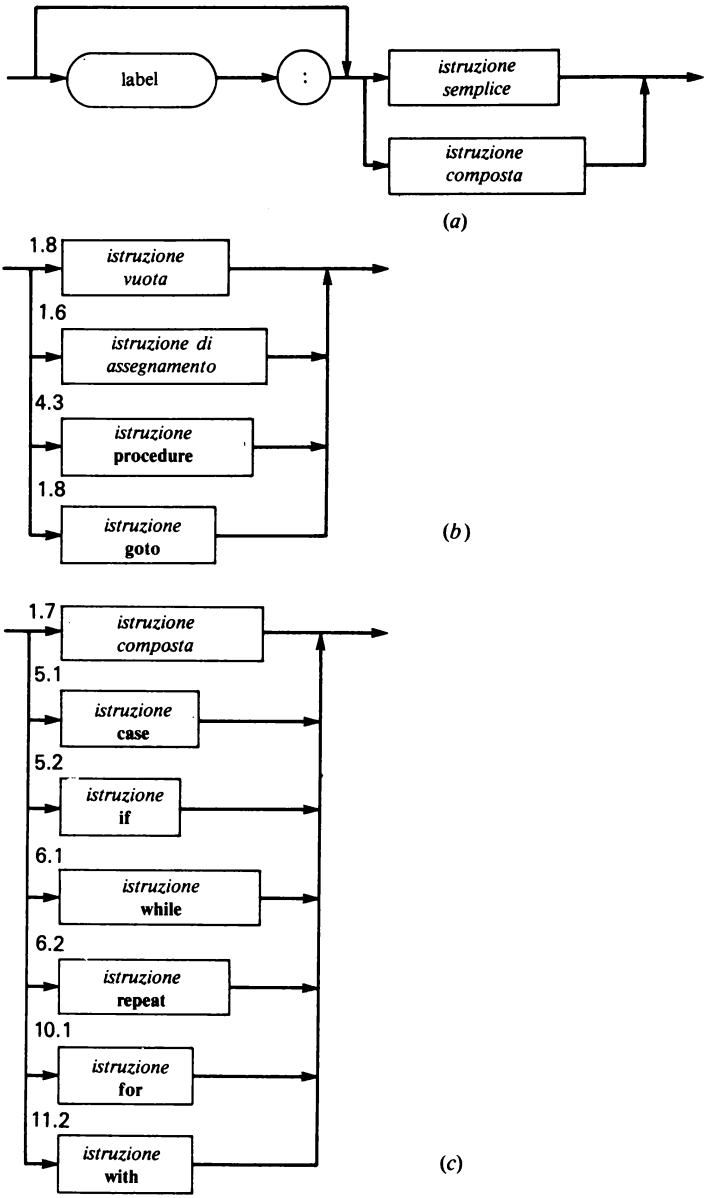


Figura 4.4 Diagramma sintattico di un'istruzione: (*a*) istruzione; (*b*) istruzione semplice; (*c*) istruzione composta. I numeri rinviano ai paragrafi del libro dove sono descritte le rispettive istruzioni

4.1.3 LABEL E ISTRUZIONI

Le label sono entità molto strane, in Pascal: il loro nome non è un identificatore, ma un piccolo intero positivo; non sono né costanti, né variabili, devono essere dichiarate ed il loro solo uso possibile è all'interno di istruzioni **goto** (si veda il paragrafo 1.8). Per di più, una label può indirizzare una sola istruzione nel corpo del blocco in cui è dichiarata. La dichiarazione obbligatoria di tutte le label consente una compilazione in una sola passata, dal momento che il livello del blocco contenente l'istruzione indirizzata è noto anche per blocchi annidati.
Si può dare ora la definizione completa di un'istruzione, come illustrato in Figura 4.4.

4.2 PARAMETRI FORMALI

I parametri formali, che sono il mezzo normale di comunicazione fra un sottoprogramma ed il suo ambiente esterno, sono completamente specificati nell'intestazione del sottoprogramma, al contrario di quanto accade per molti altri linguaggi di programmazione. In Fortran e in PL/1, l'intestazione elenca semplicemente i nomi dei parametri, la cui descrizione viene fatta successivamente nel corpo del sottoprogramma, in modo identico a quanto avviene per le variabili ordinarie. In Pascal tutte le informazioni necessarie sono concentrate in un unico punto, che costituisce l'interfaccia fra il sottoprogramma ed il suo ambiente esterno. Un'ulteriore evidenza del ruolo di intermediario, svolto dalle descrizioni dei parametri formali, è data dal fatto che le loro dichiarazioni devono essere visibili all'esterno, mentre i loro nomi sono visibili solo all'interno. La dichiarazione di un parametro formale ne descrive le caratteristiche — cioè il tipo se è un oggetto, o la sua intestazione se è un sottoprogramma. Inoltre

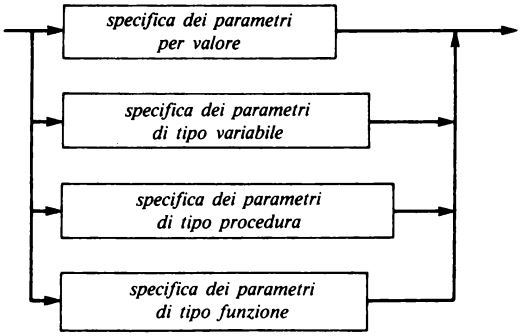


Figura 4.5 Diagramma sintattico della sezione dei parametri formali

descrive il modo di trasmissione, che può essere uno dei cinque possibili esistenti in Pascal. Questa informazione serve per sapere come ogni parametro attuale deve essere trattato e indirizzato all'interno del sottoprogramma. In questo paragrafo vengono descritti solo i primi quattro modi possibili (si veda la Figura 4.5): l'ultimo sarà presentato nel paragrafo 9.7.
Verranno dapprima esaminati i parametri nell'intestazione dei sottoprogrammi; successivamente verranno esaminati la chiamata a sottoprogrammi e i parametri attuali.

4.2.1 PARAMETRI PASSATI PER VALORE

I parametri passati per valore non hanno un equivalente in molti altri linguaggi di programmazione, e tuttavia costituiscono il modo di trasmissione più semplice e più sicuro. All'interno del sottoprogramma un parametro passato per valore è semplicemente una variabile locale — del tipo specificato — cui viene assegnato il valore del parametro attuale corrispondente, prima dell'esecuzione delle istruzioni del sottoprogramma stesso (si veda la Figura 4.6). Non esiste alcun altro legame con l'ambiente esterno durante l'esecuzione del sottoprogramma: in questo modo tale ambiente non può essere modificato da parametri passati per valore. Di conseguenza, questo metodo di trasmissione è adatto per i parametri di ingresso, cioè per i dati del sottoprogramma. Occorre osservare che il tipo dei parametri passati per valore deve avere un nome e, di conseguenza, deve essere definito in un blocco che contenga quello in esame.

ESEMPIO 4.3: PARAMETRI PASSATI PER VALORE

```
function Fahrenheit (temperatura: gradi): integer
   {converte un dato valore di temperatura da gradi Celsius a gradi Fahrenheit};
begin {Fahrenheit}
   Fahrenheit := (9 * temperatura) div 5+32
end {Fahrenheit}
```

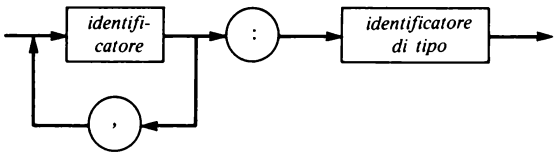


Figura 4.6 Diagramma sintattico della specifica dei parametri passati per valore

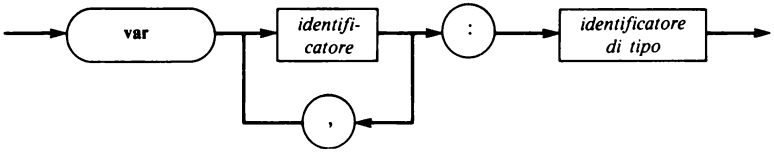


Figura 4.7 Diagramma sintattico della specifica dei parametri di tipo variabile

COMMENTI

Il tipo *gradi*, usato nell'intestazione della funzione, deve essere definito da qualche parte nell'ambiente della funzione, per esempio nel blocco immediatamente esterno a quello in esame, o in un qualsiasi altro blocco che lo contenga. Si fa l'ipotesi che questo tipo sia un sottocampo del tipo *integer*. Per esempio, nel caso di un termometro per la misura della temperatura atmosferica, potrebbe essere $[-10..+40]$.

4.2.2 PARAMETRI DI TIPO VARIABILE

La dichiarazione dei parametri di tipo variabile è identica a quella dei parametri passati per valore, a meno della parola chiave **var** che apre la dichiarazione (si veda la Figura 4.7). I parametri di tipo variabile sono all'incirca l'equivalente del modo standard di passaggio dei parametri del Fortran o del PL/1, con l'eccezione che i corrispondenti parametri attuali non possono essere un'espressione o una costante. All'interno di un sottoprogramma, il parametro formale si comporta come una qualsiasi variabile locale, ma di fatto è il corrispondente parametro attuale. In altri linguaggi di programmazione questo metodo viene chiamato *passaggio per indirizzo* (*call by reference*). Questo tipo di parametri consente ad un sottoprogramma di modificare il suo ambiente esterno e di conseguenza è adatto per i parametri di uscita, cioè per i risultati calcolati dal sottoprogramma. Tuttavia, questo metodo di passaggio dei parametri non è l'esatto simmetrico di quello precedente, e si vedrà nel paragrafo 9.6 che può essere pericoloso se usato male.

ESEMPIO 4.4: PARAMETRI DI TIPO VARIABILE

```
procedure Sfera (raggio: real: var superficie, volume: real)
   {calcola superficie e volume di una sfera di dato raggio};
   const pi = 3.1415926536;
   var temp: real {variabile ausiliaria};
begin {Sfera}
   temp := pi * sqr(raggio);
```

```
    superficie := 4 * temp {4 π R²};
    volume := 4/3 * raggio * temp {4/3 π R³}
end {Sfera}
```

4.2.3 PARAMETRI DI TIPO PROCEDURA E FUNZIONE

Nella descrizione che segue i parametri di tipo procedura e di tipo funzione saranno chiamati *sottoprogrammi parametrici*, cioè sottoprogrammi che sono parametri di altri sottoprogrammi (si veda la Figura 4.8). Sia Fortran che PL/1 offrono questa possibilità, ma nessuno di questi due linguaggi lo fa in modo generale e sicuro; non esiste cioè alcuna dichiarazione dei sottoprogrammi parametrici che consenta al compilatore di verificarne la correttezza d'uso. Una situazione analoga si era verificata nella prima definizione del Pascal e la sintassi attuale è stata introdotta solo in seguito al processo di standardizzazione effettuato dall'ISO. Di conseguenza esistono molti compilatori del Pascal che accettano ancora solo la sintassi originale, nella quale i parametri accettati da sottoprogrammi parametrici non possono essere specificati.
La dichiarazione di un sottoprogramma parametrico ha esattamente la stessa forma di un sottoprogramma ordinario: specifica il numero, modalità di trasmissione e tipo o caratteristiche dei parametri e ne assegna anche un nome. Tuttavia questo nome non ha alcun significato al di fuori della dichiarazione del sottoprogramma: lo si può considerare alla stregua di un commento obbligatorio. Ciò è stato fatto per evitare di introdurre nel linguaggio un altro costrutto sintattico che lo avrebbe reso più complicato.
I sottoprogrammi parametrici sono essi stessi un concetto intricato e vengono usati raramente. Tuttavia, quando servono, non esiste alcun altro strumento equivalente e di semplice uso che li sostituisca, e questa è la ragione principale per cui sono stati introdotti nel linguaggio. Negli esempi successivi sono illustrate due situazioni in cui un sottoprogramma ha, come parametro, un altro sottoprogramma. Non essendo ancora state descritte un numero sufficiente di caratteristiche del Pascal, il corpo dei sottoprogrammi è stato sostituito con dei commenti.

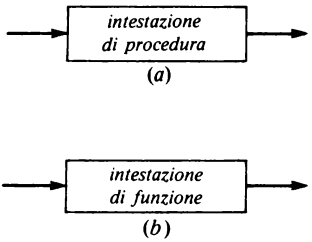


Figura 4.8 Diagramma sintattico per sottoprogrammi parametrici: (*a*) specifica dei parametri di tipo procedura; (*b*) di tipo funzione

ESEMPIO 4.5: PARAMETRI DI TIPO FUNZIONE

```
{il primo esempio riguarda una funzione che calcola un valore approssimato dell'integrale
      di una funzione in un dato intervallo; la funzione da integrare è passata come parametro di tipo funzione}
function Integrale (function f(x: real): real; da, a: real): real
      {f(x) deve essere definita nell'intervallo da ≤ x ≤ a; la funzione Integrale calcola il valore approssimato di ∫ da a f(x) dx, usando un opportuno metodo numerico};
   const epsilon = ... {un'opportuna precisione assoluta};
   var risultato: real {per calcolare l'integrale voluto};
      {vengono poi dichiarate tutte le variabili necessarie}
begin {Integrale}
   {per esempi di metodi possibili si veda la relativa letteratura; un valore della funzione, per un dato argomento, si calcola come valore := f(argomento)}
   Integrale := risultato
end {Integrale};
{la funzione seguente è un possibile parametro attuale per la funzione Integrale}
function Funz (x: real): real;
begin {Funz}
   Funz := 1 / sqrt(sqr(m * cos(n * sin(x))))
end {Funz};
{una possibile chiamata a Integrale sarebbe: writeln(Integrale(Funz, 0, pi/2))}
```

COMMENTI

1. Nella definizione originale del Pascal, *f* sarebbe comparsa senza alcun parametro nell'intestazione della funzione *Integrale*. Nella nuova sintassi si vede che il parametro di tipo funzione deve avere a sua volta un argomento reale, trasmesso per valore. Inoltre il compilatore può effettuare i controlli necessari.
2. Senza sottoprogrammi parametrici non ci sarebbe stata una soluzione semplice e soddisfacente al problema, non si sarebbe potuto così costruire una funzione in grado di integrare una qualsiasi altra funzione con argomento reale.
3. Il nome *x*, che compare nell'intestazione di *f*, inclusa nell'intestazione di *Integrale*, non ha alcuna relazione con alcun altro nome. Lo si sarebbe potuto chiamare *y* o *argomentoreale* e nulla di significativo sarebbe cambiato nell'esempio.
4. I sottoprogrammi parametrici hanno complicate ripercussioni sulla struttura a blocchi. Quando *Integrale* viene chiamato, con un parametro *Funz*, questi due nomi devono essere visibili, ma non necessariamente nella stessa regione e nello stesso campo di influenza. Per esempio, *Integrale* può essere allo stesso livello della procedura che la chiama, mentre *Funz* può essere locale a

questa procedura. Ciò significa che, quando *Integrale* chiama *Funz*, per mezzo del suo parametro formale *f*, *Funz* può non essere visibile. Quando il controllo passa da *Integrale* a *Funz*, si verifica un cambiamento completo nel contesto di visibilità e lo stesso avviene quando il controllo ritorna ad *Integrale*. I due nomi globali *m* e *n* usati in *Funz* (che possono essere costanti o variabili, probabilmente di tipo *real*), devono essere visibili da *Funz*, ma non è assolutamente necessario che siano visibili nel corpo di *Integrale* o nel sottoprogramma che lo chiama.

ESEMPIO 4.6: PARAMETRI DI TIPO PROCEDURA

Questo secondo esempio è relativo ad una procedura che deve effettuare una determinata azione su tutti i campi di una data struttura, passata per indirizzo e i cui componenti sono di tipo *T*. Il processo è passato come parametro di tipo procedura.

```
procedure ApplicaUnProcesso
   (procedure Processa(var elemento: T);
   var oggetto: tipostrutturato);
   {eventuali definizioni e dichiarazioni locali a ApplicaUnProcesso}
begin {ApplicaUnProcesso}
   {ogni volta che si isola un elemento c dell'oggetto strutturato, lo si processa tra-
         mite l'istruzione Processa(c)}
   {la procedura termina dopo aver processato tutti gli elementi}
end {ApplicaUnProcesso}
```

4.3 CHIAMATE A SOTTOPROGRAMMI E PARAMETRI ATTUALI

Una funzione è chiamata usando un identificatore di funzione, già esaminato nel paragrafo 3.3. Una procedura è chiamata usando un'istruzione di attivazione di procedura, che è l'ultima istruzione semplice che si deve ancora descrivere. La sintassi (Figura 4.9) è esattamente la stessa di quella di un identificatore di funzione. Si noti, in particolare, che nel Pascal non c'è alcuna parola chiave per questa istruzione, al contrario di quanto avviene in Fortran o PL/1. Ogni occorrenza di un identificatore di procedura all'inizio di un'istruzione è una chiamata di procedura.

Una chiamata di procedura causa l'attivazione della parte di istruzione della procedura chiamata, dopo avere associato ad ogni parametro formale il corrispondente parametro attuale. La natura di questa associazione dipende dal metodo di trasmissione dei parametri e la corrispondenza è stabilita dalle rispettive posizioni dei parametri attuali e formali nella chiamata della procedura e

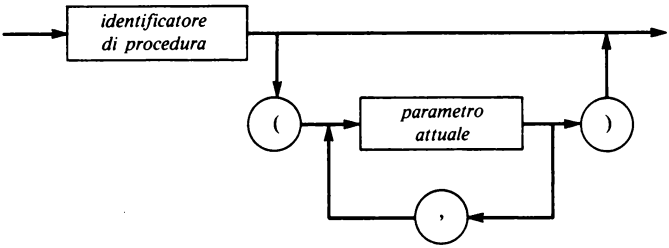


Figura 4.9 Diagramma sintattico di una chiamata di procedura

nell'intestazione della stessa. Come si è visto nel paragrafo 3.3, questa corrispondenza posizionale è la regola in quasi tutti i linguaggi di programmazione e, in Pascal, non c'è alcun modo di omettere dei parametri attuali, o fornirne più di quanti specificati.
Il resto di questo paragrafo spiega le regole di compatibilità di tipo e la corrispondenza fra parametri attuali e parametri formali, in funzione della modalità di trasmissione prescelta.

4.3.1 PARAMETRI PASSATI PER VALORE

La trasmissione di un parametro per valore è esattamente equivalente all'assegnamento rappresentato da

parametro formale := parametro attuale

Ciò significa che il parametro attuale può essere qualunque espressione (incluso una variabile o una costante) purché il suo tipo sia compatibile, nell'assegnamento, con il tipo di parametro formale specificato nell'intestazione del sottoprogramma (si veda il paragrafo 1.6). Per esempio, se il parametro formale è di tipo *real*, il corrispondente parametro attuale può essere sia un'espressione reale che un'espressione intera. Al contrario, un'espressione reale sarebbe un parametro attuale illegale per un parametro formale di tipo *integer*. Se il parametro formale appartiene ad un sottocampo di un dato tipo e il corrispondente parametro attuale non è dentro al campo specificato, si verificherà un errore, generalmente a tempo di esecuzione. La trasmissione per valore può essere usata anche per i parametri strutturati — dato che l'assegnamento è permesso per la maggior parte dei tipi strutturati — ma non si deve trascurare il costo della copia che viene fatta implicitamente.

4.3.2 PARAMETRI DI TIPO VARIABILE

Poiché un parametro formale si comporta come una variabile locale ma in realtà rappresenta il corrispondente parametro attuale, non si può trovare, nel linguaggio, alcun semplice equivalente di questo meccanismo. La migliore spiegazione, probabilmente, fa uso del termine di indirizzo, o riferimento alla memoria e questo concetto di basso livello è, di per sé, un'indicazione della potenza, ma anche della pericolosità, tipica di tutti i meccanismi di basso livello.
Il nome di una variabile semplice è un riferimento a quella zona di memoria che ne contiene i valori successivi. Mentre nella trasmissione per valore viene passato soltanto il valore corrente che è memorizzato in un'altra variabile (il parametro formale), nella trasmissione per indirizzo viene passato proprio questo indirizzo. Come conseguenza, quando il sottoprogramma esegue un'istruzione in cui compare un parametro formale, esso fa riferimento alla zona di memoria associata alla variabile che è il parametro attuale corrispondente.
Poiché non c'è alcuna restrizione sull'uso dei parametri formali, i parametri attuali corrispondenti devono fare riferimento a un oggetto significativo; deve quindi essere ciò che viene chiamato un *accesso di tipo variabile*. Ciò comprende sia nomi di variabili che riferimenti a campi di variabili strutturate, ma esclude costanti o espressioni che non sono indirizzabili da variabili.
Inoltre, poiché non si può invocare alcuna conversione implicita di tipo nel meccanismo di trasmissione per indirizzo, il parametro attuale deve essere dello stesso tipo di quello formale. Un reale richiede un reale, un intero un intero, ed un sottocampo di un dato tipo vuole lo stesso sottocampo di quel tipo e non un sottocampo generico dello stesso tipo, né tantomeno un sottocampo che abbia solo gli stessi valori limite. Se parecchi parametri formali sono elencati in una lista nell'intestazione di una procedura, i corrispondenti parametri attuali devono essere tutti dello stesso tipo.
I parametri di tipo variabile possono essere pericolosi, specialmente quando sono di tipo strutturato, argomento che verrà discusso nel paragrafo 9.6 e nei successivi. Tuttavia può essere presente un altro pericolo se nel sottoprogramma si fa uso di variabili globali, di variabili cioè che sono direttamente visibili nel corpo del sottoprogramma. Se una di queste variabili globali viene anche passata al sottoprogramma come parametro attuale di tipo variabile, essa diventa accessibile con due nomi differenti. Quando il sottoprogramma modifica una di queste due variabili, modifica automaticamente anche l'altra, dal momento che sono la stessa cosa. Questa situazione è da considerare molto sfortunata, in quanto il funzionamento interno di un sottoprogramma dipende dal suo ambiente esterno e dal modo in cui viene chiamato, e non può più essere descritto semplicemente in termini di rapporto di cause ed effetti indipendenti dal contesto. La miglior soluzione a questo problema è di evitare, per quanto possibile, di modificare le variabili globali, e perfino di farne uso come parametri variabili. Bisognerebbe considerarle come parametri impliciti di tipo variabile nel sottoprogramma che le usa, e non bisognerebbe mai usare due volte la

stessa variabile come parametro di tipo variabile (implicito o meno) nella stessa chiamata a sottoprogramma.

4.3.3 PARAMETRI DI TIPO PROCEDURA E FUNZIONE

Il parametro attuale che corrisponde ad un sottoprogramma parametrico deve essere il nome di un sottoprogramma dello stesso tipo (procedura o funzione). Il parametro formale rappresenta il parametro attuale durante l'esecuzione del sottoprogramma di cui è un parametro, ma ciò non implica il concetto di riferimento per indirizzo, e non esistono quindi i problemi descritti nel caso di parametri di tipo variabile. Se il parametro formale è una funzione, il corrispondente parametro attuale deve essere il nome di una funzione dello stesso tipo.
I parametri attuali e formali devono avere una lista di parametri *congruente*. Benché nella maggior parte dei casi la lista dei sottoprogrammi parametrici sia molto semplice, il concetto di lista di parametri formali congruente è molto generale, come verrà descritto in seguito.
Due liste di parametri formali sono congruenti se hanno lo stesso numero di sezioni di parametri formali che si corrispondono nelle rispettive posizioni. Più semplicemente, due liste di parametri formali sono corrispondenti se sono identiche, ad eccezione degli identificatori dei parametri: devono cioè avere lo stesso numero di parametri, che devono essere dello stesso tipo ed avere lo stesso meccanismo di trasmissione. Nel caso di sezioni di parametri formali che specifichino sottoprogrammi parametrici, i parametri devono essere dello stesso tipo (procedure o funzioni), devono avere valori dello stesso tipo (se si tratta di funzioni) e devono avere liste di parametri formali congruenti. Naturalmente quest'ultima parte della regola viene usata raramente, dal momento che è molto difficile trovare un esempio non artificioso di sottoprogramma parametrico con un parametro che è a sua volta un sottoprogramma parametrico. Tuttavia, dopo avere introdotto i sottoprogrammi parametrici, sarebbe stato più complicato impedire una tale possibilità, anche se non verrà mai usata in programmi reali, che non renderla disponibile.
Il Pascal impone una restrizione: il parametro attuale corrispondente ad un sottoprogramma parametrico deve essere un nome definito nel programma e non può essere quindi un sottoprogramma predefinito. Questo perché, normalmente, i sottoprogrammi predefiniti non seguono le stesse regole dei sottoprogrammi ordinari, come sarà illustrato nel paragrafo successivo. Ad esempio, la funzione predefinita *sin* non può essere usata come parametro attuale della funzione *Integrale* dell'Esempio 4.5. Occorre invece definire un'altra funzione nel programma, il cui corpo sia costituito dalla sola chiamata a *sin* che assegna il risultato al nome della funzione.

4.4 PROCEDURE E FUNZIONI PREDEFINITE

In molti linguaggi, i sottoprogrammi predefiniti sono chiamati sottoprogrammi standard, mentre sono chiamati *sottoprogrammi richiesti* nello Standard ISO del Pascal. Come tutti gli altri identificatori predefiniti, essi hanno l'intero programma come campo d'influenza. Questo significa che hanno un nome globale riconosciuto nell'intero programma, a meno che non vengano nascosti dalla definizione di identificatori con lo stesso nome. Nell'Appendice D vengono elencati tutti gli identificatori predefiniti. Le funzioni predefinite di tipo aritmetico sono descritte nel paragrafo 3.4, mentre le funzioni che operano sui file sono descritte nei Capitoli 7 e 8. Le procedure predefinite sono invece descritte nei Capitoli 7 e 8 (procedure di input/output), 9 (pack/unpack) e 13 (procedure di allocazione dinamica). La lista che segue è stata introdotta solo per fornire un riferimento unico a tutti i sottoprogrammi predefiniti e per dare una prima idea intuitiva sulla loro utilità.

Funzioni predefinite

Aritmetiche

abs(x)	valore assoluto di *x*
sqr(x)	quadrato di *x*
sin(x)	seno di *x*
cos(x)	coseno di *x*
exp(x)	esponenziale (e^x)
ln(x)	logaritmo naturale di *x*
sqrt(x)	radice quadrata di *x*
arctan(x)	arcotangente di *x*

Conversione

trunc(x)	troncamento di *x*
round(x)	valore arrotondato di *x*

Ordinali

ord(x)	ordinale di *x*
chr(x)	carattere il cui numero ordinale è *x*
succ(x)	successore di *x*
pred(x)	predecessore di *x*

Booleane

odd(x)	*true* se *x* è dispari
eof(f)	end-of-file su *f*
eoln(f)	end-of-line su *f*

Procedure predefinite

Gestione di file

rewrite(f)	inizializza *f* in generazione
reset(f)	inizializza *f* in ispezione
put(f)	passa all'elemento successivo durante la generazione
get(f)	passa all'elemento successivo durante l'ispezione
read(f, v1, v2, ..., vn)	lettura di valori da *f*
write(f, v1, v2, ..., vn)	scrittura di valori in *f*
readln(f, v1, v2, ..., vn)	lettura e passaggio alla linea successiva
writeln(f, v1, v2, ..., vn)	scrittura e passaggio alla linea successiva
page(f)	inizio nuova pagina

Allocazione dinamica

new(p)	alloca una nuova variabile dinamica
new(p, c1, c2, ..., cn)	alloca una nuova variabile dinamica con varianti
dispose(p)	elimina una variabile dinamica
dispose(p, c1, c2, ..., cn)	elimina una variabile dinamica con varianti

Conversione

pack(a, i, z)	compatta *a* in *z* cominciando da *a*[*i*]
unpack(z, a, i)	scompatta *z* in *a* cominciando da *a*[*i*]

Queste 17 funzioni e 13 procedure sono gli unici sottoprogrammi standard predefiniti in Pascal, sono cioè quelli che si devono trovare in tutte le implementazioni del linguaggio. Naturalmente una particolare implementazione del Pascal può fornire altri identificatori e sottoprogrammi predefiniti.

5
Istruzioni condizionali

L'esecuzione di una procedura si svolge per mezzo dell'esecuzione di una serie di azioni individuali. Ciascuna azione corrisponde ad un'istruzione, e ogni istruzione può essere eseguita esattamente una volta, non essere eseguita affatto, essere eseguita più volte: non esistono altre possibilità. Per ognuno di questi casi, esiste una classe diversa di strutture di istruzioni: le *strutture sequenziali*, già esaminate nel Capitolo 1; le *strutture condizionali*, oggetto del presente capitolo; e le *strutture iterative* e *ripetitive*, che verranno trattate rispettivamente nei Capitoli 6 e 10.
Le istruzioni condizionali offrono un mezzo che consente di scegliere una fra più azioni possibili. La struttura più generale è data dall'istruzione di selezione **case**, il cui equivalente, in altri linguaggi, è complicato, non disciplinato e di uso molto limitato: i GOTO calcolati del Fortran e del BASIC, i GOTO ... DEPENDING ON del Cobol e i GOTO che hanno per operando una variabile usata come label in PL/1.
Nelle situazioni in cui si devono prendere in considerazione due sole alternative, l'istruzione **if** è un'utile semplificazione dell'istruzione **case**, ed è disponibile in Pascal in una forma simile a quella di altri linguaggi di programmazione.
Tuttavia è importante osservare che l'istruzione **case** è più generale dell'istruzione **if** e consente di avere, in molti casi, programmi più semplici e più chiari, come sarà dimostrato in seguito.

5.1 L'ISTRUZIONE case

Per poter fare una scelta occorre confrontare i diversi casi possibili secondo un dato criterio; questo criterio di scelta sarà, logicamente, il valore di un tipo ordinale e, molto spesso, di un tipo scalare. I diversi casi da prendere in considerazione sono associati ai diversi valori di questo tipo, rappresentati da costanti che individuano le istruzioni corrispondenti. Se una stessa istruzione deve essere

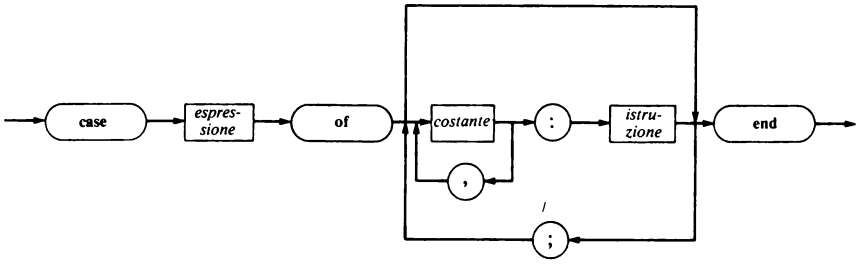


Figura 5.1 Diagramma sintattico di un'istruzione **case**

eseguita in corrispondenza di più valori, questi possono essere elencati prima dell'istruzione stessa (si veda l'Esempio 5.2). Se una particolare azione richiede più istruzioni, si può ricorrere ad un'istruzione composta, (si veda l'Esempio 5.3). Naturalmente l'ordine in cui compaiono i diversi rami della **case** non hanno alcuna importanza, dal momento che, ad un dato istante, uno solo potrà essere scelto. Analogamente in un'istruzione **case** uno stesso valore non potrà comparire due volte.

Sintassi

Il simbolo **end** è una specie di superparentesi che serve a chiudere l'istruzione **case** (si veda la Figura 5.1). Di conseguenza, si può usare una **case** in tutti i casi in cui è possibile usare un'istruzione semplice. È importante notare che, in questo caso, la parola chiave **end** viene impiegata senza il corrispondente **begin**: un altro esempio verrà fornito nel Capitolo 11.

Semantica

Data l'istruzione di selezione:

case *espressione* **of**
  $valore_1$: $istruzione_1$
  $valore_2$: $istruzione_2$
  ...
  $valore_n$: $istruzione_n$
end
se, per ogni i in $[1..n]$,
  $\{P \wedge (\text{espressione} = \text{valore}_i)\}$ $istruzione_i$ $\{Q\}$
allora $\{P\}$ istruzione **case** $\{Q\}$

Così, se *espressione* = $valore_i$, viene eseguita $istruzione_i$ e la **case** termina. Se non esiste alcun $valore_i$ che eguagli *espressione*, si ha un errore. Nella regola di verifica viene usato lo stesso Q come conseguente di tutte le istruzioni; di solito

ogni *istruzione*$_i$ ha un suo conseguente Q_i e Q è tale che ogni Q_i lo implichi. La situazione più frequente si ha quando Q è semplicemente l'unione di tutti i Q_i (come nell'Esempio 5.1).

ESEMPIO 5.1

```
type colore = (verde, giallo, rosso);
...
var semaforo: colore;
...
   case semaforo of
      rosso: writeln ('stop');
      verde: writeln ('avanti');
      giallo: writeln ('se possibile stop, altrimenti avanti')
   end
```

ESEMPIO 5.2

```
var esa: char {gli unici valori possibili sono valori esadecimali};
   decimale: 0..15 {il valore intero di una cifra esadecimale}
   ...
   case esa of
      'a', 'b', 'c', 'd', 'e', 'f':
         decimale := ord(esa) - ord('a') + 10;
      '0', '1', '2', '3', '4', '5', '6', '7', '8', '9':
         decimale =: ord(esa) - ord('0')
   end
```

ESEMPIO 5.3

```
var giorno: (dom, lun, mar, mer, gio, ven, sab);
   tempo: (nevoso, piovoso, nuvoloso, soleggiato);
   ...
   case giorno of
      lun, mar, mer, gio, ven:
         begin VaAlLavoro; Lavora; TornaACasa end;
      dom: {non fare niente};
      sab: case tempo of
         soleggiato: LavaLaMacchina;
         nuvoloso: FaUnaPasseggiata;
         nevoso, piovoso: AccendeLaTv
      end
   end
```

COMMENTI

1. Benché la sintassi non lo dica esplicitamente, è possibile mettere un punto e virgola, che non ha peraltro alcun significato, prima dell'**end**.
2. Una particolare istruzione può essere vuota se, in corrispondenza di un determinato valore, non si deve effettuare alcuna azione, come nel caso di *dom* (domenica), nell'Esempio 5.3. Omettere il valore corrispondente, nel campo label della **case**, non sarebbe stata la stessa cosa, dal momento che, qualora l'espressione avesse assunto quel valore, si sarebbe verificato un errore.
3. Sono state proposte due piccole estensioni all'istruzione **case**: consentire un intervallo di valori nel campo label (nell'Esempio 5.2 le due sequenze di label sarebbero diventate rispettivamente *'a'..'f'* e *'0'..'9'*); prevedere una sezione facoltativa, **otherwise**, per poter trattare anche tutti i valori non elencati esplicitamente come label della **case**. Queste due estensioni sono riconosciute da parecchi compilatori, ma della seconda esistono molte varianti diverse, e lo Standard ISO, al momento, non le prevede.
4. Qualche volta è veramente difficile trovare un incolonnamento adatto per strutture **case** molto complicate, come quella dell'Esempio 5.3. Se i campi label dell'istruzione **case** debbano essere allineati o no, se debbano essere scritti ciascuno su una linea, è veramente solo una questione di gusti.

Nei prossimi capitoli saranno proposti esempi di uso dell'istruzione **case** più complicati e significativi, soprattutto dal Capitolo 9 in poi.

5.2 L'ISTRUZIONE if

In molte circostanze la scelta da fare è ristretta a due sole alternative, rappresentate di solito dal valore di un'espressione booleana (un predicato). Per esempio:

```
case piove {tipo Boolean} of
   true: StaALetto;
   false: VaInSpiaggia
end
```

Benché questo costrutto sia perfettamente corretto e chiaro, è troppo diverso dal modo in cui lo si sarebbe espresso nel linguaggio parlato: lo si può sostituire con il costrutto equivalente:

```
if piove then StaALetto else VaInSpiaggia
```

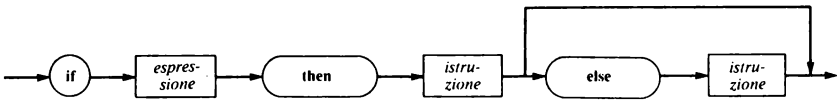


Figura 5.2 Diagramma sintattico di un'istruzione **if**

Sintassi

La sintassi dell'istruzione **if** è illustrata in Figura 5.2. Il ramo **else** è opzionale e può essere tralasciato se non si vuole svolgere alcuna azione quando l'espressione (booleana) è falsa. Ciascuna istruzione può essere sostituita da una sequenza di istruzioni racchiuse fra un **begin** e un **end** (istruzione composta); in questo modo le istruzioni **if** possono essere annidate.

```
if piove then
   if ore ≤ 10 then StaALetto
   else FaColazione
else if DistanzaDallaSpiaggia < 2 {km} then Cammina
   else PrendeUnTaxi
```

Nel caso di due istruzioni **if** di cui una soltanto abbia un ramo **else** si è di fronte ad una apparente ambiguità:

if *espressione*$_1$ **then if** *espressione*$_2$ **then** *istruzione*$_1$ **else** *istruzione*$_2$

In questo caso, la regola generale è che ciascun ramo **else** va accoppiato con il ramo **then** spaiato che gli è più vicino. Così, nell'esempio precedente, se *espressione*$_1$ vale *false*, non si deve fare niente, in quanto l'intera istruzione equivale a

```
if espressione1 then
   begin if espressione2 then istruzione1
      else istruzione2
   end
```

Semantica

Se $\{P \wedge B\}$ *istruzione*$_1$ $\{Q\}$
e $\{P \wedge \neg B\}$ *istruzione*$_2$ $\{Q\}$
allora $\{P\}$ **if** *B* **then** *istruzione*$_1$ **else** *istruzione*$_2$ $\{Q\}$

In modo analogo,

se $\{P \wedge B\}$ *istruzione*$_1$ $\{Q\}$
e $\{P \wedge \neg B\} \supset \{Q\}$
allora $\{P\}$ **if** *B* **then** *istruzione*$_1$ $\{Q\}$

Così se l'espressione boolena *B* vale *true* viene eseguita l'*istruzione*$_1$; se invece *B* è *false* viene eseguita — se esiste — l'*istruzione*$_2$.

ESEMPIO 5.4

```
var esa: char; decimale: 0..15; {si veda l'Esempio 5.2}
...
   if (esa ≤ 'a') and (esa ≤ 'f') then
      decimale := ord(esa) - ord('a') + 10
      else decimale := ord(esa) - ord('0')
      {si noti che, nell'espressione booleana, dopo if, le parentesi sono obbligatorie
         e che 'a' ≤ esa ≤ 'f' non sarebbe corretta}
```

ESEMPIO 5.5

```
procedure EquazioniQuadratiche
      (a, b, c: real {a≠0, ax²+bx+c = 0};
      var r1, i1, r2, i2: real {le radici r1 e r2 dell'equazione rappresentata da a,
         b, c sono le parti reali; i1 e i2 sono le corrispondenti parti immaginarie}
      );
var discriminante: real;
begin {EquazioniQuadratiche}
   discriminante := sqr(b) - 4 * a * c;
   if discriminante ≥ 0 then
   begin {calcola le due radici reali; per evitare errori di cancellazione quando b²
         è molto maggiore di 4ac, si calcola dapprima la radice maggiore}
      if b > 0 then
      r1 := -(b + sqrt(discriminante))/(2 * a)
      else r1 := (sqrt(discriminante) - b)/(2 * a);
      if r1 = 0 then r2 := 0 {per evitare una divisione per zero}
      else {si ricorre alla relazione x1 * x2 = c/a}
         r2 := c / (a * r1);
      i1 := 0; i2 := 0 {nessuna parte immaginaria}
   end {radici reali; (x-r1) * (x-r2) = 0}
   else {radici complesse}
   begin
      r1 := -b / (2 * a); r2 := 1;
      i1 := sqrt(-discriminante)/(2 * a); i2 := -i1
   end {radici complesse}
end {EquazioniQuadratiche}
```

COMMENTI

1. Contrariamente a quanto accade, per esempio in PL/1, un punto e virgola prima di un **else** è sempre considerato un errore, dal momento che un'istruzione non può cominciare mai con un **else**, che è solo un componente della struttura composta, come **if** e **then**. Ricordiamo che, in Pascal, il punto e virgola è un separatore e non un terminatore.
2. Un confronto fra gli Esempi 5.2 e 5.4 mostra che, anche nel caso in cui vi siano solo due alternative, l'istruzione **if** non è necessariamente preferibile a **case**. In questo particolare esempio è senza dubbio peggiore, dal momento che non consente di riconoscere i valori illegali di *esa* (come ad esempio uno spazio, o un segno di interpunzione), riconoscimento che è invece possibile con la **case**. [Si riconosce tuttavia che il modo migliore di procedere consiste nel rilevare esplicitamente i valori non permessi, usando, per esempio, un set di caratteri (si veda il Capitolo 12). Il punto importante, che si vuole sottolineare, è che l'istruzione **case** elenca in modo esplicito tutte le situazioni lecite, mentre ciò non è vero per un'istruzione **if**.] Più in generale, un'istruzione **case** è più semplice, più sicura, più comprensibile e più efficiente di un'istruzione **if** in tutti quei casi in cui ci siano più alternative, mutuamente esclusive e con la stessa probabilità di essere scelte. Benché istruzioni molto complicate, con parecchi **if** annidati, siano sintatticamente corrette, è meglio evitarle a favore di un'unica **case**.
3. Si noti la cura posta, nell'Esempio 5.5, nel calcolare le radici, per evitare errori di cancellazione. Si noti ancora, in questo primo esempio non banale, come comincino ad apparire, nei punti critici, le prime asserzioni, per convincere il lettore della correttezza della procedura.

ESERCIZI

5.1 Ricerca del minimo

Dato un tipo scalare *t*, si scriva una funzione con la seguente intestazione:

function *Minimo* (*a,b*: *t*) : *t*

che calcoli il valore minimo nella coppia (*a, b*).

*5.2 Stampa di un titolo

L'elenco che segue riporta i titoli con cui ci si può rivolgere ad una persona, in riferimento alla sua professione: Architetto, Avvocato, Ingegnere, Dottore, Professore, Geometra e Ragioniere, che sono codificati con le seguenti lettere *A*, *V*, *I*, *D*, *P*, *G*, *R*. Si scriva una procedura con la seguente intestazione:

procedure *StampaTitolo* (*titolo*: *char*)

che stampi in chiaro il titolo corrispondente a un dato codice. Viene garantito che il parametro attuale *titolo* assuma solo valori permessi.

5.3 Analisi di un carattere

Tutti i caratteri possibili sono suddivisi nei seguenti insiemi: lettere, cifre, parentesi (tonde, quadre, graffe e angolari), terminatori (dollaro, cancelletto, punti esclamativi e interrogativi), spaziature (il solo spazio) e separatori (tutti gli altri caratteri). Data la seguente definizione di tipo:

type *classe* = (*lettera, cifra, parentesi, terminatore, separatore, spazio*)

si scriva la funzione

function *Analizza* (*c*: *char*): *classe*

che valuta la classe di appartenenza di un dato carattere *c*.

6

Strutture iterative

Le istruzioni descritte nei capitoli precedenti non forniscono alcun modo per descrivere la ripetizione di un'azione: si può sempre riscrivere la stessa istruzione per il numero necessario di volte, oppure si può ricorrere a quello strumento molto potente, ma difficile da usare, che è dato dalle procedure recursive, argomento che verrà descritto solo nel Capitolo 14. Di conseguenza, i costrutti linguistici che consentono la ripetizione di un'azione costituiscono, per il Pascal, una dimensione necessaria ed estremamente importante.

Esistono tre costrutti diversi: l'istruzione **while**, l'istruzione **repeat** e l'istruzione **for**. I primi due rappresentano dei modi per costruire istruzioni iterative e vengono usati quando non si sa a priori il numero esatto di esecuzioni, che dipende dal verificarsi di una determinata condizione. Si fa uso di questo tipo di istruzione quando l'esecuzione ripetuta di un'azione è utilizzata per raggiungere un determinato obiettivo.

Al contrario, l'istruzione **for** serve per costruire istruzioni ripetitive ed è usata quando si sa in anticipo il numero di ripetizioni da effettuare. Le istruzioni ripetitive sono di solito usate quando si debba compiere la stessa azione su ciascun oggetto appartenente ad un insieme predeterminato.

Non esiste, in Fortran e in molti dialetti BASIC, un equivalente delle istruzioni Pascal **while** e **repeat**. Il Cobol mette a disposizione il verbo PERFORM con l'opzione UNTIL, ma l'istruzione che deve essere ripetuta deve comparire in un'altra parte del programma. Solo il PL/1 ha una struttura equivalente con il costrutto DO WHILE.

Dal momento che le istruzioni ripetitive sono più complicate delle istruzioni iterative e possono essere sintetizzate con l'aiuto di queste ultime, la loro descrizione è rimandata al Capitolo 10.

6.1 L'ISTRUZIONE while

Per usare questo costrutto iterativo occorre specificare due cose: l'istruzione che deve essere ripetuta e la condizione (un predicato, per esempio un'espressione booleana) che deve essere soddisfatta prima dell'esecuzione dell'istruzione.

Sintassi

Questo costrutto linguistico non è completamente parentesizzato, dal momento che nessun simbolo ne segna la fine (Figura 6.1). Di conseguenza occorre usare un'istruzione composta tutte le volte che l'azione che si deve ripetere richiede più di un'istruzione. Tuttavia l'istruzione **while** è sintatticamente equivalente ad un'istruzione semplice e può quindi essere usata laddove un'istruzione semplice è permessa. Quest'ultima osservazione è valida per ogni costrutto linguistico del Pascal e d'ora in poi non sarà più ripetuta.

Semantica

Data l'istruzione **while**

while *espressione* **do** *istruzione*
se {P ∧ espressione} *istruzione* {P}
allora {P} istruzione **while** {P ∧ ¬espressione}

Questa regola di verifica non dice in modo esplicito che l'istruzione deve essere ripetuta.
L'esecuzione di questo costrutto implica lo svolgimento dei seguenti passi:

a) dapprima viene valutata l'espressione;
b) se questa dà un valore *false*, l'istruzione **while** termina e la regola di verifica è soddisfatta;
c) se invece ha un valore *true*, l'istruzione viene eseguita (mantenendo così vera l'asserzione *P*) ed il controllo ritorna alla valutazione dell'espressione.

Così, se l'espressione è *false* prima dell'esecuzione dell'istruzione **while**, non si deve fare niente.
La regola di verifica dell'istruzione **while** è estremamente semplice e mette chiaramente in evidenza sia *P*, la condizione che è mantenuta dall'istruzione (l'invariante del ciclo), sia *espressione*, la cui negazione rappresenta l'obiettivo del ciclo (da raggiungere nell'istruzione).

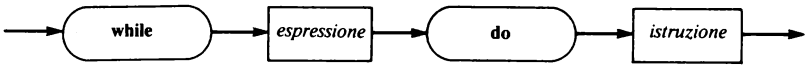


Figura 6.1 Diagramma sintattico di un'istruzione **while**

ESEMPIO 6.1: DIVISIONE FRA INTERI

```
type naturale = 0.. maxint;
procedure DivisioneFraInteri
   (a, b: naturale {dividendo e divisore; a ≥ 0, b > 0};
   var q,r: naturale {quoziente e resto; a = bq+r; 0 ≤ r < b}
   );
begin {DivisioneFraInteri}
   r := a; q := 0;
   while r ≥ b do
      begin {a = bq+r; 0 ≤ r}
         r := r-b; q := q+1
      end
   {a = b×q+r; 0 ≤ r < b}
end {DivisioneFraInteri}
```

COMMENTI

1. L'invariante del ciclo è espresso sotto forma di asserzione proprio all'inizio dell'istruzione **while**, vicino all'espressione. L'insieme di queste due condizioni rappresenta la miglior spiegazione del funzionamento del ciclo e non scriveremo mai un'istruzione iterativa senza averne chiaramente evidenziato l'invariante, sia in modo formale, come in questo esempio, che a parole, come nell'Esempio 6.3.
2. Se $a < b$, il ciclo non viene eseguito affatto, dando per risultato un quoziente pari a zero ed un resto pari a b. In questo caso, questa proprietà dell'istruzione **while** è fondamentale.
3. Dal momento che anche le persone più ragionevoli possono non trovarsi d'accordo su cosa sia il risultato di una divisione fra interi con operandi negativi, si è preferito non trattare questo caso nella procedura. La definizione del tipo corrispondente è un modo semplice e sicuro per garantire che a e b non saranno mai negativi.
4. Naturalmente questa procedura non è affatto necessaria, dal momento che il Pascal ha l'operatore **div**, che è definito anche per operandi negativi.

ESEMPIO 6.2: RICERCA LINEARE

```
type sequenza = ...
   {un tipo non specificato, che descrive una successione di oggetti di un dato tipo
      semplice T, con i seguenti operatori (procedure):
         — procedure Inizializza (var s: sequenza) prepara s ad essere esaminata;
         — function FineSequenza (var s: sequenza): Boolean true se non esistono
           più elementi di s, false se esiste almeno un elemento;
```

```
        — function Successivo (var s: sequenza): T dà come risultato l'elemento
          successivo di s, se esiste};
procedure Ricerca
    (var s : sequenza {la successione da esaminare};
    x : T {il valore da ricercare in s};
    var i : integer {il rango del primo oggetto uguale a x in s, se ne esiste uno};
    var trovato: Boolean {true se esiste, in s, almeno un oggetto uguale a x, false
                          altrimenti}
    );
begin {Ricerca}
    Inizializza(s); i := 0; trovato := false;
    while not FineSequenza(s) or trovato do
    begin {i è il rango dell'elemento successivo in s; s1, s2,..., si-1 sono tutti diversi
           da x}
    i := i+1;
    if x = Successivo(s) then trovato := true
end
    {o trovato ∧ x = i-esimo elemento di s
    o ¬ trovato ∧ nessun elemento di s vale x}
end {Ricerca}
```

COMMENTI

La condizione, nell'istruzione **while** di questo esempio, è piuttosto complicata principalmente perché il ciclo deve finire al verificarsi di una di due diverse condizioni: quando si trova *x* o quando si arriva alla fine della sequenza. Nell'Esempio 6.6 si vedrà come un uso intelligente di un tipo scalare porti ad avere un ciclo più leggibile e più efficiente.

ESEMPIO 6.3: RICERCA BINARIA

```
type tabella = ...
    {un tipo non specificato, che descrive un insieme di oggetti numerati di un dato
    tipo semplice T, con i seguenti operatori:
        — function EstremoInferiore (var tb: tabella): integer fornisce l'oggetto con
          numero minimo in tb;
        — function EstremoSuperiore (var tb: tabella): integer fornisce l'oggetto con
          numero massimo in tb;
        — function Componente (var tb: tabella; i: integer): T fornisce il valore
          dell'i-esimo elemento di tb (l'oggetto che ha numero i)};
procedure RicercaBinaria
        (var tb : tabella {la tabella da esaminare, che deve essere ordinata, cioè
          Componente (tb,j) ≤ Componente (tb,k) se j < k};
        x : T {il valore da cercare in tb};
```

```
    var trovato: Boolean {true se almeno un elemento di tb vale x, false altrimenti};
        var i : integer {se trovato è true, Componente(tb,i) = x};
        );
    var sinistro, destro: integer {limiti della sottotabella da esaminare};
begin {RicercaBinaria}
    sinistro := EstremoInferiore(tb); destro := EstremoSuperiore(tb);
    trovato := false;
    while not trovato and (sinistro ≤ destro) do
    begin {x non compare in tb prima di sinistro o dopo destro}
        i := (sinistro + destro) div 2 {elemento centrale};
        if Componente(tb, i) = x then found := true
        else if Componente(tb, i) < x then
            {x non può essere prima di i, perché tb è ordinata}
            sinistro := i + 1
        else {Componente(tb,i) > x; x non può essere dopo i}
            destro := i - 1
    end {while ¬ trovato ∧ (sinistro ≤ destro)}
end {RicercaBinaria}
```

COMMENTI

1. Esistono molte varianti di questo ben noto algoritmo. Quella qui presentata minimizza il numero di confronti necessari, ma ha una condizione di test piuttosto complicata, per la stessa ragione vista nell'Esempio 6.2. L'Esempio 6.4 presenta un'altra variante con una condizione più semplice.
2. Il ciclo di **while** è abbastanza lungo, per cui è utile ripeterne l'intestazione alla fine, sotto forma di commento. Alcuni compilatori, o anche solo dei programmi appositi per la paragrafatura, lo fanno in modo automatico.

ESEMPIO 6.4: RICERCA BINARIA (VARIANTE)

```
type tabella = ... {come nell'Esempio 6.3};
procedure RicercaBinariaDue {stessi parametri di RicercaBinaria}
        (var tb: tabella; x: T; var i: integer; var trovato: Boolean);
    var sinistro, destro: integer;
begin {RicercaBinariaDue}
    sinistro := EstremoInferiore(tb); destro := EstremoSuperiore(tb);
    while sinistro ≤ destro do
    begin {x non compare, in tb, prima di sinistro o dopo destro}
        i := (sinistro + destro) div 2;
        if Componente(tb, i) ≤ x then {x non compare prima di i}
            sinistro := i + 1;
        if Componente(tb, i) ≥ x then {x non compare dopo i}
            destro := i - 1
        end {while sinistro ≤ destro}
```

trovato := *x* = *Componente*(*tb*, *i*)
end {RicercaBinariaDue}

COMMENTI

Questa procedura richiede che esista una tabella non vuota; altrimenti il riferimento a *Componente*(*tb*, *i*) nell'ultima istruzione è illegale.

6.2 L'ISTRUZIONE repeat

In alcuni casi l'istruzione oggetto dell'iterazione deve essere eseguita almeno una volta. Il costrutto **while** va bene, anche se per certi versi è ridondante, solo se si garantisce che la condizione logica, posta al suo inizio, sia vera. In altri casi, avere a disposizione un costrutto linguistico che esamina il valore della condizione logica dopo l'esecuzione dell'istruzione rende il programma più semplice e più leggibile.

Sintassi

Dal momento che l'azione che si vuole ripetere è già racchiusa all'interno dei simboli **repeat** e **until**, non è necessario l'uso di un'ulteriore coppia di parentesi **begin-end** per le istruzioni composte. Si esamini la Figura 6.2.

Semantica

Data l'istruzione **repeat**:

repeat *istruzione* **until** *espressione*
se {P} *istruzione* {Q}
e {Q ∧ ¬espressione} *istruzione* {Q}
allora {P} istruzione **repeat** {Q ∧ espressione}

Come per l'istruzione **while**, neppure questa regola di verifica mette in evidenza il fatto che l'istruzione debba essere ripetuta. L'esecuzione di questo costrutto linguistico è la seguente:

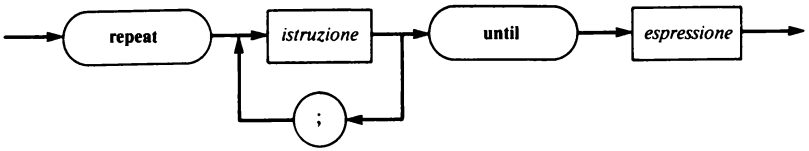


Figura 6.2 Diagramma sintattico di un'istruzione **repeat**

a) si esegue dapprima l'istruzione che precede l'asserzione Q
b) si valuta quindi l'espressione; se è falsa il controllo ritorna all'esecuzione dell'istruzione
c) se l'espressione è vera, l'istruzione **repeat** termina e la regola di verifica è soddisfatta.

La regola di verifica è più complicata dell'analoga vista per il costrutto **while**, perché deve esprimere il fatto che l'istruzione viene eseguita almeno una volta. Dal momento che l'espressione è vera quando l'istruzione **repeat** termina, questa stessa rappresenta la condizione di terminazione, mentre Q è l'invariante del ciclo. Al contrario di quanto accade nell'istruzione **while**, l'invariante di una **repeat** non è generalmente il suo antecedente.

ESEMPIO 6.5: RICERCA LINEARE

```
{negli Esempi 6.2, 6.3 e 6.4 la successione, o la tabella da esaminare, non è mai vuota, cosicché si può utilizzare un'istruzione repeat; viene qui presentata una variante dell'Esempio 6.2}
type sequenza = ... {come nell'Esempio 6.2};
procedure RicercaConRepeat {stessi parametri di Ricerca}
   (var s : sequenza; x : T; var i : integer; var trovato: Boolean);
begin {RicercaConRepeat}
   Inizializza(s); i := 0; trovato := false;
   repeat
      {i è il rango dell'elemento successivo di s
      s1, s2,..., si-1 sono tutti diversi da x}
      i := i+1;
      if x = Successivo(s) then trovato := true
   until FineSequenza(s) or trovato
end {RicercaConRepeat}
```

COMMENTI

1. L'invariante del ciclo di **repeat** è vero fin dall'inizio dell'esecuzione; in altre parole coincide con l'antecedente. Questo dimostra che un ciclo **while** sarebbe stato del tutto equivalente.
2. Osserviamo ancora che la condizione di terminazione è piuttosto complessa e che l'uso di un indicatore a tre valori, al posto della variabile booleana, consente di strutturare il ciclo in modo più efficiente e più semplice, come si vedrà nell'Esempio 6.6.

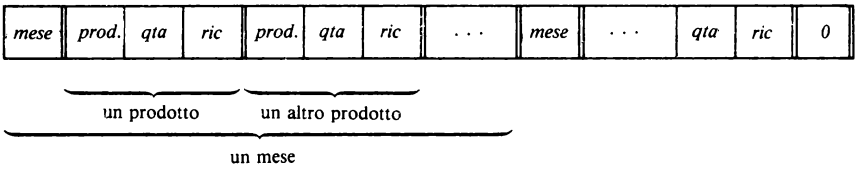


Figura 6.3 Struttura della sequenza dell'Esempio 6.6

ESEMPIO 6.6: VENDITE MASSIME

{i record delle vendite di alcuni negozi sono preparati come sequenze di informazioni codificate come interi, una sequenza per anno; la sequenza di un dato anno ha la struttura mostrata in Figura 6.3:

- mese è un intero fra 0 e 12 dove 0 denota la fine della sequenza;
- prod. è il codice di ogni prodotto, ovvero un intero fra 1000 e 99999; i prodotti sono in ordine crescente di codice in ogni mese;
- qta è la quantità di un dato prodotto venduta nel mese;
- ric è il corrispondente riciclo;

le procedure seguenti, dato un prodotto e un anno, determinano in quale mese si sono verificate le vendite più elevate ed il corrispondente riciclo; si suppone esista una procedura PreparaRecordVendite, che predispone la lettura della sequenza relativa ad un dato anno}

```
type mese = 0..12; anno = 1970..2000;
   codiceprodotto = 1000..99999;
procedure VenditeMassime
      (a: anno {l'anno da esaminare};
      prodotto: codiceprodotto {il prodotto in esame};
      var qta, ric: integer {qta massima e ric corrispondente};
      var ilmese: mese {mese corrispondente}
      );
   const finesequenza = 0;
      mincodiceprodotto = 1000 {per distinguere un prodotto da un mese};
   var mesecor: mese;
      qtacor, riccor: integer {massimi qta e ric correnti};
      n: integer;
      esitoricerca: (inricerca, trovato, mancante);
   procedure SaltaAFineMese;
      {la variabile globale n è l'ultimo valore letto, o un mese o un prodotto; al termine di questa procedura, si sarà posizionati sul primo prodotto del mese successivo, il cui numero è in n}
      var qta, ric: integer {per valori ignorati};
   begin {SaltaAFineMese}
      while n ≥ mincodiceprodotto do
         {non si è all'inizio del mese successivo}
```

```
            read(qta,ric,n)
        end {SaltaAFineMese};
    begin {VenditeMassime}
        PreparaRecordVendite(a);
        qtacor := 0; read(mesecor);
        while mesecor ≠ finesequenza do
        begin {elaborazione di un mese}
            esitoricerca := inricerca {prodotto non ancora trovato};
            read(n) {prodotto corrente};
            repeat {ricerca il prodotto desiderato}
                if (n < prodotto) and (n ≥ mincodiceprodotto) then {continua}
                    read(qta,ric,n)
                else if n = prodotto then esitoricerca := trovato
                else esitoricerca := mancante
            until esitoricerca ≠ inricerca;
            case esitoricerca of
            trovato:
                begin read(qta,ric) {parametri corrispondenti}
                    if qta > qtacor then {vendite più elevate in questo mese}
                    begin
                        qtacor := qta; riccor := ric;
                        ilmese := mesecor
                    end;
                    read(n); SaltaAFineMese
                end {trovato};
            mancante: SaltaAFineMese
            end {case esitoricerca};
            {l'ultimo valore letto è n, che non è il codice di un prodotto}
            mesecor := n
        end {elaborazione di un mese};
        qta := qtacor; ric := riccor {trasmette i risultati}
    end {VenditeMassime}
```

COMMENTI

1. Quello appena proposto è solo un esempio e non va necessariamente preso a modello; questa considerazione vale per la maggior parte delle altre procedure (o programmi completi) presentati in questo volume. Esistono, naturalmente, molte altre soluzioni al problema dato; per di più la nostra soluzione non mira ad essere perfetta. Il suo difetto principale è di non essere tollerante ai malfunzionamenti, cioè quando in input riceve dei dati non corretti, invece di inviare messaggi di errore si limita a dare dei risultati sbagliati. Per esempio, se l'oggetto cercato non appare affatto nella sequenza, *qta* = 0, ma le variabili *ric* e *mesecor* sono assolutamente prive di significato. Si può

fare inoltre un'altra osservazione, meno importante, sulle variabili locali *qtacor* e *riccor*, che non sono necessarie e possono essere sostituite ovunque da *qta* e *ric*.
Il ciclo **while** nella procedura ausiliaria *SaltaAFineMese* non può essere sostituito da un ciclo **repeat** equivalente, dal momento che questa procedura può essere chiamata anche alla fine del mese.

3. Si osservi l'uso del tipo scalare (anonimo) per la variabile *esitoricerca*. La condizione di terminazione per il ciclo **repeat** diventa ora molto semplice, dal momento che non è necessario tenere conto di due diverse condizioni di terminazione, che alla fine del ciclo sono disgiunte. Uno dei maggiori vantaggi di questa soluzione è la sua chiarezza, che rende superfluo inserire commenti nella procedura.

ESERCIZI

6.1 Flusso di cassa

Un registratore di cassa prepara un nastro con un elenco di tutti gli importi che sono stati battuti sulla sua tastiera (un numero intero in lire). In testa al nastro c'è l'identificatore di cassa. Si scriva una procedura che calcoli il flusso di cassa globale di un registratore, dato il suo nastro. Per risolvere il problema si faccia uso del tipo *sequenza* e dei relativi operatori descritti nell'Esempio 6.2.

6.2 Gioco di dadi

Due giocatori lanciano contemporaneamente un dado ciascuno. Il giocatore che fa il punteggio più alto segna un punto. Il gioco si conclude quando un giocatore raggiunge gli 11 punti. Si scriva un programma che simuli questo gioco, usando una funzione predefinita *random(n)*, che produce un numero pseudo-casuale compreso fra *1* ed il suo argomento *n*.

*6.3 Radici di un'equazione

È possibile calcolare una radice dell'equazione $f(x) = 0$ usando il metodo di Erone di Alessandria (erroneamente attribuito a Newton) che calcola il valore della serie data dalla seguente formula iterativa:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Dato un valore iniziale x_0, che sia un'accettabile approssimazione della radice, la serie converge. Si scriva una procedura che utilizza questo metodo e che verifichi la convergenza, contando il numero di iterazioni calcolate a un dato istante e fermandosi se questo valore supera il numero massimo di iterazioni prefissato, senza avere raggiunto la precisione voluta.

```
procedure RadiciDiErone
          (xzero: real {approssimazione iniziale della radice};
          function F(x: real): real {la funzione vera e propria};
          function Fprima(x: real): real {la sua derivata};
          epsilon: real {la precisione relativa desiderata};
          var convergenza: Boolean {vale true se la precisione desiderata è ottenuta prima
          del massimo numero di iterazioni previsto};
    var soluzione: real {la radice desiderata, se convergenza è true}
    );
```

7
I file

Tutti i tipi di dati introdotti nel Capitolo 2 e usati fino ad ora sono tipi semplici o scalari: tutti gli oggetti di questi tipi sono cioè atomici e non possono essere ulteriormente divisi in componenti. Naturalmente questa considerazione è vera solo a patto di non voler scendere fino al livello di rappresentazione di questi oggetti: per esempio, un numero intero è atomico, ma non lo è la sua rappresentazione decimale, che è costituita da una sequenza di cifre.
È chiaramente necessario introdurre un'altra categoria di oggetti, ciascuno dei quali consta di più elementi (o componenti), chiamati *oggetti strutturati* (o anche *aggregati*). Questi sono presenti in quasi tutti gli altri linguaggi di programmazione, ma il Pascal è stato il primo linguaggio ad inglobarli nel concetto più generale di tipo. Così in Pascal un oggetto strutturato deve appartenere ad un *tipo strutturato* ed è in realtà una collezione di oggetti che possono, a seconda dei casi, essere considerati e trattati sia singolarmente che come un tutto unico. Esistono un'infinità di strutture possibili, che differiscono fra di loro non solo per il tipo dei componenti, ma anche per le eventuali restrizioni imposte alle dimensioni, all'accesso, all'aggiornamento (tralasciando altre possibili operazioni) dei dati strutturati o dei loro singoli componenti e per l'efficienza relativa di queste operazioni. Per di più alcuni tipi astratti, come sequenze, stack o alberi binari, possono essere realizzati in molti modi diversi, ciascuno dei quali ha vantaggi e svantaggi. Perciò il Pascal, come la maggior parte degli altri linguaggi di programmazione, non fornisce direttamente queste strutture astratte, ma solo pochi metodi fondamentali, che possono essere usati per realizzare tali strutture in modi diversi.
Nei Capitoli 7, 9, 11, 12 e 13 verranno descritti i cinque metodi di definizione di tipi strutturati disponibili in Pascal. Alcuni hanno un equivalente negli altri linguaggi, altri no; verranno comunque fatti confronti con Fortran, Cobol, BASIC, e PL/1. Dal momento che un elemento di un tipo strutturato può essere a sua volta strutturato, questi cinque diversi metodi consentono al programmatore di definire una varietà infinita di tipi strutturati.
Per ciascun metodo descriveremo dapprima un tipo di dati astratto e struttura-

to, con l'elenco delle proprietà generali auspicabili per tale tipo. La struttura equivalente con cui verranno successivamente costruiti in Pascal, consentirà di ottenerne un'implementazione efficiente, al prezzo della rinuncia di alcune delle proprietà generali. Nel caso tali restrizioni non siano accettabili, si potrà ricorrere, quasi sempre, ad altre strutture dati, che saranno descritte brevemente nel testo o negli esempi.
La descrizione di un tipo aggregato deve specificare il metodo di strutturazione utilizzato ed il tipo (o i tipi) dei singoli elementi. I diversi metodi di strutturazione del Pascal sono il file, l'array, il record, il set e il puntatore e sono, rispettivamente, l'implementazione dei seguenti concetti matematici astratti: la successione, la trasformazione, il prodotto cartesiano, gli insiemi e le strutture recursive. Questo non significa assolutamente, come sarà mostrato più volte, che l'unico modo per realizzare una successione sia, per esempio, il file, o che l'unica implementazione possibile del concetto di trasformazione sia l'array.

7.1 SUCCESSIONI

La successione è una delle strutture astratte più semplici. Consiste semplicemente in una sequenza di oggetti tutti dello stesso tipo, cosicché si è in grado di conoscere il tipo di ciascun elemento indipendentemente dalla sua posizione relativa. Per definirne le proprietà, introdurremo un insieme molto limitato di operatori, ponendo molta attenzione nell'usare un'annotazione che non sia già stata usata dal Pascal, per evitare ogni possibile confusione fra metodi di strutturazione astratti e concreti. Di questi concetti, inoltre, non verrà data una formalizzazione completa, perché ciò andrebbe al di là degli scopi di questo volume e degli stessi interessi reali di un programmatore medio. Si può comunque trovare una formalizzazione completa in altri libri [si veda soprattutto il Capitolo 1 di Dahl, Dijkstra e Hoare (1972) e Alagič e Arbib (1978)].
Sia S un tipo di struttura sequenziale, i cui elementi siano di tipo T; indicheremo con $S <>$ la sequenza vuota e con $S <t>$ (t di tipo T) la sequenza che contiene solo t. Chiameremo $\&$ l'operatore di concatenazione, che ha come operandi due successioni dello stesso tipo e che produce come risultato una successione, sempre dello stesso tipo, costituita dagli elementi della prima successione, nell'ordine originario, seguiti dagli elementi della seconda successione, sempre nell'ordine originario.
Tutti i differenti oggetti di tipo S possono essere costruiti per mezzo delle seguenti due regole:

1. $S <>$ è un oggetto di tipo S.

2. Se s è di tipo S e t di tipo T, allora $s\&S <t>$ è di tipo S.

Gli operatori che completano la definizione della struttura astratta chiamata successione sono:

a) ***firstof*** *S* dà il primo elemento di *S*, se esiste; ***firstof*** *(S <t>&s)* è *t*;
b) ***restof*** *S* produce la successione che si ottiene dopo avere cancellato il primo elemento di *S* (se *S* contiene almeno un elemento); ***restof*** *(S <t>&s)* è *s*;
e) ***lastof*** *S* dà l'ultimo elemento di *S*, se esiste; ***lastof*** *(s&S<t>)* è *t*.

Una successione di tipo *S* è costituita, ad un dato istante, da due parti — la sua ***parte sinistra*** e la sua ***parte destra*** — e da un ***modo di accesso***.

La parte sinistra e la parte destra sono ambedue successioni e possono essere vuote. Il modo di accesso può assumere due valori: *generazione* e *ispezione*. La successione stessa è la concatenazione della sua parte sinistra con la sua parte destra, esattamente in quest'ordine. Il primo elemento della parte destra di una successione è, in ogni istante, l'unico elemento della successione stessa che può essere esaminato o modificato, in funzione del valore assunto dal modo di accesso.
Avendo definito queste proprietà generali, si possono immaginare diverse implementazioni per questo tipo strutturato, che differiscono fra di loro per quelle proprietà considerate realmente importanti — che devono essere gestite in maniera efficiente — e quelle giudicate invece meno importanti — che possono essere gestite in modo meno efficiente o addirittura escluse dal linguaggio. Se, per esempio, si considera importante gestire intere successioni ed accedere facilmente ai singoli elementi, è possibile utilizzare una struttura di tipo **array** (si veda il Capitolo 9), ma in questo caso l'operazione di concatenazione risulterà molto gravosa ed il numero massimo di elementi per ogni successione sarà molto limitato. Se si vogliono gestire successioni di lunghezza variabile, senza dover contemporaneamente rinunciare ad un'efficienza accettabile nell'accesso al singolo elemento, si possono usare i puntatori (si veda il Capitolo 13) per realizzare una lista, ma in questo caso la lunghezza totale della successione sarà ridotta. Se non vogliamo avere alcun limite alla lunghezza della successione, si deve rinunciare all'uso della memoria centrale del calcolatore, penalizzando peraltro le operazioni di generazione ed ispezione. In Pascal è stata effettivamente scelta e realizzata quest'ultima alternativa, che è chiamata *file sequenziale*.

7.2 FILE SEQUENZIALI

Dal momento che in Pascal tutti i file sono sequenziali, l'aggettivo «sequenziale» verrà, d'ora in poi, omesso. Tuttavia, il non fornire alcun mezzo per accedere a file non sequenziali (che, per inciso, non servono a memorizzare successioni, ma per realizzare alcuni tipi di trasformazioni; si veda il Capitolo 9) è una chiara limitazione, allo stato attuale delle cose, del linguaggio. Parecchie implementazioni del Pascal prevedono delle estensioni non standard in questo settore.
Il file in Pascal è, contemporaneamente, un'implementazione del concetto astratto di successione e un'astrazione di un dispositivo periferico di input-

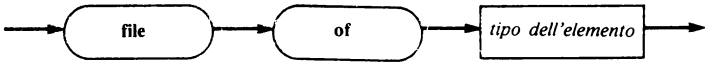


Figura 7.1 Diagramma sintattico di un tipo **file**

output, esistente nella realtà: il nastro magnetico. Perciò, al concetto di file è associato un insieme di operatori che, da una parte, rispettano le proprietà di una successione, dall'altra, le limitazioni imposte dal nastro magnetico. Le sue caratteristiche principali sono l'assenza dell'operazione di concatenazione, l'impossibilità dell'aggiornamento selettivo di un elemento e della generazione ed ispezione contemporanee di un dato file. Ciò perché il file è sequenziale: non si può perciò accedere ad un elemento senza avere prima scandito tutto il file fino a quel punto.

In un dato istante si può accedere ad un solo elemento del file, quello che corrisponde alla *posizione corrente* del file. Le operazioni sui file consentono di modificare la posizione corrente sia per accedere all'elemento successivo, sia per accedere al primo elemento dell'intero file. Per accedere all'elemento corrente il Pascal mette a disposizione una *variabile buffer*, che può essere considerata il nome di questo particolare elemento del file.

Un tipo **file** ha la forma sintattica mostrata in Figura 7.1. Il tipo dell'elemento può essere qualsiasi, semplice o strutturato, con l'unica importante limitazione che non deve contenere, né direttamente, né indirettamente, un elemento che sia ancora di tipo **file**. Questa limitazione è imposta da motivi legati all'implementazione, dal momento che il concetto di file deve essere un'astrazione efficiente di vari dispositivi periferici di input-output di tipo sequenziale: nastri magnetici, lettori o perforatori di schede, stampanti e terminali interattivi. Tuttavia il Pascal non pone alcuna limitazione sull'uso dei file come elementi di altri oggetti strutturati o sulla dichiarazione di variabili di tipo **file** in procedure interne al programma principale. Si noti che molte implementazioni impongono l'ulteriore restrizione che i file siano variabili semplici, dichiarate nel programma principale, per garantire che abbiano nomi distinti e visibili e che esistano dall'inizio alla fine dell'esecuzione di un programma. Questa importante restrizione, benché sia molto comune, non è standard, ed ha conseguenze spiacevoli nelle applicazioni del concetto di file; perciò non verrà più presa in considerazione negli esempi successivi (Esempi 7.5 e 7.7).

Data una variabile di file, cioè un identificatore di un file (non necessariamente una variabile semplice), il buffer associato è dichiarato implicitamente ed è rappresentato come indicato in Figura 7.2. Questo buffer può essere usato esatta-

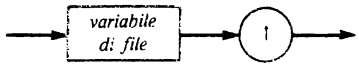


Figura 7.2 Diagramma sintattico della variabile buffer di un file

mente come ogni altra variabile, ed è dello stesso tipo degli elementi del file. L'unica restrizione è che è vietato modificare la posizione corrente di un file indirettamente mediante un riferimento al suo buffer, e ciò può capitare in tre diverse situazioni:

1. Quando il buffer è usato come parametro attuale di tipo variabile di una procedura che modifica la posizione corrente del file sia direttamente che indirettamente.
2. Quando il buffer è usato come elemento della lista di variabili di tipo **record** di un'istruzione **with** (si veda il Capitolo 11) che effettua questa modifica.
3. Quando il buffer compare nella parte sinistra di un'istruzione di assegnamento che effettua questa modifica (chiamando una funzione che provoca effetti collaterali).

Queste situazioni si verificano così raramente che la maggior parte dei compilatori non si preoccupa neppure di controllare la loro esistenza. In altri linguaggi di programmazione i file non sono oggetti strutturati, ma solo entità esterne al programma, cui si può accedere per mezzo di istruzioni dedicate. Nel BASIC standard manca la nozione stessa di file ed in Fortran esiste solo nella forma di unità logica di input-output, cui si può fare riferimento con un numero e senza che ad esso corrisponda un preciso elemento nel linguaggio. In PL/1 il «nome del file» rappesenta questa nozione, ma la filosofia implicita è la stessa del Fortran. Solo il Cobol fornisce qualche cosa di simile al Pascal, con la descrizione dei file nella DATA DIVISION: tuttavia è più restrittivo del Pascal, dal momento che gli elementi di un file hanno sempre la struttura di un record.
Per confondere un po' le idee al programmatore Cobol, osserveremo che il concetto di record esiste anche in Pascal (si veda il Capitolo 11) e che, benché non abbia una relazione particolare con il concetto di file, consente di definire e di trattare file i cui elementi sono record, situazione che si verifica comunemente in pratica.
Nessuno dei quattro linguaggi di confronto ha un concetto equivalente alla variabile buffer di un file.

7.3 OPERAZIONI SUI FILE

I tipi **file** sono gli unici su cui non sono consentite manipolazioni globali, dal momento che descrivono oggetti esterni alla memoria del calcolatore. Non è possibile effettuare operazioni di assegnamento e confronto fra file usando i normali operatori. Queste operazioni devono essere fatte in modo esplicito elemento per elemento, (si vedano gli Esempi 7.1 e 7.2). Di conseguenza, in Pascal, tutte le operazioni sui file vengono fatte per mezzo di parecchie procedure (predefinite) e di una funzione standard (anch'essa predefinita). Nella descrizione che segue, ciascuna procedura o funzione standard è descritta sia in modo

formale — con regole di verifica che fanno uso delle operazioni astratte definite sulle successioni nel paragrafo 7.1 — sia in modo informale, in italiano. Come qualsiasi altra variabile, una variabile di tipo **file** è indefinita al momento della sua dichiarazione. In tutto il resto del paragrafo si supporrà che valga la seguente dichiarazione:

var f: **file of** T

e, per convenzione, diremo che il file f si trova nello stato f_0 prima di una operazione e nello stato f_1 dopo che l'operazione è stata eseguita.
eof(f) è una funzione booleana standard che assume il valore *true* se la parte destra di f è vuota (siamo alla fine del file) e *false* in caso contrario. Se f è indefinito al momento di chiamare *eof*(f), si è in una situazione di errore. Questo predicato consentirà l'ispezione di tutto un file. Per completezza, *eof*(f) è definita anche quando il file è in fase di generazione (e vale sempre *true*), anche se ciò non è molto utile.

7.3.1 INIZIALIZZAZIONE

Esistono due operazioni per inizializzare variabili di file, in funzione del valore assegnato al modo di accesso: *rewrite* (f) prepara f per una generazione (scrittura) cancellando un eventuale contenuto precedente; *reset*(f) prepara f perché sia possibile ispezionarne (leggere) il contenuto, che deve essere stato generato in una precedente fase di scrittura. Ambedue le operazioni portano la posizione corrente del puntatore all'inizio del file.

{ }*rewrite* (f) {parte sinistra di f_1 = parte destra di f_1 = F< >
∧modo di accesso a f_1 = generazione
∧ f_1↑ è indefinito}

Dal momento che ***parte destra*** di f_1 è vuota, dopo un'operazione di *rewrite*(f) il predicato *eof*(f) vale *true*. Ricordiamo che f↑ è il buffer associato.

Caso 1

{parte sinistra di f_0 e parte destra di f_0 non sono indefinite e non sono ambedue vuote}
reset (f)
{parte sinistra di f_1 = F < >
∧ parte destra di f_1 = parte sinistra di f_0 & parte destra di f_0
∧modo di accesso a f_1 = ispezione
∧f_1↑ = firstof parte destra di f_1}

Caso 2

{parte sinistra di f_0 = parte destra di f_0 = F < >}
reset (f)

{parte sinistra di f_1 = parte destra di f_1 = F < >
∧ modo di accesso a f_1 = ispezione
∧ $f_1\uparrow$ è indefinito}

Il secondo caso è stato presentato solo per completezza, qualora si voglia ispezionare un file vuoto. Dopo avere eseguito *reset*(f), $f\uparrow$ assume il valore del primo elemento di f, se ne esiste uno; *reset*(f) è l'unica operazione che può riportare *eof*(f) al valore *false*.
Le operazioni *reset* e *rewrite* corrispondono a quelle che normalmente sono chiamate operazioni di apertura di un file, rispettivamente in input ed in output. In Cobol ed in PL/1 esistono operazioni di apertura esplicite. Tuttavia in Pascal non esiste l'operazione simmetrica di chiusura; si può pensare che *reset* (o *rewrite*), chiudano un file eventualmente già aperto prima di aprirlo. L'unica difficoltà consiste nell'assenza, nel linguaggio, della definizione dello stato del file fisico, alla fine dell'esistenza della variabile di file corrispondente. Per un file temporaneo, locale ad una procedura, si può ritenere che venga cancellato; per un file globale, definito nell'ambiente del programma (si veda il paragrafo 7.4) questo stato dipende da scelte esclusivamente fatte nella particolare implementazione.

7.3.2 INPUT E OUTPUT

Esistono due operazioni che consentono di modificare la posizione corrente in un file, una in corrispondenza del valore *generazione* del modo di accesso, l'altra in corrispondenza del valore *ispezione*. Nel primo caso deve essere stato assegnato al buffer un valore opportuno e questo valore viene concatenato alla fine del file. Nel secondo caso diventa accessibile, nel buffer, l'elemento successivo a quello corrente nel file, se esiste.

{modo di accesso a f_0 = generazione
∧ parte sinistra di f_0 non è indefinita
∧ parte destra di f_0 = F < >
∧ $f_0\uparrow$ non è indefinito}
put(f)
{modo di accesso a f_0 = generazione
∧ parte sinistra di f_1 = parte sinistra di f_0 &F $<f_0\uparrow>$
∧ parte destra di f_1 = F < >
∧ $f_1\uparrow$ è indefinito}

Ne consegue che *eof*(f) vale sempre *true* durante tutta la fase di generazione.

Caso 1

{modo di accesso a f_0 = ispezione
∧ parte sinistra di f_0 e parte destra di f_0 non sono indefinite
∧ parte destra di f_0 non è vuota

∧ restof parte destra di f_0 non è vuota}
get (*f*)
{modo di accesso a f_1 = ispezione
∧ parte sinistra di f_1 = parte sinistra di f_0 & firstof parte destra di f_0
∧ parte destra di f_1 = restof parte destra di f_0
∧ f' = firstof f_1}

Caso 2

{stesse asserzioni del caso 1, ma con restof parte destra di f_0 = F < >}
get (*f*)
{stesse asserzioni del caso 1, ma con parte destra di f_1 = F < > e f_1↑ indefinito}

Da quanto detto si vede come l'effetto di *get* (*f*) non sia definito se *eof* (*f*) vale *true* prima della chiamata: questo caso dà origine a un errore. Inoltre *put* (*f*) è un operatore valido solo durante una fase di generazione e *get* (*f*) solo durante una fase di ispezione. Ciò significa che le quattro operazioni appena definite non possono essere mischiate in alcun modo. In Figura 7.3 sono illustrate le sequenze corrette. Le due uscite dal grafo che sono senza nome corrispondono a file salvati o cancellati, a seconda del contesto in cui si opera (e probabilmente anche in funzione dei comandi del sistema operativo ospite).

ESEMPIO 7.1: COPIA DI UN FILE

```
type filediT = file of T;
   {T è di tipo qualsiasi, purché non contenga un file né direttamente né indirettamente}
procedure CopiaFile (var infile, outfile: filediT)
   {infile è copiato in outfile};
begin{CopiaFile}
```

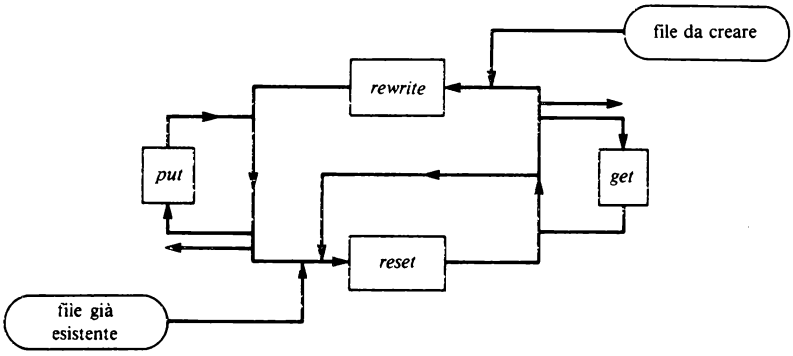


Figura 7.3 Possibili sequenze di operazioni di input-output

```
    reset(infile); rewrite(outfile);
    while not eof(infile) do
    begin {copia un elemento}
      outfile↑ := infile↑;
      put(outfile); get(infile)
    end
end {CopiaFile}
```

COMMENTI

1. La presenza del ciclo **while** rende la procedura funzionante anche nel caso il file sia vuoto. La struttura equivalente, basata su un'istruzione **repeat**, può essere usata solo se si è in grado di garantire che il file non è vuoto. Lo schema proposto in questo programma è molto generale e verrà usato in tutti i casi in cui si debba ispezionare un file.
2. Dal momento che non esiste un'istruzione di assegnamento tra file, un file non può essere un parametro attuale passato per valore ad una procedura; la trasmissione per valore implica un assegnamento del parametro attuale al parametro formale corrispondente, come si è visto nel Capitolo 4. Tutti i parametri di file saranno passati per indirizzo (parametri variabili).

ESEMPIO 7.2: CONFRONTO DI DUE FILE

```
type filediT = file of T
    {T è un qualsiasi tipo purché non contenga un file, né direttamente né indirettamente; oggetti di tipo T possono essere confrontati usando la funzione booleana data Uguali (a, b: T); naturalmente, se T è un tipo semplice, Uguali (a,b) può essere sostituita da a=b};
procedure ConfrontaFile (var file1, file2: filediT; var uguale: Boolean)
    {la variabile uguale è true se e solo se file1 e file2 sono uguali, cioè se hanno lo stesso numero di elementi e se l'i-esimo elemento di file1 è uguale all'i-esimo elemento di file2, con i che assume tutti i valori da 1 alla lunghezza dei file};
    var ricerca: Boolean;
begin {ConfrontaFile}
    reset(file1); reset(file2); ricerca := true;
    while ricerca do {parte sinistra di file1 = parte sinistra di file2}
      if eof(file1) then
        begin ricerca := false; uguale := eof(file2) end
      else if eof (file2) then
        begin ricerca := false; uguale := false end
      else if not Uguali(file1↑, file2↑) then
        begin ricerca := false; uguale := false end
      else
        begin get(file1); get(file2) end
end {ConfrontaFile}
```

COMMENTI

1. Benché l'istruzione del ciclo **while** abbia una struttura complessa, è un'istruzione semplice e come tale non richiede una coppia **begin-end**
2. Questa procedura è abbastanza rozza, dal momento che si ferma alla prima differenza di contenuto fra i due file e fornisce, come risposta solo un «sì» o un «no» senza neppure specificare quali siano gli elementi diversi nei due file. Una procedura che confronti file simili e che dica anche quali sono gli elementi diversi ha una struttura più complicata e verrà proposta come esercizio nel Capitolo 8.
3. La specifica della procedura impone l'uso di due indicatori logici, dal momento che la condizione di terminazione del ciclo è diversa dal valore assunto dalla funzione. Un indicatore a tre valori, come quello usato nell'Esempio 6.6, potrebbe semplificare il ciclo, consentendo di distinguere le seguenti situazioni: file uguali, fine di almeno un file ed elementi corrispondenti diversi. Tuttavia sarebbe necessario riesaminare la situazione al termine del ciclo e la procedura risultante sarebbe globalmente più complicata.

7.3.3 ABBREVIAZIONI

Le quattro procedure viste in precedenza, insieme alla funzione *eof*, sono del tutto sufficienti per le operazioni di input-output sui file ordinari (non cioè per quei file dedicati alla comunicazione uomo-macchina, che verranno trattati nel paragrafo successivo). Esistono tuttavia delle notazioni abbreviate per le operazioni di uso più frequente. La notazione

read (f, v_1, v_2,..., v_n)

dove v_1, v_2,..., v_n sono variabili (sia di tipo semplice che elementi di variabili strutturate) equivale a

begin *read* (f, v_1); *read* (f, v_2);...; *read* (f, v_n) **end**

e *read* (f, v) equivale a

begin $v := f\uparrow$; *get*(f) **end**

In modo del tutto analogo

write(f, e_1, e_2,..., e_n)

dove e_1, e_2,..., e_n sono espressioni, equivale a

begin *write* (f, e_1); *write* (f, e_2);...; *write* (f, e_n) **end**

e *write* (*f*,*e*) equivale a

begin *f*↑ := *e*; *put* (*f*) **end**

Si noti la completa simmetria che esiste fra le due notazioni e soprattutto il fatto che *get* (*f*) segua *v* := *f*↑. Ciò accade perché, non appena *f* è inizializzato, con un'operazione di *reset* (*f*), il suo primo elemento è disponibile nel buffer *f*↑.
Le procedure standard predefinite *read* e *write* sono comode da usare, ma in molte situazioni potrebbero comportare un overhead ed il programmatore non dovrebbe dimenticarsi dell'esistenza della *get* e della *put*, anche se non esiste, per queste, un equivalente nella maggior parte degli altri linguaggi di programmazione.

ESEMPIO 7.3: COPIA DI UN FILE (VARIANTE)

```
type filediT = file of T;
procedure CopiaFileDue (var infile, outfile: filediT)
   {una variante di CopiaFile (Esempio 7.1) usando read e write};
   var buffer: T;
begin {CopiaFileDue}
   reset(infile); rewrite(outfile);
   while not eof(infile) do
   begin {copia un elemento}
      read (infile, buffer);
      write (outfile, buffer)
   end
end {CopiaFileDue}
```

COMMENTI

Questa procedura è leggermente meno efficiente di *CopiaFile* (Esempio 7.1) a causa dei due assegnamenti impliciti, uno da *infile*↑ a *buffer* ed il secondo da *buffer* a *outfile*↑. Se il tipo *T* è complesso e di grosse dimensioni, l'overhead corrispondente non è trascurabile.

7.4 I FILE E L'AMBIENTE ESTERNO AL PROGRAMMA

Il concetto di file in Pascal è completamente indipendente dalla sua particolare implementazione su un dispositivo periferico di input-output. Per esempio, la procedura descritta nell'Esempio 7.1 potrebbe essere usata per copiare da un nastro magnetico a un disco rigido, da un floppy a un perforatore di schede, o

da una qualsiasi periferica ad un'altra, con l'unica limitazione che il primo dispositivo consenta operazioni di input ed il secondo di output e che su ambedue sia possibile la memorizzazione di oggetti di tipo *T*. Non esiste, in Pascal, alcun modo per definire nel programma le caratteristiche fisiche e logiche dei file o degli insiemi di dati reali cui sono associati i file astratti usati nel programma stesso, per il semplice motivo che queste caratteristiche — ed il modo in cui sono definite — dipendono molto dalle peculiarità dei vari dispositivi e dalle idiosincrasie dei sistemi operativi ospiti.

La definizione del Pascal nello Standard fa l'ipotesi (implicita) che i file possano essere separati in due categorie: i file interni — che sono solo uno strumento per memorizzare dati intermedi che non possono essere memorizzati nella memoria principale e che non sopravvivono al programma quando questo è terminato — e i file esterni — che esistono sia prima del programma (da cui sono ispezionati) che dopo (nel caso siano generati dal programma stesso). Per quanto riguarda la prima categoria di file, il programmatore non si deve assolutamente preoccupare della loro esatta organizzazione fisica ed un qualsiasi sistema operativo ospite, che non sia men che comprensivo, dovrebbe accettare senza difficoltà questi file temporanei. I file esterni, d'altro canto, sono stati preparati, o saranno processati, da altri programmi e deve perciò esistere un modo per descrivere i dettagli più rilevanti sul loro supporto fisico, la loro organizzazione, la loro posizione o identificazione logica e così via. La definizione dello Standard specifica che l'interfaccia fra un programma Pascal — che non dipende dal sistema ospite — ed il sistema operativo di quest'ultimo, sia descritta nell'intestazione del programma (si veda il Capitolo 1) e sia costituita dai nomi delle variabili di tipo **file**. Un programma Pascal può così essere considerato una specie di procedura, chiamata dal sistema operativo, i cui parametri sono i file esterni che ispeziona o che genera.

Occorre fare ora un'importante osservazione: la corrispondenza fra un file astratto, interno ad un programma Pascal, ed il file fisico esterno, residente su più possibili dispositivi periferici, è completamente statica, dal momento che viene stabilita solo nell'intestazione del programma. Durante la sua esecuzione non è pertanto possibile associare un file interno a file fisici esterni. Come conseguenza di tutto ciò, non è possibile scrivere, in Pascal Standard, un programma che, per esempio, elabori diversi file i cui nomi siano forniti dall'utente in modo interattivo.

Per evitare ogni possibile dipendenza dal sistema e per non imporre restrizioni implementative, la definizione data nello Standard non specifica come avvenga una chiamata ad un programma Pascal, come venga stabilita la corrispondenza fra file attuali ed i corrispondenti parametri formali di file e così via. In tutti gli esempi di programmi completi che compaiono nel resto di questo volume, tutti i nomi di file esterni saranno parametri del programma, e non vi saranno altri tipi di parametri nell'intestazione, dal momento che la loro presenza ed il loro significato dipendono esclusivamente dalla particolare implementazione.

ESEMPIO 7.4: CONFRONTO DI DUE FILE DI INTERI

```
program ConfrontaFileDiInteri (filesorgente, copiaprobabile)
   {questo programma confronta due file usando la procedura definita nell'Esempio 7.2};
   type filediT = file of integer;
   var filesorgente, copiaprobabile: filediT; risposta: Boolean;
   procedure ConfrontaFile {si veda l'Esempio 7.2, dove Uguali è sostituito da =}...;
begin {ConfrontaFileDiInteri}
   ConfrontaFile (filesorgente, copiaprobabile, risposta);
   if risposta then writeln('I due file sono identici')
   else writeln('Qualcosa non va')
end {ConfrontaFileDiInteri}.
```

COMMENTI

Come sarà spiegato nel Capitolo 8, l'intestazione di questo programma non è completa, dal momento che deve contenere anche *output*, il nome implicito del file sul quale verrà scritto il messaggio conclusivo.

7.5 ESEMPI PIÙ COMPLESSI

ESEMPIO 7.5: PREPARAZIONE DI UN FILE PER LA CONCATENAZIONE

```
type filediT = file of T;
procedure PreparaPerConcatenazione (var f: filediT)
   {dal momento che non si possono mischiare liberamente operazioni di ispezione
      e di generazione di un dato file e poiché non esiste la possibilità di chiudere un file o di sapere come sia stato aperto il file stesso, l'operazione di
      appendere una copia di un file alla fine di un altro (concatenazione di due
      file) non può essere eseguita senza fare delle ipotesi sullo stato iniziale del
      primo file; questa procedura evita il problema e lascia il parametro di tipo
      file in uno stato adatto per un'operazione di concatenazione al suo termine};
   var copialocale: filediT;
   procedure CopiaFile {si veda l'Esempio 7.1}...;
begin {PreparaPerConcatenazione}
   CopiaFile(f, copialocale);
   CopiaFile(copialocale, f);
   {parte sinistra di f1 = parte sinistra di f0 & parte destra di f0
      parte destra di f1 = filediT < >
      modo d'accesso a f1 = generazione}
end {PreparaPerConcatenazione}
```

COMMENTI

Il prezzo da pagare, per la generalità voluta, è alto, dal momento che il file deve essere copiato due volte. Il concetto di file, in Pascal, non si presta all'implementazione efficiente di alcune operazioni, fra cui la concatenazione. In modo analogo, sarebbe molto costoso cercare di modificare un elemento del file, perché l'operazione richiederebbe di fare due copie del file: la prima senza alcuna modifica, su di un file temporaneo, la seconda all'indietro verso il file originario, cambiando al volo l'elemento da modificare, nel momento in cui deve essere ricopiato. Il modo normale di modificare i file sequenziali è di raggruppare le modifiche in un lotto e quindi di usare una procedura di aggiornamento sequenziale (si vedano gli Esempi 8.5 e 11.7).

ESEMPIO 7.6: MERGE DI DUE FILE

```
type filediT = file of T
   {gli oggetti di tipo T sono, per ipotesi, confrontabili, per mezzo di una funzione
        booleana data MinoreDi (a, b): T, che probabilmente usa una chiave pre-
        sente in ogni oggetto; naturalmente se T è un tipo semplice, MinoreDi(a, b)
        può essere sostituito da a<b};
procedure MergeFile (var infile1, infile2, outfile: filediT)
   {i file infile1 e infile2 sono ordinati per mezzo della funzione MinoreDi; questa
        procedura fa il merge di infile1 e infile2, producendo outfile che contiene,
        in modo ordinato, tutti gli elementi dei due file di input};
begin {MergeFile}
   reset(infile1); reset(infile2);
   rewrite(outfile);
   while not (eof(infile1) or eof(infile2)) do
      {outfile è ordinato; outfile↑ ≤ infile1↑ e outfile↑ ≤ infile2↑}
      if MinoreDi (infile1↑, infile2↑) then
      begin {outfile↑ ≤ infile1↑ < infile2↑}
         write(outfile, infile1↑); get(infile1)
      end
      else {outfile↑ ≤ infile2↑ ≤ infile1↑}
      begin
         write(outfile, infile2↑); get(infile2)
      end;
      {al più, uno dei due file di input non è terminato}
   while not eof(infile1) do
   begin write(outfile, infile1↑); get(infile1) end;
   while not eof(infile2) do
   begin write(outfile, infile2↑); get(infile2) end
end {MergeFile}
```

COMMENTI

1. È perfettamente legale mischiare l'uso delle varie procedure di input-output *put* e *get* da un lato e *write* e *read* dall'altro. Qui *get* viene sempre usata per operazioni di input e *write* per quelle di output. Ciò evita la necessità di ricorrere a variabili temporanee esplicite per memorizzare gli elementi del file. Per le operazioni di «look-ahead» necessarie per il confronto degli elementi di ciascun file che si trovano sotto la testina di lettura, si usano i buffer dei file stessi.
2. Viene eseguito al più uno solo dei due cicli **while** che si trovano alla fine della procedura, dal momento che il ciclo di **while** principale termina quando si arriva alla fine di almeno uno dei due file di input. Perciò l'ordine relativo in cui sono posti gli ultimi due cicli è irrilevante.

ESEMPIO 7.7: ORDINAMENTO DI UN FILE

```
type filediT = file of T
procedure OrdinamentoNaturale (var f: filediT)
   {questa procedura ordina il file f secondo il metodo conosciuto come «ordina-
        mento naturale con merge»};
   var nup: integer {numero di passate};
      fip: Boolean {indicatore di fine passata};
      a, b: filediT {file temporanei}
   procedure CopiaRecord(var infile, outfile: filediT)
      {copia un elemento da infile a outfile e controlla se si è verificata una fine
         passata su infile};
      var buffer: T;
   begin {CopiaRecord}
      read(infile, buffer); write(outfile, buffer);
      if eof(infile) then fip := true
      else fip := MinoreDi (infile↑, buffer)
   end {CopiaRecord};
   procedure CopiaPassata (var infile, outfile: filediT)
      {copia una passata da infile a outfile};
   begin {CopiaPassata}
      repeat CopiaRecord(infile, outfile)
      until fip
   end {CopiaPassata}
   procedure Distribuisce {passata iniziale da f ad a e b};
   begin {Distribuisce}
      repeat CopiaPassata(f, a);
         if not eof(f) then CopiaPassata(f, b)
      until eof(f)
   end {Distribuisce};
```

```
    procedure MergePassata {da a e b ad f};
    begin {MergePassata}
      repeat {esegue il merge di una passata}
        if MinoreDi (a↑, b↑) then
        begin CopiaRecord(a, f);
          if fip then CopiaPassata(b, f)
        end
        else begin CopiaRecord(b, f);
          if fip then CopiaPassata(a, f)
          end
        until fip
    end {MergePassata};
    procedure MergeFile {a e b in f}
    begin {MergeFile}
      repeat MergePassata; nup := nup+1
      until eof(a) or eof(b);
      while not eof(a) do
      begin CopiaPassata(a, f); nup := nup+1 end;
      while not eof(b) do
      begin CopiaPassata (b, f); nup := nup+1 end
    end {MergeFile};
  begin {OrdinamentoNaturale}
    repeat {un passo di distribuzione e merge}
      reset(f); rewrite(a); rewrite(b);
      Distribuisce {passate da f ad a e b};
      reset(a); reset(b); rewrite(f);
      nup := 0; Mergefile {a e b in f}
    until nup = 1 {solo una passata su f, che viene così ordinato}
  end {OrdinamentoNaturale}
```

COMMENTI

1. La struttura globale di questa procedura è il risultato del processo adottato per la sua costruzione, di scomposizione in passi successivi. Naturalmente le chiamate alle procedure *Distribuisce, MergePassata* e *MergeFile*, che compaiono una sola volta, possono essere sostituite dalla loro espansione in linea e questo renderebbe l'intero processo più efficiente. Si noti tuttavia che la chiamata ad una procedura senza parametri è, in Pascal, estremamente economica.
2. Questa procedura fa uso di due soli file ausiliari, ma è meno efficiente di una procedura che ne usi tre, pur basandosi sullo stesso algoritmo; un terzo file consentirebbe di effettuare contemporaneamente il merge e la distribuzione di nuove successioni (più lunghe) in maniera più simmetrica. La procedura risultante compare in Alagič e Arbib (1978).

3. In realtà il processo di ordinamento del file *f* non verrebbe mai fatto su se stesso, dal momento che, così facendo, si distruggerebbe il suo contenuto originario. Con dei nastri magnetici, per esempio, il nastro iniziale verrebbe smontato subito dopo il primo passo di distribuzione e sostituito con un nastro vuoto, pronto a ricevere i valori ordinati. Non si può, tuttavia, richiedere questa gestione dei file direttamente in Pascal: occorre ricorrere a qualche procedura fornita dall'implementazione. Spesso si ottiene questo risultato aggiungendo dei parametri opzionali non standard alle procedure standard *reset* e *rewrite* o aggiungendo una procedura predefinita per la chiusura dei file.
4. La procedura *CopiaRecord* può essere resa un po' più efficiente evitando di usare la variabile ausiliaria *buffer* (si veda il commento al termine dell'Esempio 7.3). Occorre tuttavia fare molta attenzione perché il valore del buffer di un file di output è indefinito dopo una chiamata a *put*. Il corpo di *CopiaRecord* è, di conseguenza, meno leggibile di quello proposto nel seguente esempio:

```
begin {variante di CopiaRecord}
   outfile↑ := infile↑; get(infile);
   if eof(infile) then fip := true
   else fip := MinoreDi (infile↑, outfile↑);
   put (outfile)
end {variante di CopiaRecord}
```

ESERCIZI

*7.1 Elenco di elementi di un file

Un file per applicazioni gestionali è costituito da una serie di sequenze di cinque interi: rispettivamente un codice di prodotto, un prezzo unitario, una quantità (di magazzino), un mese e un anno. Gli ultimi due valori indicano quando è stato effettuato l'ultimo aggiornamento del prezzo unitario. Si scriva una procedura che produce un elenco di tutti quegli articoli i cui prezzi unitari non sono più variati a partire da una certa data, che viene passata come parametro di ingresso:

```
type filediinteri = file of integer;
procedure Elenco (var f: filediinteri; mese, anno: integer);
```

7.2 Aggiornamento selettivo di un file

Il file descritto nell'Esercizio 7.1 deve essere aggiornato per aumentare il prezzo unitario di tutti quei prodotti che non sono più stati aggiornati a partire da una certa data (*mese, anno*), di una percentuale *p1*, se il loro prezzo unitario è inferiore a *p*; di una percentuale *p2*, in tutti gli altri casi:

```
procedure Inflazione (var: vecchiofile, nuovofile: filediinteri;
                      mesecorrente, annocorrente: integer;
                      mese, anno: integer;
                      p1, p2: real {percentuale};
                      p: integer
                      );
```

7.3 Scarti e sconti

Nello stesso spirito dell'Esercizio 7.1, si decide di non trattare più quei prodotti che si ritiene essere troppo cari, o per cui le scorte siano al minimo, dal momento che questi articoli impegnano capitale in modo non redditizio. Si scriva una procedura che crea un nuovo file tenendo conto dei seguenti fattori: si deve scartare un articolo se il suo prezzo è maggiore di *prezzomassimo* o se la sua quantità in magazzino è minore di *qtaminima*. Questa procedura deve anche listare gli attributi (codice, quantità e prezzo) di tutti i prodotti scartati. Su tutti gli articoli scartati viene concesso uno sconto identico: la percentuale è *r1* se il prezzo globale è inferiore a *p1*; *r2* se il prezzo globale è compreso fra *p1* e *p2*; *r3* in tutti gli altri casi.

```
procedure Scarta (var: vecchiofile, nuovofile: filediinteri;
                  prezzomassimo, qtaminima: integer;
                  r1, r2, r3: real;
                  p1, p2: integer
                  );
```

8

Input-output di file di testo

8.1 INPUT-OUTPUT SU TERMINALE

I file presi in esame nel Capitolo 7 consentono di trasmettere (in input o in output) un valore per volta senza alcuna trasformazione di rappresentazione. Tutti i loro elementi sono dello stesso tipo che, di solito, è strutturato e sono quindi molto adatti ad immagazzinare e scambiare elevati volumi di informazioni ripetitive: proprio quello che viene in mente quando si pensa al concetto di «file». Per contro, non è possibile utilizzare questo tipo di file per gestire le comunicazioni fra uomo e macchina: infatti questi file non sono direttamente leggibili. Le loro informazioni devono dapprima essere codificate in una forma stampabile.

Una categoria di file molto importanti è pertanto rappresentata da quei file i cui elementi sono caratteri, anche se questi non sono ancora sufficienti per due ragioni. La prima è che sarebbe estremamente oneroso dover programmare le conversioni necessarie per gestire oggetti in input o in output, dalla loro rappresentazione esterna (come stringhe di caratteri) a quella interna e viceversa. La seconda ragione è che siamo abituati ad organizzare i dati prodotti non semplicemente come sequenze di caratteri, ma su linee, e questa struttura deve poter essere riconosciuta e riprodotta dal calcolatore. Un file dichiarato come **file of** *char* non sarebbe sufficiente e il Pascal fornisce un tipo predefinito particolare. Il suo nome è *text* e i file di questo tipo sono anche detti *textfiles* (o file di testo, ma preferiamo usare la prima notazione).

Molti linguaggi di programmazione fanno una distinzione fra input-output di tipo *record* e su *canale*. Nel primo caso gli elementi del file sono record, ciascuno dei quali ha più componenti, non necessariamente dello stesso tipo, mentre tutti i record di un file hanno la stessa struttura (con la possibilità di avere delle varianti). Durante le operazioni di lettura o scrittura non viene effettuata alcuna trasformazione di rappresentazione. Questo concetto è equivalente al concetto di file del Pascal, che tuttavia è più generale, poiché gli elementi di un file

possono essere di un tipo qualsiasi, essendo il record un caso particolare (si veda il Capitolo 11).
Nel caso invece di input-output su canale, i file sono considerati sequenze di caratteri, con un particolare meccanismo per controllarne la suddivisione in linee e durante ogni operazione di lettura o scrittura è necessario fare, in modo automatico, una trasformazione di rappresentazione. Questo concetto equivale al concetto di file di tipo *text* in Pascal.
La distinzione fra input-output di tipo record o su canale è particolarmente netta in PL/1, che li tratta con due insiemi distinti di istruzioni: READ e WRITE per il primo caso, GET e PUT per il secondo. Questa distinzione esiste anche in Fortran, dove l'input-output su canale (di gran lunga il più usato), fa uso dei FORMAT, che non sono richiesti da operazioni di input-output di tipo record (benché il concetto di record non sia definito in modo chiaro). In BASIC manca completamente la nozione stessa di file e tutte le operazioni di input-output sono su canale. In Cobol l'unico concetto analogo all'input-output su canale è fornito dalla coppia di verbi ACCEPT/DISPLAY, il cui uso è però molto limitato. Tutti i file sono costituiti da record, ma alcune operazioni di lettura-scrittura possono implicare delle trasformazioni di rappresentazione.
Nel caso speciale di input-output su canale, il PL/1 ed alcune versioni del Fortran fanno un'ulteriore distinzione, che riguarda il modo in cui è specificata, nel programma, la forma che devono avere i dati stampati. In PL/1 esistono tre diversi modi di trasmissione, con le tre corrispondenti varianti associate alle istruzioni GET o PUT: una trasmissione formattata (GET EDIT o PUT EDIT) dove, ricorrendo ad un descrittore di formato, si specifica la rappresentazione esatta dei valori sul supporto esterno; una trasmissione a lista (GET LIST o PUT LIST) dove questa rappresentazione è determinata dai dati in input e da valori di default in output; una trasmissione per nome (GET DATA o PUT DATA), che è una trasmissione a lista, in cui ogni dato è preceduto dal nome della variabile a cui verrà assegnato (in input) o a cui è stato assegnato (in output).
Il Fortran standard prevede, come unico meccanismo, l'input-output formattato, che molto spesso è poco flessibile e porta facilmente a commettere errori soprattutto in input. Alcune versioni consentono anche una specie di trasmissione per nome con l'istruzione NAMELIST. Il BASIC consente solo trasmissioni non formattate ed il Cobol è, per molti aspetti, equivalente all'input-output formattato.
In Pascal esiste, per l'input, solo la trasmissione non formattata. In output è possibile avere sia la trasmissione formattata che quella non formattata, ma in un modo più semplice di quanto si trovi negli altri linguaggi, dal momento che il formato dipende soprattutto dal tipo degli oggetti trasmessi e di conseguenza non è necessario rispecificarlo nelle istruzioni di input-output. Gli esempi mostreranno come sfruttare gli effetti di tutti i modi di trasmissione, generalmente in modo semplice e flessibile.

8.2 FILE DI TIPO *text*

Il tipo predefinito *text* non è equivalente al tipo **file of** *char*: gli elementi di un textfile sono caratteri, ma in più un file di tipo *text* è organizzato su *linee*. Una linea è una sequenza di caratteri, anche vuota, chiusa da un elemento speciale, l'*end-of-line*. Non è possibile distinguere questo carattere dallo spazio, se non con le quattro procedure predefinite *reset*, *writeln*, *readln* e *page* e per la funzione predefinita *eoln*. Ciò significa che non è possibile riconoscere o scrivere un end-of-line con le normali procedure predefinite *get* (o *read*) e *put* (o *write*). Si noti che nulla è detto sulla rappresentazione interna delle linee: in alcune implementazioni è previsto un carattere speciale al termine di ciascuna linea; in altre i caratteri possono essere più di uno, o può esserci un contatore di caratteri all'inizio di ogni linea, o qualsiasi altra cosa. Il punto importante è che queste differenze di implementazione sono del tutto trasparenti all'utente.
Un file di tipo *text* è una sequenza (anche vuota) di linee. Durante la fase di generazione, l'ultima linea può non essere completa, se non si è ancora generato il corrispondente simbolo di end-of-line. Tutte le funzioni o le procedure predefinite sui file operano su un file di tipo *text* come se questo fosse un **file of** *char*, ma in questo caso il simbolo di end-of-line non è riconoscibile in input e non può essere generato in output.
La funzione predefinita *eoln* (*end-of-line*) viene usata per rilevare un end-of-line in input. Se *ft* è un file di tipo *text*, *eoln*(*ft*) vale *true* se e solo se la posizione corrente di *ft* coincide con la fine della riga. Quando *eoln*(*ft*) vale *true*, *ft*↑ contiene uno spazio. Tutto questo è espresso dalla seguente regola di verifica:

se ***modo di accesso*** a *ft* = generazione
allora ***firstof parte destra*** di *ft* = end-of-line→*eoln*(*ft*) ∧ *ft*↑ = {spazio}

ESEMPIO 8.1: CONTEGGIO DI LINEE IN UN FILE DI TIPO TEXT

```
procedure ContaLinee(var ft: text; var numero: integer)
   {questa procedura conta il numero di linee in un file ft di tipo text};
begin {ContaLinee}
   reset(ft); numero := 0;
   while not eof(ft) do
   begin {la variabile numero conta le linee nella parte sinistra di ft}
   while not eoln(ft) do get(ft) {avanza di un carattere};
   {è stata esaminata un'altra linea}
   numero := numero + 1;
   get(ft) {supera l'end-of-line}
   end {while ¬eof(ft)}
end {ContaLinee}
```

COMMENTI

1. Avremmo potuto definire questa procedura come una funzione, il cui valore è il numero delle linee del file di tipo *text* specificato come parametro: avremmo però dato un brutto esempio di programmazione, per l'effetto collaterale, notevole e costoso che avrebbe comportato. Un *reset* (*ft*) alla fine della funzione avrebbe eliminato l'effetto collaterale, ma l'alto costo sarebbe rimasto. Anche nel seguito non definiremo mai una funzione al cui interno si facciano delle operazioni di input-output.
2. Come si può osservare da questo esempio, *eof*(*ft*) può assumere il valore *true* solo dopo *reset*(*ft*) (con un file vuoto) o dopo *get*(*ft*) chiamata quando *eoln* (*ft*) diventa *true*. Questa è una conseguenza del fatto che tutte le linee di un file di tipo *text*, in ispezione, sono complete, compresa l'ultima. Qualsiasi implementazione in cui questa asserzione è falsa non è un'implementazione del Pascal Standard.

I file di tipo *text* sono spesso usati per la comunicazione interattiva fra programma e utente. Il terminale interattivo è concettualmente composto da una coppia di file che si appoggiano allo stesso supporto fisico, uno per l'input, l'altro per l'output. In questo caso è necessario leggere attentamente le asserzioni introdotte per le procedure *reset* e *get*. Lo Standard ISO afferma chiaramente che i conseguenti di queste due procedure non devono essere soddisfatti subito dopo la chiamata, ma immediatamente prima del riferimento successivo al file o al buffer ad esso associato. Così, la lettura fisica del carattere successivo non verrà fatta alla chiamata di *reset*(*ft*) o di *get*(*ft*), ma solo quando si fa riferimento a *ft*↑ o quando si effettua una nuova chiamata a *get*(*ft*).
È necessario procedere in questo modo per garantire un funzionamento corretto dell'input-output interattivo. Per esempio, quando si fa un *reset* del file di input da terminale, il programma non deve mettersi subito in attesa di un dato senza avere prima stampato un messaggio di richiesta. Se si dovesse soddisfare il conseguente di *get* subito dopo la sua chiamata, ci sarebbe una fastidiosa discrepanza fra l'input e l'output su terminale e l'utente dovrebbe digitare qualche cosa prima ancora di sapere cosa. Sfortunatamente ciò è quanto accade in alcune implementazioni del Pascal, che non fanno alcuna distinzione fra file ordinari e file interattivi.
Nella maggior parte dei sistemi operativi, un programma può accedere a due file di particolare importanza, generalmente noti come *file standard*. Il file standard di input solitamente ha lo stesso supporto fisico del testo del programma ed è quella parte del flusso di input che segue il testo del programma, da cui può essere separato, in qualche caso, da comandi di controllo per il sistema operativo. Il file standard di output è quello su cui viene listata l'uscita del compilatore, insieme ad eventuali messaggi diagnostici emessi dal sistema operativo.
Per questi file di tipo *text* privilegiati esistono in Pascal due nomi predefiniti: *input* e *output*. Questi nomi sono standard perché i file corrispondenti godono,

in generale, di una particolare proprietà: *input* può essere solo ispezionato e *output* può essere solo generato; nessuno dei due può essere né riletto né riscritto.
Per ragioni prettamente storiche questi due file sono trattati in Pascal in un modo molto particolare, descritto dalle seguenti regole:

1. Sono definiti in tutti i programmi che ne fanno uso e non è perciò necessario dichiararli.
2. Come per ogni altro file esterno al programma, devono comparire nell'intestazione del programma in cui vengono usati.
3. Se si usa il file *input*, prima dell'inizio del programma viene fatta in modo automatico una chiamata a *reset* (*input*). In modo analogo, se si usa *output*, viene fatta automaticamente, prima dell'inizio del programma, una chiamata a *rewrite* (*output*).
4. Le chiamate a *reset* o *rewrite* esplicite su questi file predefiniti di tipo *text* hanno effetti che dipendono dall'implementazione (sono in generale vietate).
5. Se si tralascia il primo (o il solo) parametro delle procedure o funzioni predefinite *read*, *readln*, *eof* e *eoln* (insieme al relativo delimitatore), si richiede implicitamente il file predefinito *input*. Analogamente, tralasciando il primo (o l'unico) parametro delle procedure predefinite *write* e *writeln* si richiederà implicitamente il file predefinito *output*.

Le regole 1, 2 e 3 non possono essere ignorate. La regola 4 implica che un programma, che non debba dipendere da una particolare implementazione, non deve fare uso delle procedure di *reset* e di *rewrite* sui file predefiniti di tipo *text*. La regola 5, invece, fornisce solo una scorciatoia e può essere del tutto ignorata, cosa che faremo quasi sempre, dal momento che questa abbreviazione non aggiunge alcuna generalità al linguaggio, né ne aumenta la potenza espressiva ma, al contrario, può portare a scrivere programmi meno chiari.
Alcune implementazioni del Pascal possono imporre altre restrizioni non standard all'uso dei textfile predefiniti. In alcuni casi *output* deve comparire nell'intestazione del programma anche se non sarà usato al suo interno. Perciò si raccomanda caldamente di studiare attentamente la documentazione allegata ad ogni compilatore.

8.3 INPUT DA FILE DI TIPO *text*

La procedura predefinita *read*, introdotta nel Capitolo 7 semplicemente come un modo per abbreviare la quantità di codice da scrivere, si può applicare anche ai file di tipo *text* ed assume un significato speciale come strumento per leggere dal file più di un carattere per volta e per interpretare la rappresentazione di dati numerici. La sintassi dei parametri per *read* è descritta dal diagramma riportato in Figura 8.1.

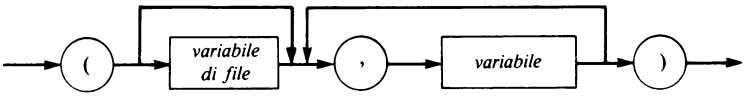


Figura 8.1 Diagramma sintattico della lista di parametri di *read*

Nel caso manchi la variabile di tipo file, si suppone, per default, che essa sia il file standard *input*. La variabile può essere sia una variabile semplice che un elemento (semplice) di una variabile strutturata. Una chiamata a *read* con più variabili equivale a molte chiamate alla stessa procedura, specificando come parametri lo stesso nome del file ed una sola variabile per volta. Tutte queste regole, come si è già visto nel Capitolo 7, sono valide per qualsiasi parametro di un sottoprogramma che operi sui file, e non sono limitate ai soli file di tipo *text*.
Nel caso si operi sui file ordinari, la variabile deve avere lo stesso tipo degli elementi del file, o un tipo compatibile in un'istruzione di assegnamento, poiché

read(*f*, *v*)

equivale a

begin *v* := *f*↑; *get*(*f*) **end**

Nel caso di parametri di textfile, tuttavia, sono consentiti tre diversi tipi per la variabile *v*:

1. Se *v* è di tipo *char*, o di un sottocampo di *char*, si è nella stessa situazione di prima; l'unica differenza fra un textfile ed un **file of** *char* è che *eoln*(*f*) può essere modificato a causa di un effetto collaterale.
2. Se *v* è di tipo *integer*, o di un suo intervallo, la chiamata a *read*(*f*, *v*) legge da *f* una sequenza di caratteri che deve formare una costante intera (con o senza segno) secondo la sintassi data nel Capitolo 1. Tutti gli spazi posti prima del primo carattere significativo (e, di conseguenza, tutti gli end-of-line) vengono saltati. La lettura termina quando il carattere corrente (il cui valore è in *f*↑) non può far parte di un intero. In altre parole, *read*(*f*, *v*) legge, in successione, degli spazi (anche nessuno), un segno facoltativo, e delle cifre decimali, fino a che non trova un carattere diverso da una cifra. Durante questa lettura si possono verificare quattro diverse condizioni di errore:
   a) si incontra un end-of-line prima di aver letto una qualsiasi cifra;
   b) non si è ancora letta una cifra quando si incontra un carattere che termina la lettura;
   c) l'intero così costruito non cade all'interno dei limiti imposti dalla definizione;
   d) l'intero così costruito non è compatibile, in un'istruzione di assegnamento, con *v*.

Come sempre accade in Pascal, ciascuno di questi errori forza la terminazione del programma.

3. Se *v* è di tipo *real*, tutto funziona come detto al punto 2, ma la sequenza di caratteri letta deve rappresentare un numero (con o senza segno); può cioè essere una costante intera o reale e l'errore (d) non si può mai verificare.

Per file di tipo *text* esiste un'altra funzione predefinita, *readln*. Una chiamata a *readln* con più parametri equivale ad una chiamata a *read*, con gli stessi parametri, seguita da una chiamata a *readln* senza parametri. Dal momento che il parametro con il nome del file può essere omesso, *readln* può essere chiamata senza parametri e ciò equivale a

readln(*input*)

La chiamata *readln*(*f*) ha semplicemente l'effetto di saltare tutti i caratteri dalla posizione corrente alla fine della riga e di posizionare *f* all'inizio della riga successiva, se tale riga esiste. (Si noti che, se la linea successiva è vuota, il suo inizio e la sua fine coincidono, *f*↑ è uno spazio e *eoln*(*f*) vale *true*). Così *readln*(*f*) equivale a

```
begin
   while not eoln (f) do get (f);
   get (f) {supera l'end-of-line}
end
```

La precedente discussione sull'input da file di tipo *text* sembra semplice, ma occorre mettere in evidenza alcuni dettagli che possono facilmente sfuggire e far risaltare alcuni difetti. La caratteristica principale dell'input di testi in Pascal è probabilmente la mancanza assoluta di qualsiasi formato che descrive i dati attesi. Così la lettura e la decodifica di un carattere è guidata sia dal tipo di dato atteso (il tipo dei parametri di una procedura *read*), che dal dato stesso. I dati numerici devono essere separati da caratteri non numerici e si possono verificare due diverse situazioni: se si devono leggere molti numeri in sequenza, questi possono essere separati da spazi o da simboli di end-of-line; una sequenza di spazi termina il numero precedente e viene saltata quando si legge il numero successivo. Se invece si legge un solo numero, il suo carattere terminatore è presente nel buffer del file e può essere esaminato — trattandolo come un carattere — qualunque esso sia.
Non si è fatto cenno, nella precedente discussione, alla lunghezza delle linee, che di fatto non ha alcuna importanza. Quando si legge dal file standard *input*, non si può fare l'ipotesi che le linee siano tutte della stessa lunghezza ed infatti la maggior parte delle implementazioni Pascal taglia le linee in input, lasciandone solo la parte significativa e rimuovendo tutti gli spazi che la seguono. Dopo avere letto uno o più numeri, è difficile predire lo stato di *eoln*, che dipende dalla presenza o meno di ulteriori spazi. Non si può così garantire che il sem-

plice ciclo, proposto qui sotto, funzioni in modo corretto:

```
while not eoln(input) do
begin read(input, n); write(output, n) end
```

Dal momento che non c'è alcuna ragione perché *eoln*(*input*) sia *true* immediatamente dopo la *read*(*input, n*), la condizione di terminazione può non verificarsi prima di raggiungere la fine del file, la qual cosa provoca un errore.
Ci sembra quindi utile suggerire di non fare alcun controllo sul valore di *eoln* quando si usano le forme estese di *read*. Allo stesso modo non si può usare *eof*, che assume il valore *true* solo dopo che *eoln* è diventato *true* a sua volta e, come si è visto, non si può garantire che *eoln* valga *true* immediatamente dopo una *read*. Infatti, se si incontra un *eof* mentre la *read* sta leggendo un numero, si verifica un errore che termina l'esecuzione del programma. La chiamata ad una *read* con più dati numerici letti contemporaneamente è particolarmente pericolosa; se si incontra un *eof* prima di aver letto tutti i numeri richiesti, il programma termina la sua esecuzione. Per leggere e copiare più numeri — separati da spazi — fino alla fine del file, occorre usare il seguente ciclo:

```
while not eof(input) do
if input↑ = ' ' then get(input)
else begin read (input, n); write(output, n) end
```

Esiste inoltre un'altra ragione che riduce l'utilità delle forme estese della *read* solo a quei pochi casi in cui si può garantire l'esatto formato dei dati in input: se il numero da leggere contiene un qualsiasi errore — per esempio un errore di perforazione o la mancanza di un separatore — il programma non può riprendere il controllo e la sua esecuzione termina. Questo fatto non è accettabile in molte situazioni — per esempio quando si stanno leggendo dei dati introdotti da un terminale interattivo. In questo caso il programma deve definirsi delle proprie routine di lettura, con qualche meccanismo per la gestione degli errori sui dati introdotti (si veda l'Esempio 8.2). Alcune implementazioni offrono degli strumenti non standard per trattare gli errori che si verificano a tempo di esecuzione.
Infine bisogna mettere in evidenza che il Pascal non ha alcuno strumento per leggere valori di un tipo scalare o stringhe di caratteri. In particolare non si può leggere direttamente un valore di tipo booleano. La ragione principale per cui non è stato implementato l'input-output dei tipi scalari è che i relativi identificatori sono esclusivamente locali al programma e non hanno alcun significato al di fuori di esso. Per di più, il trattamento degli errori sul valore dei dati sarebbe ancora più primitivo. Per quanto riguarda le stringhe di caratteri, il concetto in sé non è una caratteristica del Pascal ed è molto semplice scrivere una procedura di *read* per il tipo di dati equivalente (si veda il Capitolo 9).

ESEMPIO 8.2: LETTURA DI INTERI IN UN FILE DI TIPO TEXT

```
procedure LetturaInteri
      (var ft: text {il textfile di input};
      var risultato: integer {il numero letto su ft};
      var errore: Boolean {true se viene letto un numero non corretto};
         lunghezza: integer {numero di caratteri da leggere; se il formato è libero
         vale zero}
      );
   const spazio = ' ';
      base = 10 {si leggono numeri decimali};
   var finecampo, segno, cifrelette: Boolean {flag};
      nch: integer {numero di caratteri letti fino a quel punto};
   procedure ProssimoCarattere;
      {passa al carattere successivo, se possibile; assegna un valore a errore, nch e
         finecampo}
   begin {ProssimoCarattere}
      if eof(ft) or finecampo then errore := true
      else begin nch := nch+1; get(ft);
         if nch = lunghezza then finecampo := true
      end
   end {ProssimoCarattere}
begin {LetturaInteri}
   error := false; finecampo := false; nch := 0;
   risultato := 0;
   while (ft↑ = spazio) and not errore do
      {salta gli spazi che precedono un numero}
      ProssimoCarattere;
   if not errore then
   begin segno := false; {elabora l'eventuale segno}
      if ft↑ = '+' then ProssimoCarattere
      else if ft↑ = '-' then
         begin segno := true; ProssimoCarattere end;
      if not errore then
      begin {elabora il numero vero e proprio}
         cifrelette := false;
         while (ft↑ ≥ '0') and (ft ≤ '9') and not (errore or finecampo)
         do
         begin cifrelette := true;
            risultato := base * risultato+ord(ft↑)-ord('0');
            ProssimoCarattere
         end;
         if not cifrelette or (lunghezza≠0) and not finecampo then
            errore := true
         else if segno then risultato := -risultato
```

```
        end {elaborazione del numero vero e proprio}
    end {elaborazione dopo aver saltato gli spazi}
end {LetturaInteri}
```

COMMENTI

1. Questa procedura è piuttosto complicata, perché deve leggere sia numeri in formato libero che numeri in formato fisso, ed anche perché non deve segnalare eventuali errori, per dati errati, una volta in esecuzione. Sfortunatamente non può avere un comportamento *fault-tolerant* neppure in presenza di un errore di programmazione, dal momento che il programma termina comunque la sua esecuzione se la procedura viene chiamata senza che il file *ft* sia stato aperto in ispezione. Ciò accade perché il modo di accesso ad un file non è accessibile all'interno del programma.
2. Naturalmente, in presenza di dati errati, quando la procedura assegna alla variabile *errore* il valore *true*, la responsabilità di gestire la situazione ricade sul programma chiamante, che può per esempio saltare tutti i caratteri illegali fino a che sia di nuovo possibile richiamare la procedura.

8.4 OUTPUT SU FILE DI TIPO *text*

Anche la procedura predefinita *write* opera, in modo del tutto analogo alla *read*, su file di tipo *text*, per scrivere più di un valore per volta, per codificare i valori numerici che si devono scrivere, e per descrivere alcuni attributi della codifica stessa. Il diagramma sintattico della lista dei parametri di *write* (applicata ad un file di tipo *text*) è mostrato in Figura 8.2.
Se manca la variabile di tipo file, il file di uscita sarà implicitamente il file predefinito *output*. Una chiamata a *write* con più di un parametro equivale a molte chiamate con un solo parametro per volta e tutte con lo stesso identificatore di file. Esiste inoltre una differenza importante con quanto visto, nel Capitolo 7, per i parametri della procedura *write* che, quando è applicata a file di tipo *text*, possono avere tre forme:

espressione
espressione : *dimensione-totale*
espressione : *dimensione-totale*: *cifre-decimali*

espressione è il valore che si deve scrivere sul file di tipo *text*; *dimensione-totale* e *cifre-decimali* sono espressioni intere che determinano con esattezza il formato di output del valore, nel modo seguente:

1. *dimensione-totale* e *cifre-decimali*, se presenti, devono essere maggiori di zero.

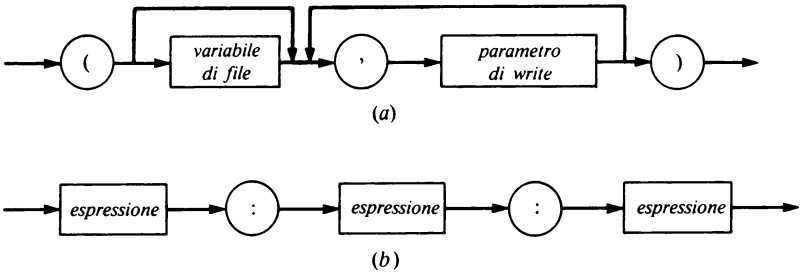


Figura 8.2 Diagramma sintattico della lista dei parametri di *write*: (*a*) lista dei parametri di *write*; (*b*) parametri di *write*

2. *dimensione-totale* è il minimo numero di caratteri che si devono scrivere; se il valore di *espressione* richiede più cifre per la sua rappresentazione, si sceglie la più piccola dimensione necessaria, in funzione del valore; se basta un minor numero di caratteri, il numero viene allineato a destra facendolo precedere da un adeguato numero di spazi.
3. Se non si specifica il valore della *dimensione-totale*, viene usato un valore di default che dipende sia dal tipo dell'espressione che dall'implementazione.
4. Il campo *cifre-decimali* ha senso solo se l'espressione è di tipo *real*.

L'*espressione* può essere di uno di cinque tipi diversi, ciascuno dei quali ha regole di rappresentazione diverse.

8.4.1 OUTPUT DI CARATTERI

Il valore di default della *dimensione-totale* è 1. Di conseguenza, si scrivono un numero di spazi pari a (*dimensione-totale* − *1*), seguiti dal carattere corrispondente al valore dell'espressione. Il numero di spazi che precedono il carattere può anche essere zero.

8.4.2 OUTPUT DI INTERI

Il valore di default della *dimensione-totale* è, generalmente, funzione dell'implementazione e di solito è pari alla lunghezza necessaria a scrivere il maggior intero possibile. Se necessario, vengono aggiunti degli spazi a sinistra per allineare a destra il numero intero stesso. I valori negativi sono preceduti dal segno meno.

8.4.3 OUTPUT DI REALI

Sono possibili due diverse rappresentazioni per i valori reali: in virgola mobile, se non si specifica il numero delle cifre decimali ed in virgola fissa, in caso contrario. Queste rappresentazioni corrispondono rispettivamente ai formati E ed F del Fortran o del PL/1.
In una rappresentazione in virgola mobile la successione di caratteri scritti è la seguente:

a) spazi, in numero sufficiente per avere un numero totale di caratteri pari a quanto specificato da *dimensione-totale*;
b) un segno meno se il numero è negativo, altrimenti uno spazio;
c) la cifra più significativa della rappresentazione decimale del numero reale;
d) un punto (equivalente alla virgola decimale);
e) le cifre per la parte decimale: una sola se *dimensione-totale* è troppo piccola, altrimenti un numero di cifre pari a (*dimensione-totale* − *cifre-dell'esponente* − 5), dove *cifre-dell'esponente* è il numero di cifre (dipendente dall'implementazione), previste per l'esponente (di solito 2 o 3);
f) il carattere per l'esponente ('e' o 'E', in funzione dell'implementazione);
g) un numero di cifre, per l'esponente, pari al valore di *cifre-dell'esponente*, precedute, se del caso, da spazi.

Nella rappresentazione in virgola fissa, si deve invece scrivere la seguente successione di caratteri:

a) un numero di spazi sufficiente ad avere un numero di caratteri pari a *dimensione-totale*;
b) un segno meno, se il numero è negativo;
c) la parte intera della rappresentazione, in base dieci, del numero reale;
d) un punto di separazione per la parte decimale;
e) le prime cifre significative della parte decimale.

8.4.4 OUTPUT DI VALORI BOOLEANI

Se l'espressione è di tipo *Boolean*, allora l'istruzione

write (*ft, espressione*: *wt*)

equivale a

if *espressione* **then** *write* (*ft, 'true'* : *wt*)
else *write* (*ft, 'false'* : *wt*)

La stringa di caratteri viene scritta in maiuscolo o in minuscolo in funzione dell'implementazione. Anche il valore di default della *dimensione-totale* dipende dall'implementazione.

8.4.5 OUTPUT DI STRINGHE

Anche se le stringhe di caratteri verranno trattate solo nel Capitolo 9, per ragioni di completezza descriveremo ora il comportamento della procedura *write* quando un suo parametro è una stringa di caratteri. Se la *dimensione-totale* è maggiore della lunghezza della stringa, la stringa stessa viene allineata a destra mediante l'aggiunta di spazi. Se invece è minore, vengono scritti solo i primi caratteri, fino ad esaurimento dello spazio previsto da *dimensione-totale*. Questo è l'unico caso in cui lo spazio previsto non viene allargato per contenere l'intero valore da scrivere. Il valore di default della *dimensione-totale* è la lunghezza della stringa.
La funzione predefinita *writeln* può operare solo su file di tipo *text*. Una chiamata a *writeln* con più parametri equivale ad una chiamata alla procedura *write*, con gli stessi parametri, seguita da una chiamata a *writeln* senza alcun parametro. Come la *readln*, anche la *writeln* può essere chiamata senza alcun parametro, ed in questo caso opera sul file predefinito *output*. La chiamata *writeln*(*ft*) ha come unico effetto quello di appendere un end-of-line al file di tipo *text*, *ft*, per terminare la linea corrente.
La funzione predefinita *page* ha un solo parametro di tipo *text*. Il suo effetto dipende dall'implementazione, ed il suo risultato è di cominciare una nuova pagina tutte le volte che il file è inviato ad un dispositivo su cui abbia senso parlare di pagine. Se la linea corrente non è completa, la procedura *page* esegue una chiamata implicita a *writeln* sul file su cui è chiamata ad operare. Dal momento che l'effetto della *page* dipende dall'implementazione, non è possibile predire cosa succederà quando il file verrà successivamente ispezionato; potranno esserci dei caratteri invisibili, può non succedere assolutamente niente, o può anche essere segnalato un errore.
Alla chiamata della procedura *reset* su di un file di tipo *text* aperto in fase di generazione, occorre appendere un end-of-file all'ultima linea, nel caso questa non sia completa. Quest'operazione è necessaria per garantire che tutti i file di tipo *text*, in ispezione, contengano esclusivamente linee complete. La situazione è analoga quando si vuole stampare un file di tipo *text* dopo la terminazione di un programma: in questo caso non occorre mettere una chiamata finale a *writeln* al termine del programma.
La descrizione precedente sulle procedure predefinite è in perfetta sintonia con lo Standard ISO. Tuttavia alcune implementazioni Pascal meno recenti si scostano da quanto detto: non prevedendo un *writeln* implicito per i file di tipo *text*, danno un significato speciale al primo carattere di ciascuna linea stampata, il cui scopo non è quello di venire stampato, ma di controllare la stampante (come in Fortran); non definiscono la procedura predefinita *page*. Si può solo sperare che queste varianti, prima o poi, scompaiano.
Per un programmatore abituato alle complessità del Fortran o del PL/1, le possibilità offerte dal Pascal per l'output di testi potranno sembrare troppo limitate, e probabilmente non abbastanza potenti neppure per le normali applicazioni. Di fatto bastano per descrivere il più complicato dei format di output,

con la sola differenza, rispetto agli altri due linguaggi, che quanto è già precostituito in Fortran ed in PL/1, in Pascal deve essere programmato. Quest'operazione può essere fatta molto facilmente, perché non si è costretti a preparare completamente una riga, prima di scriverla, dal momento che questa può essere scritta passo passo. Nei capitoli successivi verranno presentati esempi di preparazione di forme complesse di output.

8.5 ESEMPI COMPLETI

ESEMPIO 8.3: COPIA DI UN FILE DI TIPO TEXT

```
procedure CopiaTesto (var infile, outfile: text)
   {copia infile in outfile conservandone la struttura a linee};
begin {CopiaTesto}
   reset(infile); rewrite(outfile);
   while not eof(infile) do
   begin {siamo all'inizio di una linea}
      while not eoln(infile) do
      begin {elabora un carattere normale}
         outfile↑ := infile↑;
         get(infile); put(outfile)
      end {fine di una linea (eventualmente vuota)};
      writeln(outfile); readln(infile)
   end {fine di infile}
end {CopiaTesto}
```

COMMENTI

1. La procedura *CopiaFile*, presentata nell'Esempio 7.1, non funziona per file di tipo *text*; la *get* infatti non riconosce il simbolo di end-of-line (che la procedura *put* non può scrivere), per cui non lo si sarebbe potuto trascrivere sul file di output.
2. Usata con i file predefiniti *input* e *output*, come parametri attuali, la procedura *CopiaTesto* copierebbe il file standard *input* nel file standard *output* solo nel caso in cui la particolare implementazione non effettui alcuna operazione in corrispondenza delle chiamate *reset* (*input*) e *rewrite* (*output*). Alcune implementazioni potrebbero trattare queste chiamate come degli errori, mentre altre potrebbero interpretarle in tutt'altro modo.

ESEMPIO 8.4: COPIA DI UN FILE DI TIPO TEXT CON CONTROLLO DI LINEA

```
procedure CopiaPagine (var infile, outfile: text)
   {poiché outfile sarà stampato, occorre numerare le pagine};
   const lunghpagina = ... {numero di linee in una pagina};
```

```
        lunghpaginapiu1 = ... {lunghpagina + 1};
    var contalinea: 1..lunghpaginapiu1;
        contapagina: 0..maxint;
begin {CopiaPagine}
    reset(infile): rewrite(outfile);
    contapagina := 0;
    contalinea := lunghpagina {per forzare un salto pagina iniziale};
    while not eof(infile) do
    begin {siamo all'inizio di una linea}
        contalinea := contalinea + 1 {conta la linea da stampare};
        if contalinea > lunghpagina then
        begin {siamo all'inizio di una pagina}
            page(outfile);
            contapagina := contapagina + 1;
            writeln(outfile, 'pagina', contapagina: 1)
                {campo minimo per contapagina, che sarà stampato allineato a
                sinistra};
            writeln(output); writeln(output) {stampa due linee vuote};
            contalinea := 1
        end {intestazione della pagina};
        while not eoln(infile) do
        begin {elabora un carattere ordinario}
            read(infile, outfile↑) {variante dell'Esempio 8.3};
            put(outfile)
        end {fine di una linea};
        writeln(outfile); readln(infile);
    end {fine di infile}
end {CopiaPagine}
```

COMMENTI

Un confronto fra gli Esempi 8.3 e 8.4 mostra che, aggiungendo delle opportune azioni all'algoritmo descritto nel primo esempio, è molto facile programmare qualsiasi trasformazione su file di tipo *text*. Questo schema è illustrato qui sotto in forma astratta:

```
while not eof(infile) do
begin < azione all'inizio di una linea >;
    while not eoln(infile) do
    begin < azione su di un carattere >;
        get(infile)
    end;
    < azione alla fine di una linea >;
    readln(infile)
end
```

ESEMPIO 8.5: AGGIORNAMENTO SEQUENZIALE DI UN FILE DI TIPO TEXT

```
program AggiornaTextFile(input, output, infile, outfile)
  {questo programma aggiorna il file infile di tipo text, in funzione dei comandi ricevuti e produce outfile; vengono anche fornite informazioni sull'esito delle operazioni e messaggi di errore; vengono riconosciuti quattro comandi di aggiornamento, della forma <lettera> <spazi> <intero> <end-of-line>:
     C n copia linee da infile a outfile fino alla linea n di infile;
     D n cancella linee da infile (non le ricopia) fino alla linea n;
     I n copia n linee da terminale (immediatamente seguenti il comando) a outfile;
     E (senza parametri) termina l'elaborazione copiando tutte le righe rimanenti da infile a outfile};
type codiceerrore = 1..5;
var infile, outfile: text;
  comando: char;
  operando, numlinea, contatore: 0..maxint;
  finenormale, incontratoerrore: Boolean;
procedure CopiaLinea (var filesorgente: text)
  {la destinazione è sempre outfile};
begin {CopiaLinea}
  while not eoln(filesorgente) do
  begin outfile↑ := filesorgente↑;
    get(filesorgente); put(outfile)
  end {fine della linea};
  readln(filesorgente); writeln(outfile)
end {CopiaLinea};
procedure MessaggioDiErrore(errore: numeroerrore);
begin {MessaggioDiErrore}
  writeln(output); writeln('output');
  case error of
      1: writeln(output, 'operando < numlinea');
      2: writeln(output, 'fine prematura del file');
      3: writeln(output, 'fine prematura dell'input');
      4: writeln(output, 'comando illegale');
      5: writeln(output, 'manca il comando E')
  end;
  incontratoerrore := true
end {MessaggioDiErrore};
begin {programma AggiornaTextFile}
  reset(infile); rewrite(outfile); numlinea := 1;
  finenormale := false; incontratoerrore := false;
  while not (eof(input) or finenormale) do
  begin {elabora un comando}
```

```
    read(input, comando); write (output, comando);
    if (comando = 'I') or (comando = 'C') or (comando = 'D') then
    begin {comando ordinario}
      readln(input, operando);
      writeln(output, '', operando: 1);
      case comando of
        'C', 'D':
          begin {copia o cancella fino alla linea 'operando'}
            if operando < numlinea then
              MessaggioDiErrore(1)
            else begin
              while not eof(infile) and (numlinea < operando) do
              begin {processa una linea da infile}
                if comando = 'C' then
                  CopiaLinea(infile)
                else readln(infile);
                numlinea := numlinea + 1
              end {fine di copia o cancella};
              if numlinea < operando then
                MessaggioDiErrore(2)
            end {non c'è errore 1}
          end {comando 'C' o 'D'};
        'I':
          begin {inserisce un numero di linee pari a 'operando'}
            contatore := 0;
            while not eof(input) and (contatore < operando) do
            begin {copia una linea dall'input}
              CopiaLinea(input); contatore := contatore + 1
            end {fine dell'inserimento};
            if contatore < operando then
              MessaggioDiErrore(3)
          end {comando 'I'}
    end {comando ordinario}
    else if comando = 'E' then
    begin {fine aggiornamento}
      while not eof(infile) do CopiaLinea(infile);
      finenormale := true; readln(input);
      writeln(output)
    end
    else begin MessaggioDiErrore(4); readln(input) end
  end {elaborazione di tutti i comandi};
  if not finenormale then MessaggioDiErrore(5);
  if incontratoerrore then
    writeln(output, '***Errori nell'aggiornamento precedente***')
end {AggiornaTextFile}.
```

COMMENTI

1. La difficoltà maggiore, nell'esempio precedente, è data dall'esatta gestione degli end-of-line nei file standard *input* e *output*.
2. Questo programma non è perfettamente *fault-tolerant*, dal momento che un qualsiasi carattere illegale, dato dopo un comando, ne provoca la terminazione. Per evitare questo comportamento, occorrerebbe sostituire la chiamata di *read* all'inizio dell'elaborazione di un comando ordinario, con una versione leggermente modificata della procedura *LetturaInteri* (Esempio 8.2). Questa procedura dovrebbe trattare anche operandi negativi.
3. Se i file predefiniti *input* e *output* fossero associati ad un terminale interattivo, sarebbe probabilmente superfluo inviare un eco di quel che viene letto, dal momento che ciò verrebbe già fatto direttamente dal terminale o dal sistema operativo.

ESERCIZI

8.1 Lettura di un record

Un file di tipo *text* contiene dei record con il seguente formato:

ca	(spazio)	codice	(spazio)	identificatore	(spazio)	prezzo	(spazio)	iva	eoln
1	1	5	1	20	1	6	1	4	4

che corrispondono al seguente FORMAT Fortran:

```
I1,1X,I5,1X,20A1,1X,F6.2,1X,F4.2
```

ed alla seguente descrizione Cobol:

```
01 TEXTFILE-RECORD
  02 CA PICTURE 9.
  02 FILLER PICTURE X.
  02 COD PICTURE 9(5).
  02 FILLER PICTURE X.
  02 IDENTIFICATION PICTURE A(20).
  02 FILLER PICTURE X.
  02 PRICE PICTURE 9(4)V99.
  02 FILLER PICTURE X.
  02 TAX PICTURE 9(2)V99.
```

Si scriva una procedura Pascal che scrive un record di questo formato con la seguente intestazione:

```
procedure LeggeUnRecord (var ca: integer; var codice: tipocodice; var identificatore:
        stringa20; var prezzo, iva: real);
```

Il *tipocodice* è un sottocampo del tipo *integer*. Sul tipo strutturato *stringa20* sono definiti tre operatori:

a) *Inizializza* (**var** *s*: *stringa20*) prepara *s* ad un assegnamento;
b) *Appende* (**var** *s*: *stringa20*; *c*: *char*) appende *c* ad *s*
c) *Chiude* (**var** *s*: *stringa20*) termina quello che era stato inizializzato con il primo operatore.

8.2 Lettura di un record II

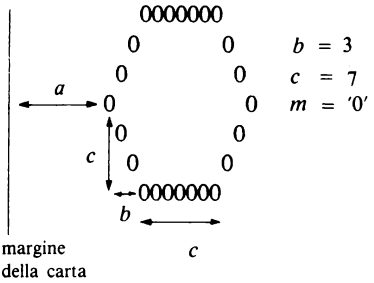
Questo esercizio è identico al precedente, ma questa volta mancano gli spazi di separazione. Ciò corrisponde al FORMAT Fortran

I1,I5,20A1,F6.2,F4.2

o alla descrizione Cobol priva dei campi FILLER.

8.3 Progetto di un formato

Dati i tre parametri interi *a*, *b*, *c*, che rappresentano delle lunghezze espresse in numero di caratteri, ed un carattere campione *m*, si scriva una procedura che stampi il seguente disegno:



8.4 Impaginatore

Si scriva un insieme di procedure per generare delle relazioni impaginate secondo il formato mostrato in Figura 8.3. Si suggerisce di usare le seguenti procedure:

procedure *LineaDiCroci*(*lunghezzatotale*: *lunghezza*)
procedure *IntestazioneRelazione*(*intestazione*: *stringa20*; *larghezzausata*: *stringa20lunghezza*; *lunghezzatesto*: *lunghezza*)
procedure *IntestazioneColonna*(*intestazione*: *stringa20*; *larghezzausata*: *stringa20lunghezza*; *lunghezzacolonna*: *lunghezza*)
procedure *StampaReale*(*numero*: *real*; *lunghezzafrazione*: *integer*; *lunghezzacolonna*: *lunghezza*)

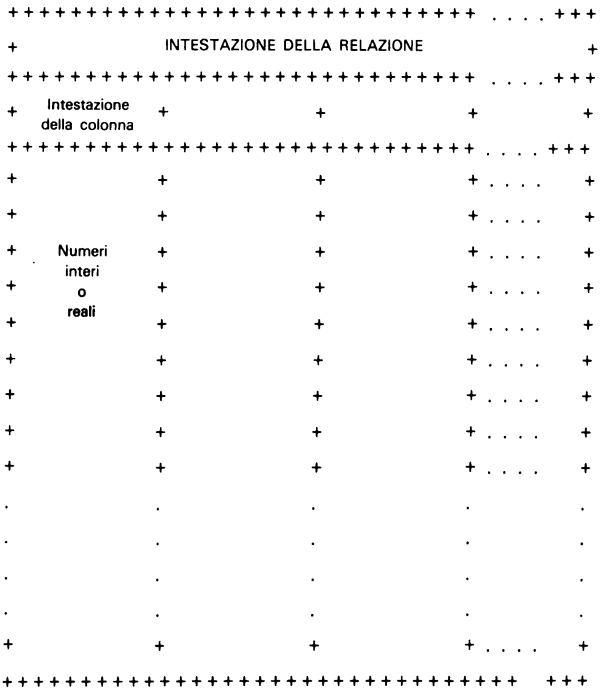


Figura 8.3 Formato dell'impaginazione per l'Esercizio 8.4

procedure *StampaIntero*(*numero*: *integer*; *lunghezzacolonna*: *lunghezza*)
procedure *FineColonna*

*8.5 Confronto di file

Si devono confrontare file di tipo *text*, con linee di lunghezza variabile, generate da un text editor. Ciascuna linea contiene un numero progressivo di linea (un intero generato dall'editor), seguito da un testo (prodotto dall'utente). Le linee del file di riferimento sono numerate in ordine strettamente crescente a partire dalla linea 1. Anche il file da confrontare ha linee ordinate in ordine crescente, ma sono possibili dei buchi, dovuti alla cancellazione di righe, o linee senza numero, che sono state aggiunte in un secondo momento. Se una linea viene solo modificata, ma non cancellata o aggiunta, il suo numero non varia.
Si scriva una procedura con tre parametri: il file di tipo *text* di riferimento, il file da verificare e un nuovo file di tipo *text* di riferimento che si ottiene rinumerando le linee del secondo file. Questa procedura deve stampare una successione di messaggi che mostrino le seguenti differenze fra i due file di input:

1. Modifiche di una particolare linea: vengono stampate la linea di riferimento e la linea da stampare, dal primo all'ultimo carattere diverso.
2. Inserimento o cancellazione di una linea: tale linea viene stampata.

I numeri di linea sono letti usando la chiamata a *read*(*f*, *numlinea*), e sono riscritti con la chiamata di *write*(*f*, *numlinea*: 5, ' ').

9

Gli array

La struttura dati più frequentemente utilizzata è l'array e in alcuni linguaggi — come Fortran, Algol 60, BASIC e APL — è addirittura l'unica disponibile. Questo non significa però che sia anche la più importante, o la più semplice, e neppure la più generale. Per di più, ne viene spesso limitato l'uso e ristretto il campo di definizione, senza che esista un'apparente necessità; può così essere utile ed interessante esaminare dapprima la struttura astratta di riferimento, di cui l'array è solo una particolare realizzazione.

9.1 TRASFORMAZIONI

In una struttura sequenziale si può accedere solo ad un elemento per volta. Inoltre, per accedere ad un dato elemento è necessario scandire tutti gli elementi che lo precedono. Per strutture dati completamente differenti — in cui il meccanismo di accesso è diretto o casuale invece che sequenziale — valgono altre proprietà e si riscontrano comportamenti diversi. La trasformazione è appunto una struttura diretta.
Una *trasformazione* stabilisce una corrispondenza fra valori appartenenti ad un dato dominio e valori in un dato codominio. Il dominio sorgente è costituito dall'insieme di valori di *tipo indice* ed il codominio da quelli di *tipo elemento*. In altre parole, dato un valore per l'indice, la trasformazione produce il valore dell'elemento corrispondente. Questo concetto è differente da quello di funzione, poiché la trasformazione non è una porzione di programma che calcola l'elemento, ma una struttura che lo contiene.
Data una trasformazione di tipo M, da un indice di tipo I ad un elemento di tipo E, si indica con $M <e^+>$ (con e di tipo E) la trasformazione che determina e per ogni valore dell'indice, cioè la trasformazionale in cui tutti gli elementi sono uguali ad e. Si indica inoltre con $M <m;i:e>$, dove m, i, e sono di tipo M, I, E rispettivamente, la trasformazione che è identica a m ad eccezione dell'elemento con indice i che è e. È possibile costruire tutti gli oggetti di tipo

M con le due regole seguenti:

1. Se e è di tipo E, allora $M <e^+>$ è di tipo M.
2. Se m è di tipo M, i di tipo I ed e di tipo E, allora $M <m;i:e>$ è di tipo M.

Chiameremo la notazione $M <m;i:e>$ una *modifica selettiva* della trasformazione m: il componente con indice i è modificato ed assume il valore e. Il significato di questa notazione, in un'operazione di composizione, è chiarito dalla regola seguente:

3. Se m è di tipo M, i, i' di tipo I ed e, e' di tipo E, allora $M <M <m;i:e>; i':e'>$ ha il seguente significato: se $i = i'$ equivale a $M<m;i':e'>$; altrimenti essa equivale a $M <M<m;i':e'>; i:e>$. Un'abbreviazione possibile per questa notazione è $M <m;i:e, i':e'>$.

Definiamo infine la *selezione di un elemento* come quell'operazione che, dato un indice, determina l'elemento corrispondente e che si rappresenta con l'indicatore della trasformazione seguito dall'indice fra parentesi quadre. La selezione di un elemento è definita dalle due regole seguenti:

4. Se i è di tipo I ed e di tipo E, allora $M <e^+>[i]$ è e (poiché tutti gli elementi di $M <e^+>$ sono e).
5. Se m è di tipo M, i, i' di tipo I ed e di tipo E, allora $M<m; i:e>[i']$ ha il seguente significato: se $i = i'$ vale e (si è modificato in modo selettivo proprio questo elemento), altrimenti è $m[i']$.

Le regole precedenti sono sufficienti per definire una struttura di dati molto astratta, che però è praticamente impossibile da usare — senza definire su di essa altre operazioni — ma che, di per sé, non obbliga a ricorrere ad una particolare implementazione per la sua realizzazione. Si noti, per esempio, che non è stata posta alcuna restrizione sul tipo dell'indice, che può perciò essere di qualsiasi tipo finito (ma tutti i tipi devono per forza essere finiti se li si vuole rappresentare su un calcolatore). D'altro canto la notazione proposta per la modifica selettiva suggerisce che, data una particolare trasformazione, per modificarne un elemento occorre costruire un'altra trasformazione, che è una copia della prima ad eccezione dell'elemento modificato.
Esattamente come si è visto a proposito del concetto astratto di sequenza, anche per la trasformazione possono esistere diverse implementazioni, in funzione delle restrizioni che si considerano accettabili e delle operazioni che devono essere effettuate in modo efficiente. Per esempio, se si sa che il numero degli elementi sarà molto grande, diventa necessaria un'implementazione su un dispositivo periferico, e l'accesso casuale rende necessario l'uso di un cosiddetto *dispositivo ad accesso casuale* come un disco rigido o un dischetto. Se l'intervallo dei valori del tipo indice — che determina il numero massimo di elementi possibili — è molto più ampio del numero di elementi realmente presenti o utilizzabili, la struttura risultante è spesso chiamata *tabella associativa* — se realizzata in memoria centrale — o *file indicizzato* — se realizzata su memoria di massa. Se

c'è un numero di elementi esattamente uguale al numero di valori distinti dell'indice, la struttura risultante è un *file ad accesso diretto* — su memoria di massa — e un *array* in memoria centrale. Quest'ultima alternativa è stata scelta e implementata in Pascal, come nella maggior parte degli altri linguaggi, come Fortran e PL/1. Tuttavia la struttura chiamata array in BASIC, e perfino in APL, è più una tabella che non un array vero e proprio, perché l'elemento corrispondente ad un dato valore di un indice generalmente non esiste fino a che non viene referenziato.

9.2 GENERALITÀ SUGLI ARRAY

Un array è un insieme di elementi tutti dello stesso tipo, ciascuno dei quali accessibile separatamente. Il numero di elementi è fissato nel momento in cui l'array viene definito, ed è pari alla cardinalità del tipo dell'indice. Ciò significa che gli indici possono essere solo di un tipo ordinale: sottocampi di interi o caratteri, tipi scalari (fra cui i booleani). D'altra parte, non vi è alcuna ragione per porre dei limiti sul tipo dell'elemento, che può essere sia semplice che strutturato.

Mentre un matematico definirebbe una funzione con la notazione *funzione* : *dominio*→*codominio*, l'array del Pascal è definito dalla sintassi mostrata nella Figura 9.1, dove l'indice può essere di tipo ordinale, mentre l'elemento può essere di un tipo arbitrario. Nel paragrafo 9.3 verrà trattato il caso di più indici, di tipo diverso, separati da una virgola. Le parentesi quadre che compaiono nel diagramma sintattico sono ridondanti, ma ricordano la notazione usata per gli indici delle matrici.

Sugli elementi di trasformazioni astratte sono state definite notazioni per la costruzione e la modifica selettiva. Non è peraltro desiderabile dare queste definizioni anche per gli array reali, perché ciò implicherebbe la costruzione di nuovi oggetti o la loro copia ogni volta che se ne modifica un elemento. Infatti, l'array Pascal è una struttura residente in memoria e la selezione di un suo elemento (tramite un indice), è un modo per accedere ad un elemento di questa struttura. Al posto delle operazioni definite sulle trasformazioni, il Pascal definisce per l'array le seguenti operazioni, molto simili a quelle trovate in altri linguaggi:

1. *Assegnamento*: come per ogni oggetto, ad eccezione dei file (che non sono una struttura residente in memoria centrale), è possibile effettuare un asse-

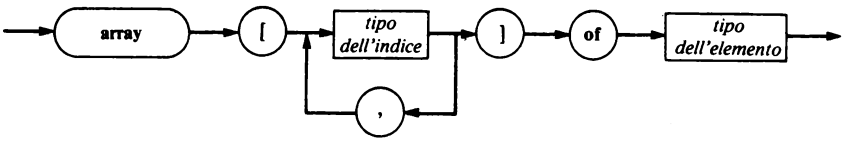


Figura 9.1 Diagramma sintattico di un tipo array

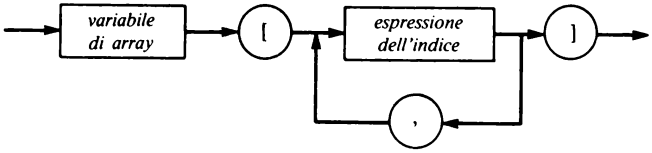


Figura 9.2 Diagramma sintattico di una variabile indicizzata

gnamento fra due array dello stesso tipo. Naturalmente, se le dimensioni di un array sono grandi, questa operazione può essere molto costosa.

2. *Indicizzazione*: si può selezionare un elemento di un array tramite la notazione mostrata in Figura 9.2, dove *variabile di array* deve essere una variabile di tipo array ed *espressione dell'indice* deve essere un'espressione compatibile in assegnamento con il tipo dell'indice. La possibilità di elencare più espressioni, nel campo indice, verrà trattata nel paragrafo 9.3. La variabile indicizzata è del tipo dell'elemento generico dell'array.

Non esiste perciò alcuna notazione particolare per la costruzione di un array o per la modifica di un suo elemento. La seconda operazione viene effettuata assegnando un nuovo valore ad una variabile indicizzata, mentre la prima avviene modificando ripetutamente tutti gli elementi di un array. Immediatamente dopo la sua definizione, il valore di un array è indefinito e rimane tale fino a che non è stato assegnato un valore a tutti i suoi elementi.

ESEMPIO 9.1: DICHIARAZIONI DI ARRAY E INDICIZZAZIONI

```
type numgiorni = 28..31;
     mese = (Gen, Feb, Mar, Apr, Mag, Giu, Lug, Ago, Set, Ott, Nov, Dic);
     anno = array [mese] of numgiorni;
     colore = (blu, rosso, verde, giallo);
var x: array [-4..20] of real;
     y: array [colore] of colore;
     z: array [char] of mese;
     i, j: integer;
{x ha 25 elementi di tipo real, con indici -4, -3, ..., 0, 1, ..., 20; y ha 4 elementi di
     tipo colore, con indici blu, rosso, verde, giallo; il numero di elementi di z è il nu-
     mero di caratteri definiti nell'implementazione, con tutti i caratteri disponibili co-
     me possibili indici, per esempio 'a', '4', ';',...
     x[10] è di tipo real, come x[i+j];
     y[rosso] e y[y[rosso]] sono di tipo colore;
     z['+'] è di tipo mese e x[ord(z[input!])] è di tipo real}
```

COMMENTI

1. In Pascal un array non è tanto un tipo quanto un descrittore di tipo, che consente la definizione di un'infinita varietà di tipi. Può essere un tipo esplicito, come *anno* in questo esempio, o anonimo, come ad esempio le variabili *x*, *y* e *z*.
2. Al contrario di quanto accade in Fortran, BASIC, PL/1 e Cobol, il tipo dell'indice di un array non è vincolato ad essere un sottocampo degli interi, né il suo estremo inferiore ad essere 1.
3. Il numero degli elementi di un array è determinato dalla cardinalità del tipo indice. Dal momento che i tipi sono definiti staticamente, questo numero di elementi è fissato al momento della scrittura del programma e non può variare durante l'esecuzione. La definizione di *array dinamici*, come in PL/1, non è prevista in Pascal.
4. L'uso di indici di tipo *integer* è legale, ma il loro impiego è generalmente impossibile, dal momento che le dimensioni dell'array risultante sarebbero troppo grandi. Il tipo *real* è illegale, dal momento che non è un tipo ordinale, e non si può quindi determinare il numero di valori distinti che si trovano in un qualsiasi intervallo di numeri reali.

ESEMPIO 9.2: RICERCA BINARIA

```
type tabella = array [limiteinferiore..limitesuperiore] of T
{limiteinferiore e limitesuperiore sono due costanti intere; T è un tipo semplice};
procedure RicercaBinaria
        (var tb: tabella {la tabella ordinata da esaminare};
        x: T {il valore da ricercare in tb};
        var trovato: Boolean {vale true se almeno un elemento di tb è uguale a x,
            false altrimenti};
        var i: integer {se trovato vale true, tb[i] = x}
        );
    var sinistro, destro: integer;
begin {RicercaBinaria; si veda l'Esempio 6.4}
    sinistro := limiteinferiore; destro := limitesuperiore;
    while sinistro ≤ destro do
    begin {x non compare in tb prima di sinistro o dopo destro}
        i := (sinistro + destro) div 2;
        if tb[i] ≤ x then sinistro := i + 1;
        if tb[i] ≥ x then destro := i - 1
    end {while sinistro ≤ destro};
    trovato := x = tb[i]
end {RicercaBinaria}
```

COMMENTI

1. Il tipo dell'indice di *tabella* non può essere un qualsiasi tipo ordinale a causa delle operazioni che devono venir eseguire su di esso. Mentre si potrebbe sostituire $i+1$ e $i-1$ con *succ*(*i*) e *pred*(*i*), l'espressione (*sinistro* + *destro*) **div** *2* non può essere generalizzata ai tipi scalari. Tuttavia, per il tipo *char* (o per un suo sottocampo), si può usare l'espressione *chr* ((*ord*(*sinistro*) + *ord*(*destro*)) **div** *2*); perciò si può ancora effettuare la ricerca binaria su array i cui indici siano caratteri.
2. I limiti degli indici sono fissati dalla definizione del tipo; di conseguenza questa procedura non può trattare un array con limiti diversi e neppure un array con gli stessi limiti ma di tipo diverso. Nel paragrafo 9.7 mostreremo come rendere questo vincolo meno restrittivo.

ESEMPIO 9.3: FREQUENZA RELATIVA DELLE LETTERE

```
program FrequenzaRelativa (input, output)
  {il file input di tipo text viene letto e copiato su output, seguito dalla frequenza
     relativa di tutte le lettere minuscole che vi compaiono};
  type naturali = 0..maxint;
  var contatore: array['a'..'z'] of naturali;
     ch: char;
  function EUnaLettera(ch: char): Boolean
     {dal momento che non si conosce il set di caratteri su cui si opera, non è pos-
        sibile garantire che tutti i caratteri fra 'a' e 'z' siano lettere; questa fun-
        zione tratta tutti i set di caratteri correntemente in uso, specialmente l'ISO
        (ASCII) e l'EBCDIC};
  begin {EUnaLettera}
     EUnaLettera :=
        ('a' ≤ ch) and (ch ≤ 'i') or
        ('j' ≤ ch) and (ch ≤ 'r') or
        ('s' ≤ ch) and (ch ≤ 'z')
  end {EUnaLettera};
begin {FrequenzaRelativa}
  ch := 'a';
  while ch ≤ 'z' do {inizializza contatore}
  begin contatore[ch] := 0; ch := succ(ch) end;
  while not eof(input) do
  begin
     while not eoln(input) do
     begin read(input, ch); write(output, ch);
        if EUnaLettera(ch) then
           contatore[ch] := contatore[ch] + 1
     end;
```

```
        writeln(output); readln(input)
    end {while ¬ eof(input)};
    page(output); writeln(output, 'Frequenza delle lettere');
    writeln(output) ch := 'a';
    while ch ≤ 'z' do
    begin
        if EUnaLettera(ch) then
            writeln(output, ch: 4, contatore[ch]: 9);
        ch := succ(ch)
    end
end {FrequenzaRelativa}
```

9.3 ARRAY MULTIDIMENSIONALI

Tutti gli array presentati negli esempi precedenti hanno componenti di tipo semplice: gli elementi di un array possono tuttavia essere di un qualsiasi tipo strutturato. Così, un programmatore può definire array di file, di record (si veda il Capitolo 11), di set (Capitolo 12) o di puntatori (Capitolo 13) e anche array di array, normalmente chiamati *array multidimensionali*. Il tipo *matrice* definito come

```
type matrice = array [m..n] of array [p..q] of real
```

ha $ord(n) - ord(m) + 1$ elementi, indici m, $succ(m)$, ..., n, che sono array con un numero di elementi pari a $ord(q) - ord(p) + 1$ di tipo *real* e con indici p, $succ(p)$,..., q. Se *mat* è una variabile di tipo *matrice*, *mat*[*i*] è di tipo **array**[*p..q*] **of** *real* e *mat*[*i*][*j*] è di tipo *real*.
Per semplicità e per non scostarsi troppo dagli altri linguaggi, il Pascal consente notazioni abbreviate per gli array di array. In una descrizione di tipo,]**of array**[può essere sostituito da una virgola e, nella variabile,][può ancora essere sostituito da una virgola. Ciò significa che la seguente definizione di tipo è del tutto equivalente alla precedente:

```
type matrice = array [m..n, p..q] of real
```

e che *mat*[*i*] [*j*] è sempre identico a *mat*[*i, j*]. Per di più, si può definire il tipo *matrice* ancora in un modo diverso:

```
type vettore = array [p..q] of real;
     matrice = array [m..n] of vettore;
```

che ancora una volta è del tutto equivalente alle due definizioni precedenti. Qualunque sia la forma esatta della definizione del tipo, *mat* è sempre un array di array, la variabile *mat*[*i*] è legale e denota un array di reali che in un certo

senso è un sotto-array o una riga di *mat*. Naturalmente, queste abbreviazioni sono valide per un qualsiasi numero di dimensioni.

ESEMPIO 9.4: PRODOTTO DI MATRICI

```
type matrice = array [1..n, 1..n] of real {matrice quadrata}
procedure ProdottoDiMatriciQuadrate (a, b: matrice; var c: matrice)
      {calcola c = ab};
   var i, j, k: 0..n;
      somma: real;
begin {ProdottoDiMatriciQuadrate}
   i := 0;
   while i < n do
   begin i := i+1; j := 0;
      while j < n do
      begin j := j+1; k := 0; somma := 0;
         while k < n do
         begin k := k+1;
            somma := somma+a[i, k] * b[k, j]
         end {ciclo su k; somma = Σ_{m=1}^{n} a_im b_mj};
         c[i, j] := somma
      end {ciclo su j}
   end {ciclo su i; c_ij = Σ_{k=1}^{n} a_ik b_kj}
end {ProdottoDiMatriciQuadrate}
```

COMMENTI

Per molte ragioni (elencate qui sotto) questa procedura è molto rozza, e versioni successive le aggiungeranno tutte quelle caratteristiche di cui attualmente manca.

1. I cicli non sono naturali e richiedono, per le tre variabili, un intervallo di definizione che abbia come estremo inferiore 0 e non 1; questi cicli verranno sostituiti dall'istruzione **for** nel Capitolo 10.
2. Le matrici *a* e *b*, essendo parametri passati per valore, sono di fatto variabili locali alla procedura e richiedono altrettanto spazio per la loro memorizzazione di quello richiesto dai parametri attuali; inoltre è necessaria un'operazione di copiatura dei parametri formali all'atto della chiamata della procedura. Questo costo può essere evitato ricorrendo a parametri di tipo variabile, il cui uso richiede un po' di attenzione (si veda il paragrafo 9.6).
3. La dimensione delle matrici che si devono moltiplicare è fissata al momento della definizione del tipo e questa procedura non può trattare matrici di numeri reali di dimensioni diverse, e neppure calcolare il quadrato di matrici

delle stesse dimensioni ma di tipo diverso (un tipo identico con un nome diverso, per esempio, o un tipo anonimo con la stessa struttura); per superare questa difficoltà si possono usare i *conformant-array* (si veda il paragrafo 9.7).

Esempi più significativi dell'uso di array multidimensionali o più complicati verranno presentati nei paragrafi e capitoli successivi.

9.4 ARRAY COMPATTATI

La rappresentazione di una qualsiasi struttura dati, su di una macchina, impone sempre un compromesso fra il tempo necessario per l'accesso dei suoi elementi e lo spazio richiesto per la loro memorizzazione. Questo compromesso è particolarmente critico quando gli elementi della struttura non hanno dimensioni che sono multipli dell'unità elementare di memoria del calcolatore. Per esempio, non esiste una rappresentazione perfetta per un array di *Boolean* su di un calcolatore con una lunghezza di parola di 32 bit; se si usa una parola per ciascun elemento dell'array, il loro accesso e la loro modifica risultano operazioni efficienti, ma si sprecano 31 bit per ogni bit utile; se invece si compattano 32 valori per parola, si fa un uso migliore della memoria, ma ciascun accesso e ciascuna modifica risultano rallentati dall'uso di maschere e dalla rotazione di bit.
Idealmente un compilatore potrebbe scegliere la rappresentazione migliore in funzione delle dimensioni e della frequenza d'uso dell'array. In molte situazioni il programmatore sa a priori quale dei due fattori — occupazione o velocità — privilegiare e può informare di conseguenza il compilatore.
Ciascuna definizione di strutture dati in Pascal (sia in una definizione di tipo che in una dichiarazione di variabile) può essere preceduta dalla parola chiave **packed**, che informa il compilatore che, se possibile, occorre privilegiare l'occupazione di memoria, anche se così si penalizzano le operazioni di accesso e di modifica degli elementi. In assenza di questa richiesta esplicita, invece, la decisione implicita è di privilegiare la velocità delle operazioni.
È importante notare che il programmatore Pascal non ha alcun modo di specificare una particolare rappresentazione di oggetti strutturati, che infatti dipende dal calcolatore ospite e dalla particolare implementazione. Per di più, il compilatore ha la completa libertà di ignorare del tutto l'attributo **packed**; esso non deve assolutamente cambiare il significato di un programma. Alcuni compilatori possono cercare di compattare solo gli elementi di certi tipi e non di altri. Probabilmente un array compattato di reali ha la stessa rappresentazione di uno non compattato.
Un altro aspetto importante è che sono molto pochi i casi in cui il prefisso **packed** è realmente utile e si possono dividere in due categorie: la prima riguarda gli array che devono essere trattati come stringhe di caratteri, la cui discussione è rimandata al paragrafo 9.5; la seconda riguarda array (o strutture dati) di grosse dimensioni con elementi molto piccoli. Compattare un array di piccole dimensioni può essere controproducente per quanto riguarda l'occupazione di

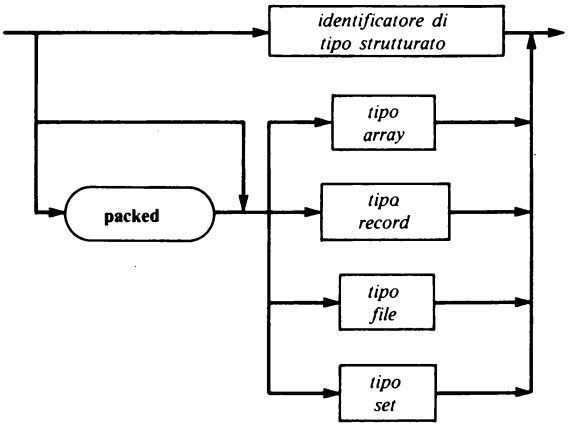


Figura 9.3 Diagramma sintattico di un tipo strutturato

memoria, dal momento che il codice necessario per accedere ai suoi elementi può avere dimensioni maggiori dello spazio risparmiato compattando.
Un qualsiasi tipo strutturato può essere compattato, in accordo con quanto mostrato nella Figura 9.3. I tipi record e set saranno trattati rispettivamente nei Capitoli 11 e 12. L'attributo **packed** si applica anche ai tipi file, ma generalmente non ha alcun effetto, dal momento che è difficile trovare un significato per questa operazione. Tuttavia, su un calcolatore a 16 bit, che utilizzi il codice ISO, un **file of** *char* potrebbe usare una parola per carattere, mentre un **packed file of** *char* potrebbe compattare due caratteri in una parola: questa osservazione avrebbe però senso solo se si potesse parlare di «parole» per un dispositivo periferico.
Si devono ora esaminare alcuni punti molto importanti, anche se l'attributo **packed** non ha ripercussioni sulla semantica di un programma, ma solo sulle sue prestazioni.

1. Nel caso di array multidimensionali, il compattamento può essere fatto a più livelli. Dati i due tipi ordinali (o sottocampi di essi) *tiporiga* e *tipocolonna*, si possono verificare quattro casi diversi:

   a) **array** [*tiporiga*] **of array** [*tipocolonna*] **of** *tipoelemento*
   {Equivale a array [tiporiga, tipocolonna] of tipoelemento}

   b) **packed array** [*tiporiga*] **of array** [*tipocolonna*] **of** *tipoelemento*
   {Questa dichiarazione ha senso probabilmente solo se ciascuna riga richiede molto meno spazio delle dimensioni di una «parola» del calcolatore}

   c) **array** [*tiporiga*] **of packed array** [*tipocolonna*] **of** *tipoelemento*

   d) **packed array** [*tiporiga*] **of packed array** [*tipocolonna*] **of** *tipoelemento*

```
{Questa dichiarazione equivale a packed array [tiporiga, tipocolonna] of tipoelemento; nella forma abbreviata, l'attributo packed gode della proprietà transitiva}
```

Il caso c) è generalmente l'unico da tenere presente, perché molti compilatori non accettano di compattare elementi più lunghi di una «parola».

2. L'operazione di compattamento è una caratteristica del tipo e, di conseguenza, valgono le regole di compatibilità di tipo. Un tipo e la sua versione compattata non sono compatibili in un'istruzione di assegnamento o nel passaggio di parametri. Inoltre non si può neppure definire un tipo compattato a partire dalla definizione di un altro tipo:

   Tipocompattato = **packed** *T*

   è perciò illegale.

3. Il parametro attuale corrispondente ad un parametro formale di tipo variabile non può essere un elemento di una struttura compattata, perché ciò implicherebbe che il sottoprogramma sappia come questa struttura è stata effettivamente compattata. Tuttavia, in tutti gli altri casi, un elemento di una struttura compattata non manifesta alcun comportamento speciale e può benissimo corrispondere, ad esempio, ad un parametro formale. Inoltre i vari parametri di tutti i sottoprogrammi predefiniti non sono descritti come parametri di tipo variabile e quindi questa regola non si applica; in particolare è perfettamente legale leggere elementi di una struttura compattata con le procedure predefinite *read* e *readln*.

4. Esistono due procedure predefinite che consentono di trasferire valori da array compattati ad array che non lo sono (*unpack*) e viceversa (*pack*). Si abbiano le seguenti dichiarazioni, o altre equivalenti:

   var *noncompattata* : **array** [*tipoindice1*] **of** *T*;
   compattata : **packed array** [*tipoindice2*] **of** *T*;

   e siano *basso* e *alto* rispettivamente il valore minimo e massimo di *tipoindice2*. Se *expr* è un'espressione compatibile in un'istruzione di assegnamento con *tipoindice1*, allora l'istruzione:

   pack (*noncompattata*, *expr*, *compattata*)

   equivale a

```
begin temp1 := expr ; temp2 := pred(basso);
   while temp2 < alto do
   begin temp2 := succ(temp2);
      compattata [temp2] := noncompattata [temp1];
```

```
        if temp2 ≠ alto then temp1 := succ(temp1)
    end
end
```

dove *temp1* e *temp2* sono variabili ausiliarie che non servono altrove nel programma. In modo del tutto analogo l'istruzione:

unpack (*compattata, noncompattata, expr*)

è equivalente a

```
begin temp1 := expr; temp2 := pred(basso);
    while temp2 < alto do
    begin temp2 := succ(temp2);
        noncompattata [temp1] := compattata [temp2];
        if temp2 ≠ alto then temp1 := succ(temp1)
    end
end
```

Queste due procedure effettuano perciò una copia parziale di un array non compattato ed una copia completa di un array compattato. Questa mancanza di simmetria, a prima vista, può generare confusione. I tipi *tipoindice1* e *tipoindice2* possono non avere niente in comune, ma tutti i riferimenti a *noncompattata* [*temp1*] devono essere legali; deve perciò esistere in *tipoindice1* un numero di valori, con estremo inferiore uguale a *expr*, che sia almeno pari alla cardinalità di *tipoindice2*.

9.5 STRINGHE

Nel Capitolo 1 si è definita una stringa come una costante che rappresenta una sequenza di caratteri. L'unico uso che abbiamo fatto fino ad ora delle stringhe è stato nelle stampe. Ora abbiamo a disposizione tutti gli elementi necessari per completare la descrizione delle stringhe di caratteri.
Una stringa di caratteri che abbia un solo elemento è una costante di tipo *char*, ma una stringa di caratteri con più di un elemento è una costante di un determinato *tipo stringa*, che abbia lo stesso numero di elementi. La definizione di un tipo stringa è della forma **packed array** [1.. *maxchar*] **of** *char* o equivalenti, dove *maxchar* è una costante maggiore di 1. Occorre osservare le due restrizioni imposte da questa definizione: un tipo stringa è sempre compattato e l'estremo inferiore è sempre 1. Si noti anche che un array compattato i cui elementi siano un sottocampo del tipo *char* non è una stringa.
Con le costanti di caratteri e le variabili di tipo stringa è possibile trattare le stringhe quasi nello stesso modo delle variabili di tipo semplice, con in più la possibilità di accedere al singolo carattere ed in meno l'impossibilità di usare le

stringhe come valori calcolati da funzioni (dal momento che in effetti non sono un tipo semplice).
Per le stringhe di tipo compatibile sono definiti tutti gli operatori relazionali (benché questi non siano definiti sugli array ordinari). Due stringhe sono uguali se hanno la stessa lunghezza (altrimenti non potrebbero essere neppure confrontate) ed elementi uguali in posizioni uguali. Sono diverse se almeno una coppia di elementi è diversa. Gli operatori $<$, $\leq$, $\geq$, $>$ denotano un ordinamento lessicografico: date due stringhe confrontabili (cioè compatibili) *s1* e *s2*, $s1 < s2$ se e solo se esiste un indice *p*, compreso fra *1* e *maxchar*, tale che per tutti gli *i* da *1* a $p-1$, $s1[i] = s2[i]$, e $s1[p] < s2[p]$. In modo del tutto analogo si definiscono gli operatori $\leq$, $\geq$, $>$. Questo significa che è determinante l'ordinamento implicito dell'insieme dei caratteri di riferimento. Per esempio, la posizione di uno spazio, in questo ordinamento, può implicare che *'ab '* sia maggiore di *'abc'*, contrariamente a quanto si riscontra nei normali vocabolari. Per evitare questo problema, di solito tutti gli spazi a destra di una stringa vengono sostituiti con *chr(0)*, un carattere che non corrisponde ad alcuna lettera in nessuna delle codifiche normalmente utilizzate.
Le stringhe godono di un'ulteriore proprietà: è infatti possibile scriverle direttamente in file di tipo *text* (si veda l'Esempio 8.5). Tuttavia le stringhe non possono essere lette direttamente con la procedura predefinita *read*: la ragione principale risiede nel fatto che è molto semplice scrivere un apposito programma di lettura, che risulta fra l'altro più flessibile, sfruttando i molti modi esistenti per determinare la lunghezza di ciò che si vuole leggere, utilizzando gli end-of-line.
Benché le stringhe, in Pascal, siano sufficienti per alcune applicazioni, non sempre consentono di fare tutto ciò che si vorrebbe. In confronto al PL/1 o a linguaggi specializzati nel trattamento di stringhe, mancano di flessibilità e di generalità, soprattutto perché hanno una lunghezza fissa che non può cambiare in alcun modo. Di conseguenza non è definito alcun operatore di concatenazione; una data stringa può essere assegnata solo a stringhe di ugual lunghezza (benché molte implementazioni siano più flessibili) ed il loro uso come parametri di sottoprogrammi è limitato (con l'unica eccezione descritta nel paragrafo 9.7). Nel Capitolo 13 vedremo un esempio di compattamento con definizioni di tipo e dichiarazioni di procedure che potrebbero essere usate per definire le stringhe in un modo più generale e completo.

ESEMPIO 9.5: CONCORDANZA DI TESTI

```
program ConcordanzaDiTesti(input, output);
  {questo programma legge un testo e ne esegue una semplice analisi di concordanza, cioè prepara un elenco alfabetico delle parole che compaiono nel testo. Una parola è, per definizione, un insieme di lettere che non contengono alcun altro carattere}
  const lunghparola = 20 {massima lunghezza utile per le parole};
```

```
    lunghdizionario = 1000 {massima lunghezza del dizionario};
  type tipoparola = packed array[1..lunghparola] of char {una stringa};
  var dizionario: array[1..lunghdizionario] of tipoparola;
    parola: tipoparola {la parola corrente};
    dimdizionario, indice: 0..lunghdizionario {due indici nel dizionario};
  function EUnaLettera(ch: char): Boolean; ... {si veda l'Esempio 9.3};
  procedure LeggeUnaParola
    {il prossimo carattere in input è una lettera; legge la parola corrispondente
      nella variabile globale parola};
    var parolaletta: array[1..lunghparola] of char {non compattato};
      k: 0..lunghparola;
  begin {LeggeUnaParola}
    k := 0;
    repeat {EUnaLettera(input↑)}
      if k < lunghparola then
      begin k := k+1;
        parolaletta[k] := input↑
      end;
      get(input)
    until not EUnaLettera(input↑);
      {la parola è stata letta}
    while k < lunghparola do {lo si completa con spazi}
    begin k := k+1;
      parolaletta[k] := ' '
    end;
    pack(parolaletta, 1, parola)
      {si spera che sia meglio di una lettura diretta}
  end {LeggeUnaParola};
  procedure RicercaEInserisce
    {la parola viene inserita nel dizionario se non è già presente e se c'è almeno
      una cella libera; il dizionario è già ordinato};
    var sinistra, centro, destra: 0..lunghdizionario;
  begin {RicercaEInserisce}
    if dimdizionario = 0 then {dizionario vuoto, si inserisce la parola}
    begin dizionario[1] := parola; dimdizionario := 1 end
    else begin {caso generale}
      sinistra := 1; destra := dimdizionario;
      while sinistra ≤ destra do {ricerca binaria}
      begin centro := (sinistra+destra) div 2;
        if dizionario[centro] ≤ parola then
          sinistra := centro+1;
        if dizionario[centro] ≥ parola then
          destra := centro-1
      end;
      if dizionario[centro] ≠ parola then
```

```
                {inserisce se possibile}
              if dimdizionario < lunghdizionario then {ci sta}
              begin dimdizionario := dimdizionario + 1;
                 destra := dimdizionario;
              if dizionario[centro] < parola then
                 {ma dizionario[centro + 1] > parola}
                 centro := centro + 1;
              while destra > centro do {sposta le parole}
              begin dizionario[destra] := dizionario[destra - 1];
                 destra := destra - 1
              end;
              dizionario[centro] := parola
           end
        end {caso generale}
     end {RicercaEInserisce}
  begin {programma ConcordanzaDiTesti}
     dimdizionario := 0;
     repeat { ¬eof(input)}
        if EUnaLettera(input↑) then {inizio di parola}
        begin LeggeUnaParola { ¬EUnaParola(input↑)};
           RicercaEInserisce
        end
        else get(input) {salta il carattere diverso da una lettera}
     until eof(input);
     page(output);
     writeln(output, 'Concordanza'); writeln (output);
     indice := 0;
     repeat
        indice := indice + 1; writeln (output, dizionario[indice])
     until indice = dimdizionario
  end {ConcordanzaDiTesti}.
```

COMMENTI

1. Questo programma non fornisce un'analisi di concordanza reale fra testi, ma solo un indice di parole. Sarebbe utile aggiungere un elenco dei numeri delle linee dove compare una data parola, oppure le linee stesse, opportunamente ruotate, in modo che la parola esaminata compaia sempre al centro della linea. La prima soluzione è nota con il nome di *cross-reference*, soprattutto nel caso in cui il testo analizzato sia un programma; nel Capitolo 13 verrà descritto un programma di questo genere. La seconda soluzione fornisce un'analisi di concordanza vera e propria, utilizzabile nell'analisi di testi e di vocabolari e può richiedere, ad esempio, un file esterno che viene ordinato prima di essere stampato.

2. La procedura *RicercaEInserisce* fa uso di un metodo di ricerca ed inserzione binario, che non rappresenta sicuramente la soluzione più efficiente, dal momento che inserzioni ripetute obbligano a spostare molte parole già inserite e quest'operazione è piuttosto costosa. L'uso di un metodo di codifica «hashing» è più efficiente, come sarà mostrato in un esempio nel Capitolo 13.

9.6 ARRAY COME PARAMETRI DI SOTTOPROGRAMMI

L'uso di array come parametri di un sottoprogramma crea due problemi notevoli che verranno discussi qui di seguito. Il primo riguarda il modo di trasmissione, che è stato trattato solo parzialmente nel Capitolo 4. Se un parametro di tipo array è passato per valore, il parametro formale corrispondente è una variabile locale al sottoprogramma. Dal momento che questo parametro è usato come parametro di input, non esiste alcun problema semantico, ma occorre tenere ben presente l'efficienza risultante: l'array ha bisogno di memoria anche all'interno della procedura e la copia iniziale porta via del tempo. Si è fatta la stessa osservazione nell'Esempio 9.4 (prodotto di matrici) e dobbiamo ammettere, in quell'esempio, di aver trascurato l'efficienza.
Per evitare questi due inconvenienti è consigliabile usare parametri di tipo variabile invece che passati per valore. Si evita, in questo modo, il costo della doppia memorizzazione e della copia, ma si paga con l'introduzione di un'importante fonte di errore: dal momento che l'array formale non è locale al sottoprogramma, si può modificare, inavvertitamente, il corrispondente parametro attuale. Per di più possono sorgere seri problemi se uno stesso array compare più volte come parametro di tipo variabile nella stessa chiamata di un sottoprogramma.
Si consideri per esempio la procedura *ProdottoDiMatriciQuadrate* dell'Esempio 9.4. Dati gli array *ma*, *mb* e *mc* di tipo *matrice*, la chiamata *ProdottoDiMatriciQuadrate* (*ma*, *mb*, *mc*) dà come risultato in *mc* il prodotto di *ma* per *mb* e queste due matrici rimangono inalterate, sia che siano passate per valore (come nell'esempio) che per indirizzo. Al contrario la chiamata *ProdottoDiMatriciQuadrate* (*ma*, *mb*, *ma*) può causare danni imprevedibili nel caso di parametri di tipo variabile, dal momento che i parametri *a* e *c* sono sostituiti con lo stesso parametro attuale *ma*. Quando si fa un assegnamento a *c*[*i*, *j*], di fatto si modifica *ma* ed è questo valore modificato che viene usato nei passi successivi del ciclo. È praticamente impossibile predire il risultato finale, ma sicuramente esso non sarà il prodotto delle due matrici iniziali *ma* e *mb*.
Un problema simile nasce quando un parametro di tipo variabile non è un array completo, ma un elemento di un array (o di un qualsiasi tipo strutturato). L'accesso all'elemento è calcolato prima di entrare nel sottoprogramma, e l'array può essere quasi del tutto dimenticato. Si consideri la procedura *ProdottoEsteso* (**var** *a*: *matrice*; **var** *b*: *real*), che moltiplica ogni elemento di *a* per *b* (naturalmente un esempio di questo tipo si verifica abbastanza raramente, dal momento che non è assolutamente necessario che *b* sia un parametro di tipo

variabile). La chiamata di *ProdottoEsteso* (*ma*, *x*) modifica *ma* nel modo voluto; ciò non è più vero se la chiamata diventa *ProdottoEsteso* (*ma*, *ma*[*i*, *j*]). Non appena il particolare elemento *ma*[*i*, *j*] viene modificato all'interno della procedura, cambia anche il parametro formale *b* e, da quel momento, la procedura usa un valore diverso.
Questi pericoli insiti nell'uso di parametri di tipo variabile sono seri; fortunatamente capitano solo in particolari condizioni, quando uno stesso array, (o più generalmente una variabile strutturata) compare due o più volte nella chiamata di un sottoprogramma come parametro attuale, o come parte di un parametro attuale che corrisponde ad un parametro formale di tipo variabile. Si deve sempre evitare questa situazione, non solo nei casi più complessi — quando, ad esempio, l'array non compare neppure come parametro esplicito e viene usato e modificato all'interno del sottoprogramma come una variabile globale.

9.7 PARAMETRI DI TIPO CONFORMANT-ARRAY

Il secondo problema in cui ci si può imbattere con parametri di tipo array è stato evidenziato nell'Esempio 9.2, dove abbiamo anche promesso una soluzione. Il problema è vecchio quanto il Pascal e la soluzione qui presentata è quella definita dallo Standard ISO come il miglior compromesso fra la semplicità e il rigore propri del linguaggio originario ed i requisiti della programmazione reale. Se, per esempio, si devono scrivere procedure di ricerca binaria, o di ordinamento, o di moltiplicazione di matrici senza volere aggiungere nuove definizioni al Pascal, occorre congelare la lunghezza esatta delle matrici al momento stesso in cui si scrivono le procedure. Se una procedura di ricerca binaria tratta matrici il cui indice è definito nell'intervallo *m..n*, non le è più consentito gestire array con altri tipi di indice, dal momento che il tipo di quest'ultimo fa parte integrante del tipo dell'array. Questa difficoltà può in parte essere evitata se si hanno più array, della stessa lunghezza totale, ciascuno dei quali copre un diverso intervallo di valori, ma questa non è una soluzione di validità generale. Di conseguenza lo Standard ISO introduce (solo per l'implementazione del livello 1) il concetto di parametro di tipo *conformant-array*, che è un parametro di tipo diverso da quelli finora visti (per valore, o di tipo variabile, procedura e funzione). Nell'intestazione di un sottoprogramma i conformant-array vengono passati per valore o per indirizzo, ma al posto del nome del tipo compare lo schema di dichiarazione di un conformant-array, che è molto simile alla dichiarazione di un tipo di array, come si vede in Figura 9.4. Dal diagramma non si capisce che, se compare l'attributo **packed** nella specifica, non può esserci più di una dimensione di tipo conformant (l'ultima specificata). Inoltre, nel caso di compattamento, si può utilizzare una forma abbreviata simile a quella spiegata nel paragrafo 9.3, ma meno generalizzata.

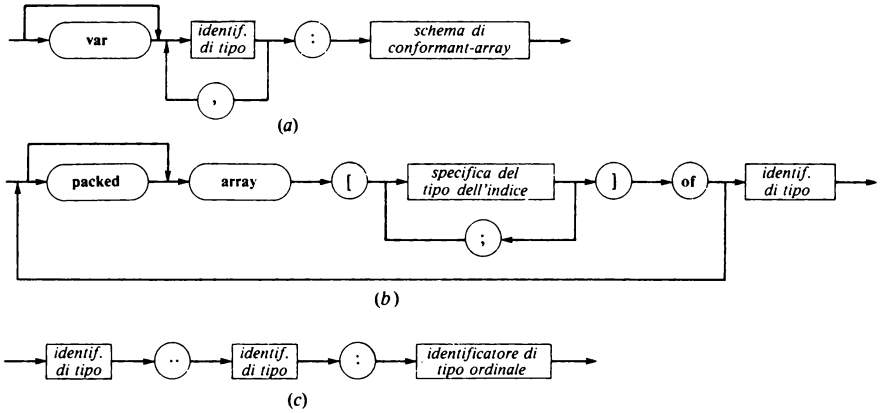


Figura 9.4 Diagramma sintattico dei parametri di un conformant-array: (*a*) specifica dei parametri; (*b*) schema di un conformant-array e (*c*) specifica del tipo dell'indice

ESEMPIO 9.6: SPECIFICA DEI PARAMETRI DI UN CONFORMANT-ARRAY

```
type naturali = 0..maxint;
     tabella = array [1..100] of real;
     stringa = packed array [1..lunghstringa] of char;
...
{seguono ora alcuni esempi di parametri di conformant-array:
    var a: array [i..j: naturali] of real;
        b: packed array [s1..s2: naturali] of char;
        c: array [m..n: char] of stringa;
    var d: array [m1..n1: char] of packed array [m2..n2: naturali] of char;
    var e: array [pbasso..palto: integer; sbasso..salto: integer] of real;
quest'ultima dichiarazione è del tutto equivalente alla seguente:
    var e: array [pbasso..palto: integer] of array [sbasso..salto: integer] of real;}
```

Un parametro formale di tipo conformant-array equivale, nel sottoprogramma nella cui intestazione è dichiarato, ad un array con indice di tipo non completamente noto a tempo di compilazione, in modo simile al Fortran o al PL/1. Tuttavia le differenze rilevanti sono:

a) il tipo ospite del tipo indice è noto;
b) gli estremi esatti del sottocampo di questo tipo sono noti a tempo di esecuzione: possono cioè essere diversi da una chiamata all'altra del sottoprogramma.

I due identificatori che compaiono fra parentesi quadre nella dichiarazione di

un conformant-array sono detti *identificatori di limite* e possono essere usati, all'interno della procedura, come se fossero delle costanti — e quindi in espressioni — ma non nella parte sinistra di un'istruzione di assegnamento. Non sono tuttavia delle vere e proprie costanti e non possono essere usati nella definizione di un tipo, perché ciò andrebbe contro uno dei cardini su cui si basa, in Pascal, la dichiarazione dei tipi: le dimensioni di un oggetto qualsiasi devono sempre essere note durante la compilazione. Nel corpo del sottoprogramma, gli identificatori di limite denotano i valori limite del tipo indice del parametro attuale corrispondente.

Per quanto riguarda la compatibilità di tipo, un parametro formale conformant-array è di tipo diverso da qualsiasi altro tipo, perché anche se i suoi elementi sono di tipo fissato e noto già durante la compilazione, il tipo dell'indice può variare da chiamata a chiamata. Di conseguenza un conformant-array non può essere utilizzato, in istruzioni di assegnamento, a meno che in esse non compaia un altro parametro formale di tipo conformant-array eventualmente presente nella stessa lista di identificatori nella dichiarazione dei parametri. Di fatto è molto raro utilizzare un conformant-array come tale: il suo uso più comune avviene referenziando i suoi singoli elementi.

Alla chiamata del sottoprogramma, il parametro attuale corrispondente al conformant-array, è una variabile — nel caso di parametro di tipo variabile — o un'espressione (che si riduce ad una variabile) o una stringa — nel caso di parametro per valore. Il parametro attuale denota un array di tipo adatto: il tipo dell'elemento deve essere lo stesso tipo di quello che compare nella dichiarazione; il tipo dell'indice deve essere compatibile ed i suoi estremi devono cadere nell'intervallo di valori previsto. Il compattamento, se richiesto, deve avvenire esattamente come specificato. Se più parametri compaiono nella stessa lista di dichiarazione di parametri formali di tipo conformant-array, i parametri attuali corrispondenti devono essere tutti dello stesso tipo, come nell'analoga situazione che si verifica per i parametri di tipo variabile. Allo stato attuale delle cose, questa limitazione è essenziale, perché è possibile avere una sola coppia di identificatori di limiti.

Nel caso il parametro sia passato per valore, prima della chiamata della procedura, viene creata una variabile anonima di tipo adatto: a questa variabile è assegnato il valore del parametro attuale ed alla procedura viene comunicato l'indirizzo di questa variabile anonima. In questo modo, l'effetto è equivalente alla normale trasmissione dei parametri per valore; è consentito quindi modificare il parametro formale all'interno della procedura, ma tale modifica non ha alcun effetto sul parametro attuale. Un conformant-array passato per valore non può essere, a sua volta, un conformant-array passato per valore ad un'altra procedura: non si può quindi ritrasmettere per valore un parametro ricevuto per valore. Se ciò fosse consentito, non sarebbe possibile calcolare le dimensioni della variabile anonima da creare prima della seconda trasmissione. Tuttavia è possibile una ritrasmissione per indirizzo, dal momento che, in questo caso, non è necessario ricorrere a variabili anonime.

Se un parametro di tipo conformant-array è dichiarato come **packed array of**

char, con un indice di tipo *integer* (o di un suo sottocampo), il parametro attuale corrispondente può essere una stringa di caratteri. Naturalmente l'estremo inferiore del tipo dell'indice specificato deve essere minore o uguale a 1. All'interno del sottoprogramma il parametro formale non può essere considerato come una variabile di tipo stringa, dal momento che il suo estremo inferiore non è noto a tempo di compilazione. Di conseguenza, non valgono le proprietà speciali dei tipi stringa (non è possibile usare gli operatori relazionali né avere un conformant-array come argomento della procedura *write*). Occorre scrivere esplicitamente procedure di confronto e di scrittura, nel caso si debbano fare queste operazioni.

Nella regola, data nel paragrafo 4.3, sulla congruenza dei parametri, non si è fatto cenno ai parametri di tipo conformant-array, dal momento che questo quinto modo di passaggio dei parametri non era ancora noto. Due sezioni di parametri formali si corrispondono quindi anche in un quinto modo, quando sono tutte e due conformant-array, con lo stesso numero di parametri e lo stesso schema di costruzione. Questa regola è proposta più per ragioni di completezza che per altro, dal momento che sottoprogrammi parametrici, con parametri di tipo conformant-array, sono estremamente rari e molto poco comuni nei normali programmi.

I conformant-array non sono equivalenti agli *array dinamici*; in Pascal tutti i tipi sono completamente descritti a tempo di compilazione in modo che il compilatore possa determinare in modo esatto la quantità di memoria necessaria per ogni tipo. Si può fare un breve confronto con quanto avviene in altri linguaggi di programmazione: un compilatore Pascal che si fermi al livello 0 (senza conformant-array) è equivalente, per quanto riguarda i limiti di un array, al Cobol. Un compilatore che riconosca il livello 1 ha le stesse caratteristiche del Fortran e del PL/1. Non esiste in Pascal la possibilità di dichiarare array i cui estremi siano calcolati a tempo di esecuzione, che sono addirittura incompatibili con una delle scelte fondamentali fatte dal Pascal, come avviene invece in Algol 60 e 68 e in APL.

Probabilmente i conformant-array sono una delle caratteristiche più complicate del linguaggio e sarà necessario fare molti esempi. Verranno ora proposte delle generalizzazioni di esempi incontrati nei capitoli e paragrafi precedenti. Nei capitoli successivi verranno forniti altri esempi.

ESEMPIO 9.7: RICERCA BINARIA (MIGLIORATA)

```
type T = ...{un tipo semplice se si devono usare gli operatori relazionali, oppure un
            tipo su cui siano definite funzioni di confronto};
procedure RicercaBinaria
   (var tb: array[limiteinferiore..limitesuperiore: integer] of T
      {la tabella ordinata da esaminare};
   x: T {il valore da ricercare in tb};
   var trovato: Boolean {vale true se almeno un elemento di tb è uguale a x,
         false altrimenti};
```

```
    var i: integer {se trovato vale true, tb[i] = x}
    );
    ... {il corpo è del tutto identico all'Esempio 9.2, se T è un tipo semplice; se invece occorre usare un predicato di confronto, le modifiche necessarie sono del tutto evidenti}
    ...
{il primo parametro attuale di questa procedura può essere un qualsiasi array of T il cui tipo indice sia un sottocampo di integer}
```

ESEMPIO 9.8: PRODOTTO DI MATRICI (MIGLIORATO)

```
procedure ProdottoDiMatrici
  (var a: array [pbassoa..paltoa: integer; sbassoa..saltoa: integer] of real;
   var b: array [pbassob..paltob: integer; sbassob..saltob: integer] of real;
   var c: array [pbassoc..paltoc: integer; sbassoc..saltoc: integer] of real
  )
{questa procedura calcola il prodotto di a e b in c; i parametri a e b sono di tipo variabile, per evitare il costo delle copie; non è possibile esprimere direttamente, per mezzo di una dichiarazione di tipo verificabile dal compilatore, la condizione di validità sui limiti degli identificatori; occorre invece effettuare un controllo a tempo di esecuzione, all'interno della procedura, e chiamare la procedura globale Errore se non sono verificate le condizioni sui limiti degli indici};
  var pa, pb, pc, sa, sb, sc: integer;
      somma: real;
begin {ProdottoDiMatrici}
  if (saltoa - sbassoa ≠ paltob - pbassob)
      {seconda dimensione di a ≠ prima dimensione di b}
    and (paltoa - pbassoa ≠ paltoc - pbassoc)
      {prima dimensione di a ≠ prima dimensione di c}
    and (saltob - sbassob ≠ saltoc - sbassoc)
      {seconda dimensione di b ≠ seconda dimensione di c}
  then Errore else
  begin {ciascun array ha il suo insieme di indici}
    pa := pbassoa; pc := pbassoc;
    repeat {ciclo sulla prima dimensione di c e di a}
      sb := sbassob; sc := sbassoc;
      repeat {ciclo sulla seconda dimensione di c e di b}
        sa := sbassoa; pb := pbassob; somma := 0;
        repeat {ciclo sulla seconda dimensione di a e sulla prima di b}
          somma := somma + a[pa, sa] * b[pb, sb];
          sa := sa + 1; pb := pb + 1
        until sa > saltoa;
        c[pc, sc] := somma;
        sc := sc + 1; sb := sb + 1
      until sc > saltoc;
```

```
            pc := pc+1; pa := pa+1
        until pc > paltoc
    end
end {ProdottoDiMatrici}
```

COMMENTI

Dodici identificatori di limite diversi fra loro e sei indici rendono questa procedura estremamente complessa anche se molto generale. Normalmente non ci si può permettere una tale generalità, e nel Capitolo 10 verrà presentata una procedura più semplice.

ESEMPIO 9.9: MESSAGGI DI ERRORE

```
procedure MessaggiDiErrore
    (messaggio: packed array [li..ls: naturali] of char;
    grado, numlinea: integer
    )
{questa procedura, o una simile, può essere utilizzata per generare messaggi di errore
    in un programma che elabora testi, come un compilatore, un macroprocessore, o
    più semplicemente un programma di aggiornamento di file come quello proposto
    nell'Esempio 8.5}
    var i: naturali;
begin {MessaggiDiErrore}
    writeln(output);
    write(output, '****', grado: 2, ' ');
    i := li;
    repeat write(output, messaggio[i]); i := i+1
    until i > ls;
    writeln(output, 'alla linea', numlinea: 1)
end {MessaggiDiErrore}
```

COMMENTI

Questa procedura può essere chiamata sia con una variabile di tipo stringa come suo primo parametro attuale (se, ad esempio, il messaggio comprende dei dati variabili) che con una costante, dello stesso tipo. Questa soluzione è molto più pratica di quella utilizzata nell'Esempio 8.5. Tuttavia non si può trattare, all'interno della procedura, il parametro formale come un parametro di tipo stringa; di conseguenza occorre programmarne esplicitamente la scrittura e non lo si può confrontare con altre stringhe, neppure se dello stesso tipo.

ESERCIZI

9.1 Successioni

Nell'Esempio 6.2 si è fatto uso del tipo *sequenza* per descrivere una successione di oggetti di un tipo semplice *T*, con i seguenti operatori:

procedure *Inizializza*(**var** *s*: *sequenza*) {prepara s per essere esaminata};
function *FineSequenza*(**var** *s*: *sequenza*): *Boolean* {vale true se non ci sono più elementi di s, altrimenti vale false};
function *Successivo* (**var** *s*: *sequenza*): *T* {restituisce il successivo elemento di s, se esiste};

Supponendo che una successione non possa avere più di 50 oggetti di tipo *T*, si costruisca il tipo *sequenza*, ricorrendo ad un array e si riscrivano i tre operatori su di esso definiti.

*9.2 Analisi di tabelle

Una tabella *tb* di dimensioni fisse, di lunghezza *lunghezzatb*, contiene elementi costituiti da una chiave univoca (una stringa di caratteri) e da informazioni di tipo *T*. Per migliorare l'accesso a questa tabella, si è deciso di ordinare i dati in funzione della lunghezza delle chiavi usate: la chiave di *tb*[*1*] ha la lunghezza minima e quella di *tb*[*lunghezzatb*] la massima. Ciò stabilisce un ordinamento parziale. Se *i* e *j* sono gli indici del primo elemento con una chiave di lunghezza rispettivamente *lunghezzai* e *lunghezzai* + *1*, allora la ricerca dell'indice *k* di un elemento di cui si conosce la chiave (la cui lunghezza è *lunghezzai*) si effettua con una ricerca lineare fra *tb*[*i*] e *tb*[*j* − *1*].
Si costruisca la struttura dati necessaria per questa elaborazione e si scriva la funzione:

function *Ricerca*(*chiave*: *tipochiave*; **var** *k*: *indice*): *Boolean*

dove *tipochiave* = **packed array**[*1..lunghezzachiave*] **of** *char* e *indice* = *1..lunghezzatb*. Questa funzione assume il valore *true* se esiste un elemento con la chiave ricercata (*k* è il suo indice) e *false* in caso contrario. Se una chiave è costituita da *n* caratteri diversi dallo spazio, con *n* < *lunghezzachiave*, allora *n* è la lunghezza usata.
Si scriva una procedura che costruisca le strutture dati necessarie a partire da un file di tipo *text* che contiene *lunghezzatb* elementi, ordinati come nella tabella (per valori crescenti della lunghezza delle chiavi usate).

9.3 Merge

Si scriva la seguente procedura:

procedure *Merge*
(**var** *sorgente1*: **array**[*s1basso..s1alto: integer*] **of** *integer*;
var *sorgente2*: **array**[*s2basso..s2alto: integer*] **of** *integer*;
var *destinazione*: **array**[*dbasso..dalto: integer*] **of** *integer*);

che faccia il merge da *sorgente1* e da *sorgente2*, ordinati in modo crescente, e produca *destinazione*, che deve risultare ordinato.

9.4 Traduzione di un messaggio Morse

Si scriva la seguente procedura:

procedure *TraduzioneMorse* (**var** *inMorse*, *inItaliano*: *text*)

che produca un messaggio in italiano a partire da un messaggio codificato in Morse. Il dizionario Morse è disponibile nel file *CodiceMorse*, di tipo *text*, che ha il seguente formato: ciascun codice Morse, terminato da uno spazio, è seguito dal suo carattere corrispondente e ciascuna coppia (codice Morse, carattere) è separata da uno spazio; tutti i codici sono ordinati per lunghezza crescente e ci sono 39 codici diversi.
Nel messaggio, che compare nel file *inMorse*, uno spazio separa due codici Morse nella stessa parola e due o più spazi delimitano una parola. Il messaggio è chiuso da un codice speciale *FineMessaggio* ('.-.-.'). Il codice più lungo ha lunghezza 8. Se nel messaggio compare un codice illegale, lo si traduce in un punto interrogativo. Si suggerisce di usare l'Esercizio 9.2 per organizzare il dizionario.

9.5 «Paroliamo»

Lo spettacolo televisivo «Paroliamo» richiede che i giocatori costruiscano, in un tempo finito, la parola più lunga possibile usando un insieme di lettere [*numerolettere* (= *8*)], scelte a caso. Questa parola deve far parte di un dato dizionario di riferimento. Usando una tabella che contiene tutte le parole del dizionario di lunghezza minore di *numerolettere*, organizzate come nell'Esercizio 9.2, si scriva la seguente procedura:

procedure *Paroliamo*(*insiemedilettere*: *parola*; **var** *LaPiuLunga*: *parola*)

con *parola* = **packed array** [*1*.. *numerolettere*] **of** *char*.

10
Istruzioni ripetitive

Di solito la stessa azione (semplice o composta) deve essere eseguita per ogni elemento di un particolare insieme di oggetti (specialmente per ogni elemento di un array) o per ogni valore in un sottoinsieme con un numero di elementi finito. In molti casi, inoltre, l'ordine nel quale questi elementi sono manipolati e il modo col quale è selezionato l'elemento successivo sono irrilevanti. Le due istruzioni iterative, introdotte nel Capitolo 6, non sono completamente adeguate, in quanto non enfatizzano abbastanza il fatto importante che lo stesso lavoro è fatto per tutti gli oggetti di un certo gruppo; essi danno troppa importanza all'ordine in cui questi oggetti sono processati e non vanno del tutto bene in certe circostanze poco comuni.
Per esempio, se una certa azione $A(x)$ deve essere eseguita su tutti gli elementi di un certo array a di tipo **array**[*m..n*] **of** *T*, si deve dichiarare una variabile ausiliaria i di tipo *m..succ*(*n*) (che non è una costruzione legale in Pascal), e si deve usare il costrutto

$i := m$; **repeat** $A(a[i])$; $i := succ(i)$ **until** $i > n$

Si noti che i deve essere dichiarato in un campo che abbia un elemento in più del tipo dell'indice di a. In alcune situazioni ciò sarebbe impossibile — per esempio, se questo tipo indice fosse un tipo scalare non ottenuto a partire da un tipo ospite, come nell'esempio seguente:

```
type colore = (viola, blu, verde, giallo, arancio, rosso);
var tabella = array[colore] of qualchetipo;
    indice: colore;
begin {...}
    {si vuole richiedere l'azione A per tutti gli elementi della tabella; non si può inizializzare l'indice con un valore inferiore a viola, che non esiste, e quindi bisogna trattare in modo speciale il primo elemento dell'array}
    indice := viola; A(tabella[indice]);
```

```
while indice < rosso do
begin indice := succ(indice);
   A(tabella[indice])
end {indice adesso vale rosso}
```

10.1 L'ISTRUZIONE for

Un nuovo costrutto linguistico è perciò molto utile per esprimere le istruzioni ripetitive. Esso è costituito dall'istruzione **for**, che esiste in quasi tutti i linguaggi di programmazione e che costituisce il solo modo per esprimere le ripetizioni nella maggior parte di essi, specialmente in Fortran e BASIC. In BASIC il costrutto corrispondente inizia con l'istruzione FOR <variabile> = <espressione> TO <espressione> STEP <espressione> e finisce con l'istruzione NEXT <variabile>. In Fortran è l'istruzione DO <label> <variabile> = <espressione> <espressione> <espressione>, e le istruzioni controllate finiscono all'istruzione che ha la label corrispondente (molto spesso un CONTINUE). In Cobol, è il VARYING...FROM...BY...UNTIL..., forma del verbo generale PERFORM. In PL/1, è una variabile dell'istruzione DO, della forma DO <variabile> = <espressione> BY <espressione> TO <espressione>, con la possibilità di omettere alcune parti aggiungendo una condizione ed elencando più di una di tali specificazioni. Paragonata con tutti questi linguaggi, l'istruzione **for** del Pascal è più semplice e più generale, ma leggermente meno potente, poiché non si può scegliere il passo della progressione.

Sintassi

La sintassi dell'istruzione **for** è illustrata nella Figura 10.1. Poiché non ci sono metaparentesi per chiudere questo costrutto linguistico, è necessario usare un'istruzione composta se l'azione controllata è composita. La variabile di controllo deve essere una variabile semplice di un tipo ordinale, che viene dichiarata localmente nel blocco in cui si trova il costrutto linguistico. Le due espressioni devono essere compatibili in un'istruzione di assegnamento con il tipo della variabile di controllo. L'istruzione controllata non deve assegnare un valore alla variabile di controllo, direttamente o indirettamente (per esempio, chiamando una procedura). Tutte queste restrizioni sono imposte per consentire un'im-

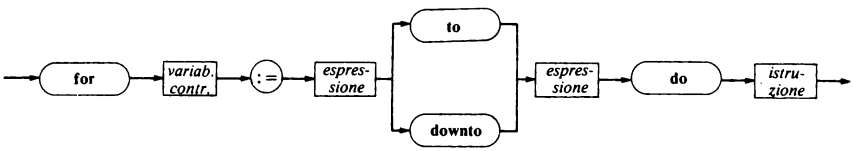


Figura 10.1 Diagramma sintattico di un'istruzione **for**

plementazione efficiente della struttura di controllo e per assicurare che il significato apparente del costrutto sia il suo significato reale.

Semantica

Data l'istruzione **for**:

for *variabile* := *espressione1* **to** *espressione2* **do** *istruzione*(*variabile*)
se {(variabile = espressione1) ∧ P} *istruzione*(*variabile*) {Q(espressione1)}
e {Q(pred(x))} *istruzione*(*variabile*) {Q(x)}
per ogni x tale che *espressione1* $< x \leq$ *espressione2*
allora {(espressione1 ≤ espressione2) ∧ P} istruzione **for** {Q(espressione2)}
e {(espressione1 > espressione2) ∧ P} istruzione **for** {P}

Un'identica regola di verifica si può ottenere sostituendo **downto**, *succ*, >, ≥, < rispettivamente a **to**, *pred*, <, ≤, > nella regola riportata sopra.
Queste due regole sono davvero complesse e si può fornire una spiegazione più semplice dell'istruzione **for** per mezzo di uno schema di programma equivalente. L'istruzione

for *v* := *e1* **to** *e2* **do** *corpo*

è equivalente a

```
begin
   temp1 := e1; temp2 := e2;
   if temp1 ≤ temp2 then
   begin v := temp1;
      corpo;
      while v ≠ temp2 do
      begin v := succ(v);
         corpo
      end
   end
end
```

Uno schema di programma simile può essere ottenuto sostituendo **downto**, ≥ e *pred* rispettivamente a **to**, ≤ e *succ* nello schema sopra riportato. In entrambi i casi *temp1* e *temp2* sono due variabili ausiliarie, di un tipo adatto che non appare da nessun'altra parte nel programma.
Le conseguenze importanti delle descrizioni semantiche precedenti sono le seguenti:

1. Le espressioni limite dell'intervallo sul quale è definita la variabile di controllo sono valutate prima di entrare nel ciclo; cambiando uno qualsiasi di

questi valori durante l'esecuzione del ciclo, non si ha alcun effetto sul numero delle ripetizioni.

2. Se l'intervallo è vuoto (*espressione1* > *espressione2*, nel caso in cui ci sia un **to**, o *espressione1* < *espressione2* nel caso in cui ci sia un **downto**) il corpo non è affatto eseguito, in contrasto per esempio con il Fortran, dove il corpo di un ciclo DO è sempre eseguito almeno una volta.

3. Il valore dell'incremento (o del decremento) della variabile di controllo non può essere scelto: per una variabile di controllo di tipo intero è 1 oppure −1; ciò è una conseguenza del fatto che la variabile di controllo può anche essere un carattere o un tipo scalare, caratteristica molto utile, come si vedrà in seguito.

4. Il numero di ripetizioni del corpo di un'istruzione **for** può sempre essere calcolato prima di entrare nel ciclo, in contrasto con le due istruzioni iterative (**while** e **repeat**), nelle quali la terminazione del ciclo viene controllata nell'istruzione ripetuta e non è assolutamente garantita se la logica dell'istruzione è errata.

5. Lo schema del programma è soltanto un modo per descrivere la semantica dell'istruzione **for** e non implica che essa debba venir implementata in tal modo. Infatti questo schema non mostra un'importante ulteriore restrizione: si suppone che il valore della variabile di controllo sia indefinito alla fine del ciclo, senza obbligare ad usare alcun metodo implementativo. Perciò sarebbe un errore usare *v* alla fine della precedente istruzione **for**, senza effettuare prima un'assegnazione.

ESEMPIO 10.1: PRODOTTO DI MATRICI (VERSIONE FINALE)

```
procedure ProdottoDiMatrici
      (var a: array[pbassoa..paltoa: integer; sbassoa..saltoa: integer]
         of real;
       var b: array[pbassob..paltob: integer; sbassob..saltob: integer]
         of real;
       var c: array[pbassoc..paltoc: integer; sbassoc..saltoc: integer] of real
      )
   {questa procedura calcola il prodotto matriciale fra a[m x p] e b[p x n] in
      c[m x n]; tutti gli indici partono da 1; se gli indici dei parametri attuali non
      sono del tipo corretto, viene chiamata la procedura globale Errore e c rimane
      invariato};
   var m, n, p, i, j, k: 1..maxint;
      somma: real;
begin {ProdottoDiMatrici}
   if (pbassoa ≠ 1) or (sbassoa ≠ 1) or (pbassob ≠ 1) or (sbassob ≠ 1)
      or (pbassoc ≠ 1) or (sbassoc ≠ 1)
```

```
    then Errore else
    begin m := paltoa; n := saltoa; p := saltob;
      for i := 1 to m do
        for j := 1 to n do
        begin somma := 0;
          for k := 1 to p do
            somma := somma+a[i,k]*b[k,j];
          c[i,j] := somma
        end
    end
end {ProdottoDiMatrici}
```

COMMENTI

1. Non è fornita alcuna dichiarazione per assicurare la correttezza della procedura precedente. Infatti, sebbene la sua regola di verifica sia complicata, l'istruzione **for** è così naturale in un caso come questo che non è necessario alcun commento nel programma.
2. Questa procedura è un'applicazione tipica dell'istruzione **for**: l'azione ripetuta deve essere fatta una volta per ogni valore nell'intervallo, che viene esso stesso fissato prima di entrare nel ciclo; inoltre, è irrilevante l'ordine in cui sono fatte le ripetizioni, e le varianti **downto** potrebbero essere usate per le tre istruzioni **for** annidate. L'istruzione **for** è assai utile per tali situazioni, ma non dovrebbe mai essere usata in altre circostanze. Il fatto che in Fortran o in BASIC tutti i cicli implichino la variazione di qualche variabile intera può nascondere completamente la semplicità e la naturalezza di algoritmi che non mostrano una tale proprietà.

ESEMPIO 10.2: FREQUENZA DEI DIAGRAMMI

```
program Diagrammi(input, output)
{il file di tipo text in input viene letto e copiato in output, seguito dalla frequenza di
   tutti i diagrammi (coppie consecutive di lettere) trovati; queste informazioni pos-
   sono essere molto utili in crittografia};
   type naturali = 0..maxint; lettera = 'a'..'z';
   var contatore: array[lettera, lettera] of naturali;
      prima, seconda: lettera {indici dell'array contatore};
      ch: char {il carattere corrente};
   function EUnaLettera (ch: char): Boolean;... {si veda l'Esempio 9.3};
begin {programma Diagrammi}
   for prima := 'a' to 'z' do
      for seconda := 'a' to 'z' do {inizializza contatore}
         contatore[prima, seconda] := 0;
```

```
    while not eof(input) do {sempre il solito schema già visto}
    begin
      while not eoln(input) do {elabora una linea}
      begin read (input, ch); write(output, ch);
        if EUnaLettera(ch) then
          if EUnaLettera(input↑) then {si è trovato un diagramma}
            contatore[ch, input↑] := contatore[ch, input↑] + 1
      end {fine di una linea};
      writeln(output); readln(input)
    end {while ¬eof (input)};
    page(output); writeln(output, 'Frequenza dei diagrammi');
    writeln(output);
    for prima := 'a' to 'z' do
    begin {elabora una serie di diagrammi}
      for seconda := 'a' to 'z' do
        if contatore[prima, seconda] ≠ 0 then
          write(output, prima, seconda, contatore[prima, seconda]);
      writeln(output) {una linea vuota di separazione}
    end
  end {Diagrammi}.
```

COMMENTI

1. Ancora una volta si incontra il problema della mancanza di standardizzazione degli insiemi dei caratteri. Se si usa la codifica ISO, l'array *contatore* ha esattamente $26 \times 26 = 676$ elementi e la funzione *EUnaLettera* può essere più semplice di quella dell'Esempio 9.3: usando però la codifica EBCDIC, ci sono 924 elementi in più, che sono completamente inutili, dal momento che la cardinalità dell'intervallo *'a'..'z'* è 40 in tale codice ($40 \times 40 - 676 = 924$). In circostanze diverse potrebbe essere utile determinare una trasformazione fra lettere e interi da 1 a 26 e calcolare questa trasformazione con un **array**[*'a'..'z'*] **of** *0..26*, che dà *0* per tutti i caratteri che non sono lettere. La funzione *EUnaLettera* perciò sarebbe inutile.
2. Si è evitato l'uso di *EUnaLettera* nel doppio ciclo alla fine del programma perché le coppie di lettere consecutive che includono altri caratteri non sono calcolate affatto e, di conseguenza, la loro ricorrenza rimane zero. Se si dovessero scrivere tutte le possibili coppie di lettere, anche quelle che non appaiono nel testo (per esempio, *'tq'* o *'zb'* in un testo scritto in un linguaggio naturale) questo trucco non funzionerebbe.
3. Naturalmente il punto più interessante del programma precedente è che i valori dei caratteri possono essere usati in un'istruzione **for** così come per gli indici di array. Si noti anche il modo in cui *input*↑ viene usato come un carattere di «look-ahead».

ESEMPIO 10.3: ORDINAMENTO DI SHELL

```
procedure OrdinamentoDiShell (var tabella: array[basso..alto: integer] of T)
  {questa procedura riordina l'array tabella usando l'algoritmo di Shell; T è un da-
     to tipo, su cui è definito il predicato MinoreDi: la funzione MinoreDi (x, y:
     T): Boolean assume il valore true se x precede y in una qualche relazione
     d'ordine; nella tabella, elementi con indici negativi vengono usati come senti-
     nelle; l'estremo inferiore del parametro attuale è sempre 1; con i quattro passi
     utilizzati in questa procedura, l'estremo inferiore deve essere −8 o meno; l'al-
     goritmo di Shell non è il migliore possibile e ne verrà descritto uno più poten-
     te nel Capitolo 14};
  const ordinemassimo = 4 {numero di differenti incrementi};
  var oggetto: T;
     ordine: 1..ordinemassimo;
     incremento: array[1..ordinemassimo] of integer;
     i, j, passo, sentinella: integer;
begin {OrdinamentoDiShell}
  incremento[1] := 9; incremento[2] := 5;
  incremento[3] := 3; incremento[4] := 1;
  for ordine := 1 to ordinemassimo do {usa incrementi discendenti}
  begin passo := incremento[ordine]; sentinella := −passo
     {min(sentinella) = −9};
     for i := passo+1 to alto do
     begin oggetto := tabella[i]; j := i−passo;
        if sentinella = 0 then sentinella := −passo;
        sentinella := sentinella+1 {min(sentinella) = −8};
        tabella[sentinella] := oggetto {il più basso deve essere ≤ −8}
        {la sentinella è posizionata};
        while MinoreDi(oggetto, tabella[j]) do
        begin {inserzione}
           tabella[j+passo] := tabella [j];
           j := j−passo
        end;
        tabella[j+passo] := oggetto
     end {ciclo su i}
  end {ciclo su ordine}
end {OrdinamentoDiShell}
```

ESERCIZI

*10.1 Valutazione di un polinomio

Dato un polinomio di grado n

$$P = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

si scriva la funzione

function *ValorePolinomio*(**var** *a*: **array**[*li..ls*: *integer*] **of** *real*; *x*: *real*): *real*

che calcola il valore di P in x usando la tabella a di coefficienti. Si suggerisce il noto metodo di Horner.

10.2 Concordanza di un testo (continua)

Nell'Esempio 9.5 numerose istruzioni iterative potrebbero essere sostituite con istruzioni ripetitive. Si riscriva questo programma usando istruzioni **for** dove è possibile e modificando la stampa dei risultati, in modo che le parole appaiano su più colonne. Ogni colonna ha una lunghezza pari a *lunghparola* ed è separata dalle colonne vicine da due spazi. Una costante *lunghlinea* dà la massima lunghezza della linea di output.

10.3 Master Mind

Il gioco seguente è una versione semplificata del noto Master Mind. Due giocatori hanno un ugual numero di pioli pari a *npioli*, identici per ogni giocatore; i pioli sono di vari colori, il cui numero è *maxcolori*. Il giocatore chiamato A sistema un numero di pioli pari a *maxpioli* su di un apposito supporto (nascosto al secondo giocatore B). Lo scopo di B è di dedurre il colore e la posizione dei pioli di A. Egli procede per ipotesi successive, sistemando alcuni pioli sul suo supporto che è simile a quello di A, ma che non è nascosto. Dopo ogni tentativo, A risponde nel modo seguente:

a) di fronte ad ogni piolo completamente corretto per posizione e colore mette un segno nero;
b) di fronte ad ogni piolo del colore esatto, ma errato per posizione, mette un segno bianco;
c) le altre posizioni restano vuote.

Tenendo conto delle risposte di A, B deve dedurre la soluzione corretta, in un numero minimo di tentativi.
Si scriva un programma che legga il gioco di A, costruisca la serie di pioli di B e la prima mossa di B, e simuli la partita completa fino a che B non ha scoperto il codice segreto di A. Si usino le seguenti definizioni di tipo e di costanti:

```
const maxpioli = 4; maxcolori = 6;
   npioli = 24 {npioli = maxpioli * maxcolori};
type piolo = (rosso, verde, giallo, blu, marrone, arancio);
   risposta = (bianco, nero, nessuna);
   gioco = array[1..maxpioli] of piolo {sia per A che per B};
   ipioli = array[rosso..arancio] of 0..maxpioli {per B};
```

11
I record

11.1 PRODOTTO CARTESIANO E RECORD SEMPLICI

Il mezzo più semplice e generale per strutturare un oggetto è quello di raggruppare parecchi oggetti di qualsiasi tipo (possibilmente oggetti strutturati) e considerare il gruppo come oggetto unico. Per esempio, un numero complesso è una coppia ordinata di reali, rispettivamente la sua parte reale e la sua parte immaginaria. Le coordinate polari in un piano sono un'altra coppia di reali, che rappresentano un modulo e una caratteristica. Una persona può essere descritta da diversi dati: nome e cognome (sequenze di caratteri), data di nascita, (un oggetto strutturato), sesso, stato civile, ecc.
Tutto ciò corrisponde a quella struttura astratta che i matematici chiamano *prodotto cartesiano*, di cui ora parleremo. Dato un tipo prodotto *P* dei tipi *T1*, *T2*, ..., *Tn*, un oggetto di tipo *P* ha *n* elementi di tipo *T1*, *T2*, ..., *Tn*, rispettivamente. Si definisce *p*<*x1*, *x2*, ..., *xn*> un oggetto di tipo *P* i cui elementi sono *x1* (di tipo *T1*), *x2* (di tipo *T2*), ... *xn* (di tipo *Tn*). Tutti gli oggetti di tipo *P* possono essere definiti usando questo costrutto.
Per fare riferimento a qualsiasi elemento di un prodotto cartesiano, dobbiamo dare un nome a questi elementi, poiché il loro ordine non ha alcun significato. Inoltre questi elementi possono essere di tipi differenti e se si usasse la stessa notazione degli array, non si potrebbe sapere il tipo di *p*[*i*] (con *p* di tipo *P*), senza calcolare *i*. Come conseguenza, la definizione di un tipo *P* deve includere i nomi dei suoi elementi *e1*, *e2*, ..., *en*. Se *x* = *p*<*x1*, ..., *xn*> allora *x.e1* = *x1*, *x.e2* = *x2*, ..., *x.en* = *xn*. La notazione col punto *x.ei* costituisce il *selettore di un elemento* nel prodotto cartesiano.
Esiste lo stesso problema visto per le trasformazioni; avremo cioè bisogno di una notazione per la modifica selettiva di uno degli elementi di un record. Viene usata la stessa soluzione; si può usare la notazione di selezione di un elemento di un record nella parte sinistra di un'istruzione di assegnamento. In modo più formale, ricorrendo allo stesso tipo e agli stessi nomi introdotti prima,

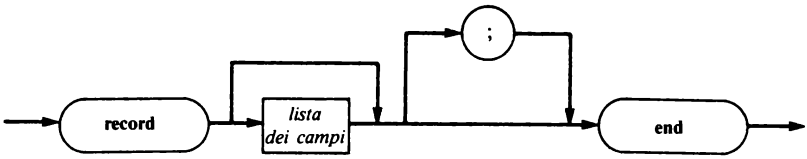


Figura 11.1 Diagramma sintattico di un tipo record

x.e1 := *x1* è equivalente a *x* := *p*<*x1*, *x.e2* ,..., *x.en*>; *x.e2* := *x2* è equivalente a *x* := *p*<*x.e1*, *x2*,..., *x.en*> e così via.
Le regole precedenti sono sufficienti a definire il prodotto cartesiano come struttura astratta, come abbiamo fatto per le successioni nel Capitolo 7, e per le trasformazioni nel Capitolo 9. Un'importante differenza, tuttavia, è che il prodotto cartesiano è una struttura così semplice che può essere implementata in modo lineare. La realizzazione di questo concetto astratto non richiede particolari restrizioni. Solo per ragioni storiche, il nome effettivamente utilizzato non è «prodotto cartesiano», ma *record*.
Una struttura a record è una struttura ad accesso casuale come l'array, con un numero finito e determinato di elementi chiamati *campi*, ciascuno dei quali è accessibile separatamente. Contrariamente a quanto accade per gli array, elementi diversi possono essere di tipi differenti, il che implica che devono avere ciascuno un nome prestabilito, cosicché il tipo di un elemento selezionato possa essere già noto durante la compilazione; il nome di un campo di un record non può essere valutato. Un tipo record in Pascal ha la sintassi mostrata nella Figura 11.1. Per il momento supporremo che la lista dei campi sia semplicemente una lista di sezioni di record, separate da un punto e virgola. La definizione reale, che è molto più complessa, sarà data nel paragrafo 11.3. La sezione di definizione di un record assomiglia alla lista di dichiarazioni di variabili; la sua sintassi è mostrata nella Figura 11.2. La sintassi di un tipo record è, in qualche modo, permissiva, poiché consente un punto e virgola opzionale prima dell'**end** e la lista dei campi può mancare del tutto. Il tipo la cui descrizione è **record end** è semplicemente un record vuoto di scarsissima utilità (si veda però il paragrafo 11.3).

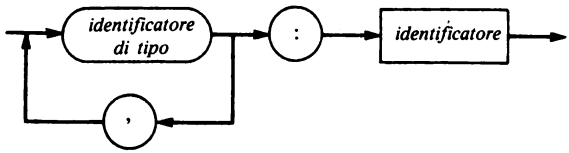


Figura 11.2 Diagramma sintattico di una sezione di record

ESEMPIO 11.1: DEFINIZIONE DI TIPI DI RECORD

```
type complessi = record partereale, parteimmaginaria: real end;
   stringa = packed array[1..lunghezzastringa] of char;
   tiposesso = (maschio, femmina) {definito in modo tale che maschio
      < femmina};
   tipodata =
      record giorno: 1..31; mese: 1..12; anno: 1800..2000 end;
   tiponome =
      record nomeproprio, cognome: stringa; secondonome: char end;
   tipogenitore = (padre, madre);
   statocivile = record
      nome: tiponome;
      datadinascita: tipodata;
      genitori: array[tipogenitore] of tipodata;
      numbambini: 0..maxbambini;
      bambino: array[1..maxbambini] of record
         datadinascita: tipodata
         sesso: tiposesso;
         nomeproprio: stringa end;
      sesso: tiposesso end {tipo statocivile};
```

COMMENTI

1. La disposizione nel programma dei vari elementi costituenti la definizione di un record (parole chiave, nome dei campi, ecc.) dipende solo dal particolare contesto e dai gusti personali del programmatore, per cui non si possono stabilire delle regole generali. Si può mettere l'intera definizione su una linea, se ci sta, e perfino sulla stessa linea di definizione del nome. D'altra parte si può ricorrere alla marginatura, inventando una gran quantità di regole su come iniziare e finire ciascuna linea. La sola cosa importante è che la definizione sia leggibile e facile da capire.
   Un tipo che è usato una sola volta può essere definito in modo anonimo, come si è già visto nel Capitolo 2. In questo caso si sarebbe potuto fare, ad esempio, per il *tipogenitore*; non sarebbe comparsa alcuna definizione del tipo stesso e l'identificatore di tipo del campo *genitori* nella definizione del tipo di *statocivile* sarebbe stato un **array** [(*padre, madre*)] **of** *tipodata*. Tuttavia, non si sarebbe più potuta dichiarare una variabile di *tipogenitore*. In questo modo, tutti i riferimenti a un genitore o all'altro sarebbero stati fatti soltanto con un indice costante, rendendo l'array stesso quasi inutile. Generalmente, programmando in Pascal, si devono inventare molti nomi differenti, con molta attenzione e perspicacia. Se il tipo di un nome è chiamato *nome*, nessun nome può essere più chiamato «nome» e ciò può infastidire. Questo spiega la frequenza di nomi di tipo che contengono la parola *tipo*,

ma rivela un difetto importante in quei compilatori, non conformi allo Standard, che riconoscono soltanto i primi *n* caratteri di un identificatore, generalmente con *n* uguale a 8. Normalmente, in inglese, gli identificatori di tipo terminano con l'attributo «tipo»; perciò, anziché avere un *tipogenitore* si avrebbe un *genitoretipo* che non sarebbe distinguibile da *genitore*, nell'ipotesi limitativa appena considerata. È un vero peccato che, per ragioni di portabilità, molti programmatori che sono abituati a un compilatore conforme allo Standard debbano rinunciare ad usare pienamente le sue capacità da questo punto di vista.

3. Il tipo *statocivile* mostra due modi differenti per definire i record gerarchicamente strutturati, cioè quei record con alcuni campi di tipo record. I campi *nome* e *datadinascita* sono record il cui tipo è anonimo (e che contengono un altro campo di tipo record).
4. Riguardo al numero possibile di bambini, il tipo *statocivile* non è la miglior implementazione possibile del concetto che si deve rappresentare. Se la costante *maxbambini* è piccola (3 o 4), alcune situazioni non si possono rappresentare. Se questa costante è grande (15 o 20), è sprecata molta memoria per quasi tutti gli oggetti di questo tipo. Una soluzione migliore verrà fornita nel Capitolo 13.
5. Tutte le combinazioni dei valori di campo sono legali in un tipo di record dato, anche quelle concettualmente assurde. Per esempio, il 29 febbraio esiste per tutti gli anni possibili nel tipo *tipodata* e perfino il 31 febbraio è ammesso (si veda l'Esempio 11.3).

11.2 USO DEI RECORD

Un *designatore di campo*, che è un altro caso di variabile, ha una sintassi molto semplice (si veda la Figura 11.3). L'indicatore di variabile di tipo record è una variabile di qualche record e l'identificatore di campo deve essere l'identificatore di un campo che compaia in questo record. Infatti gli identificatori di campo sono dichiarati nella definizione di un tipo record, ma sono locali ad esso, senza creare conflitti con gli altri identificatori dichiarati all'esterno. Se parecchie definizioni di record sono annidate, si può accedere ad un campo interno usando una catena di designatori di campo.

ESEMPIO 11.2: VARIABILI DI TIPO RECORD E DESIGNATORI DI CAMPO

```
{questo esempio fa uso delle definizioni di record dell'Esempio 11.1}
var data: tipodata; nome: tiponome; nomeproprio: stringa;
    padre, madre, figlio, figlia: statocivile;
    parrocchia: array[1..maxfamiglie] of statocivile;
    i: 1.. maxfamiglie; j: 0..maxbambini;
    annocorrente: 1800..2000;
```

```
begin
...
   for i := 1 to maxfamiglie do {per ciascun capofamiglia}
      for j := 1 to parrocchia[i] numbambini do {per ciascun bambino}
         if parrocchia[i].bambino[j].nomeproprio = Clementina
         then writeln(annocorrente-parrocchia[i].datadinascita.anno)
{questa istruzione stampa l'età corrente di ciascun capofamiglia di una parrocchia che
   ha un bambino il cui nome ha lo stesso valore di Clementina, che è una variabile
   o una costante che probabilmente contiene la stringa 'Clementina', con un nume-
   ro di spazi sufficienti a soddisfare il tipo stringa};
{parrocchia[i].datadinascita.anno si riferisce all'anno di nascita del capofamiglia con-
   siderato;
   parrocchia[i].bambino[j].datadinascita.anno si riferisce all'anno di nascita del
   bambino preso in considerazione}
...
   {seguono istruzioni di assegnamento legali}
   padre := parrocchia[i]; parrocchia[i+1] := madre;
   figlio.datadinascita := madre.bambino[j].datadinascita;
   if padre.bambino[j].sesso = femmina then
      figlia.datadinascita := data
      else figlio.nome := nome
      ...
end
```

COMMENTI

1. La visibilità dei nomi all'interno e all'esterno dei tipi di record segue esattamente le stesse regole delle procedure e delle funzioni. Nel tipo *statocivile*, i campi *datadinascita* e *sesso* appaiono due volte ciascuno, ma senza ambiguità, poiché queste due occorrenze non sono allo stesso livello. In modo simile, l'identificatore *nome* è usato sia come variabile che come nome di un campo.
2. Nessuna abbreviazione è permessa in riferimenti complessi, né esiste alcuna notazione alternativa, anche se non c'è ambiguità (ciò è in netto contrasto con il Cobol e il PL/1). Per esempio, non c'è altro modo di fare riferimento all'anno di nascita del *j*-esimo bambino dell'*i*-esimo capofamiglia della parrocchia che con *parrocchia[i].datadinascita.anno*. L'unico problema è la lunghezza di questo riferimento e il costo del suo calcolo e più avanti verrà presentata una scorciatoia.
3. Come qualsiasi altro oggetto strutturato (tranne i file), un record può essere assegnato come un oggetto unico ad una variabile di record dello stesso ti-

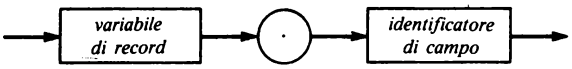


Figura 11.3 Diagramma sintattico di un designatore di campo

po. Tuttavia, le regole di compatibilità di assegnamento dimostrano che questi record devono essere esattamente dello stesso tipo e non, per esempio, di due tipi differenti che abbiano casualmente la stessa struttura e gli stessi nomi e tipi per tutti i loro campi.

È il momento di un breve confronto con il Cobol e il PL/1 sul concetto e la struttura di record. È lasciato come esercizio per il lettore tradurre in uno di questi linguaggi la definizione del tipo *statocivile*. Se non si ritengono importanti i dettagli sintattici, la principale differenza è che la gerarchia di strutture di record è indicata in Pascal con l'annidamento di più definizioni di tipi di record, al posto di numeri di livello. Si è infatti dovuta aggiungere una quantità minima di sintassi al Pascal, grazie alla disponibilità delle definizioni di tipo. Altre differenze sono le seguenti:

a) tutti i riferimenti ai campi di record devono essere completi in Pascal, mentre il Cobol e il PL/1 consentono alcune abbreviazioni se non ci sono ambiguità;
b) in Pascal non c'è «assegnamento per nome»: i tipi di record o sono gli stessi — e l'assegnamento è globale — oppure sono differenti — e allora il campo deve essere assegnato individualmente;
c) non si può specificare in Pascal l'implementazione dettagliata dei campi di un record; si può solo richiedere il compattamento dell'intero record, il che significa che il compilatore deve cercare di risparmiare memoria (se possibile) invece di allinearsi sistematicamente al limite della parola, come quando non si richiede il compattamento.

11.2.1 L'ISTRUZIONE with

Era stata preannunciata una scorciatoia per evitare il costo di complicati designatori di campo, sia per scriverli che per valutarli. Si tratta dell'istruzione **with**, la cui sintassi è mostrata nella Figura 11.4. La sua semantica è spiegata dalla seguente regola di verifica:

Se x è di tipo **record** *f1*: *T1*; *f2*: *T2*; ...; *fn*: *Tn* **end**
allora l'istruzione **with** x **do** è equivalente a

istruzione $\begin{matrix} f^1, f^2, \ldots, f^n \\ x.f_1, x.f_2, \ldots, x.f_n \end{matrix}$

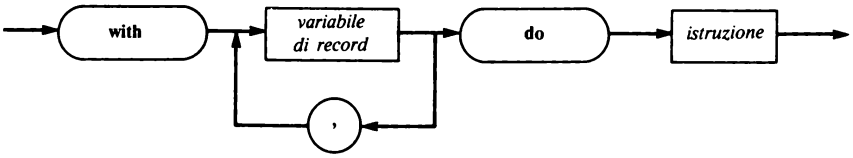


Figura 11.4 Diagramma sintattico di un'istruzione **with**

Questo significa che l'istruzione **with** «apre» la definizione di record e rende tutti i nomi di campo della definizione di record accessibili direttamente all'interno dell'istruzione controllata, come se essi fossero dichiarati al suo inizio. I possibili conflitti di nome sono risolti allo stesso modo dei conflitti nelle procedure o funzioni annidate. Per esempio, nell'istruzione controllata da **with** *parrocchia*[*i*].*bambino*[*j*] **do**..., la variabile *nome* non è piu accessibile a causa del conflitto con il campo omonimo, e *datadinascita* si riferisce a *parrocchia*[*i*].*bambino*[*j*].*datadinascita*; tuttavia, *parrocchia*[*i*]. *datadinascita* resta accessibile usando il suo designatore completo.
Più variabili di tipo record che compaiono dopo un **with** hanno lo stesso significato di più istruzioni **with** annidate: **with** *x1*, *x2*, ..., *xm* **do** ... è equivalente a

```
with x1 do
   with x2 do
      ...
      with xm do
```

Per esempio, nell'istruzione controllata da

```
with parrocchia[i], bambino[j], do ...
```

sesso si riferisce al campo di questo nome per il *j*-esimo bambino e *nome* si riferisce al nome del *j*-esimo capofamiglia della parrocchia.
Si deve fare un'importante annotazione: è proibito — e sarebbe un errore — modificare la variabile del costrutto **with** all'interno dell'istruzione controllata dal costrutto stesso. Per esempio, all'interno dell'istruzione controllata da **with** *parrocchia*[*i*].*bambino*[*j*] **do**, sarebbe un errore modificare *i* o *j* direttamente (per mezzo di un'istruzione di assegnamento) o indirettamente (chiamando una procedura che modifichi *i* o *j*). Questa restrizione viene fatta per ragioni di chiarezza ed efficienza.

ESEMPIO 11.3: VERIFICA DI UNA DATA

```
var oggi: tipodata {tipo definito nell'Esempio 11.1}
   feb: 28..29;
begin ...
      with oggi do
         case mese of
         4, 6, 9, 11: if giorno > 30 then Errore;
         2: begin {è un anno bisestile?}
               if (anno mod 4 = 0) and (anno mod 100 ≠ 0)
               or (anno mod 400=0) then feb := 29
            else feb := 28;
            if giorno > feb then Errore
```

```
        end {febbraio};
    1, 3, 5, 7, 8, 10, 12: {sempre legale}
end
```

COMMENTI

In questo esempio l'istruzione **with** serve soltanto come abbreviazione e non ha alcuna influenza sul lavoro fatto dal programma, poiché un riferimento al campo di un record può essere valutato dal compilatore e, conseguentemente, non ha alcun costo a tempo di esecuzione.

ESEMPIO 11.4

```
{di seguito vengono riscritte le due istruzioni for annidate presentate nell'Esempio
        11.2}
for i := 1 to maxfamiglie do
    with parrocchia[i] do
        for j := 1 to numbambini do
            if bambino[j].nomeproprio = Clementina then
                writeln(annocorrente - datadinascita.anno)
```

COMMENTI

Poiché i limiti di un ciclo **for** sono valutati una volta soltanto, l'istruzione **with** non sarebbe utile se il riferimento a *parrocchia*[*i*] (che richiede una valutazione) non apparisse all'interno dell'istruzione controllata, oltre che come limite superiore del ciclo su *j*.

11.3 UNIONE DI TIPI E RECORD CON VARIANTI

In Pascal tutti gli oggetti devono avere un tipo definito, ma qualche volta è utile considerare un tipo come l'unione di parecchi altri tipi, o considerare questi altri tipi come varianti del primo. Per esempio, un determinato tipo *coordinate* può rappresentare, a seconda delle circostanze, coordinate cartesiane o coordinate polari. Le operazioni sugli elementi di un tipo devono riconoscere la particolare variante, mentre le operazioni sull'oggetto nel suo insieme possono ignorarla. Per indicare quale particolare variante è stata scelta per un certo oggetto di un determinato tipo, quest'oggetto deve contenere come elemento un indicatore (un campo) che in Pascal è chiamato *discriminante di variante* (*tag*). A causa di questo campo è piuttosto naturale considerare il concetto di un tipo con variante come una generalizzazione della struttura di record. La sintassi

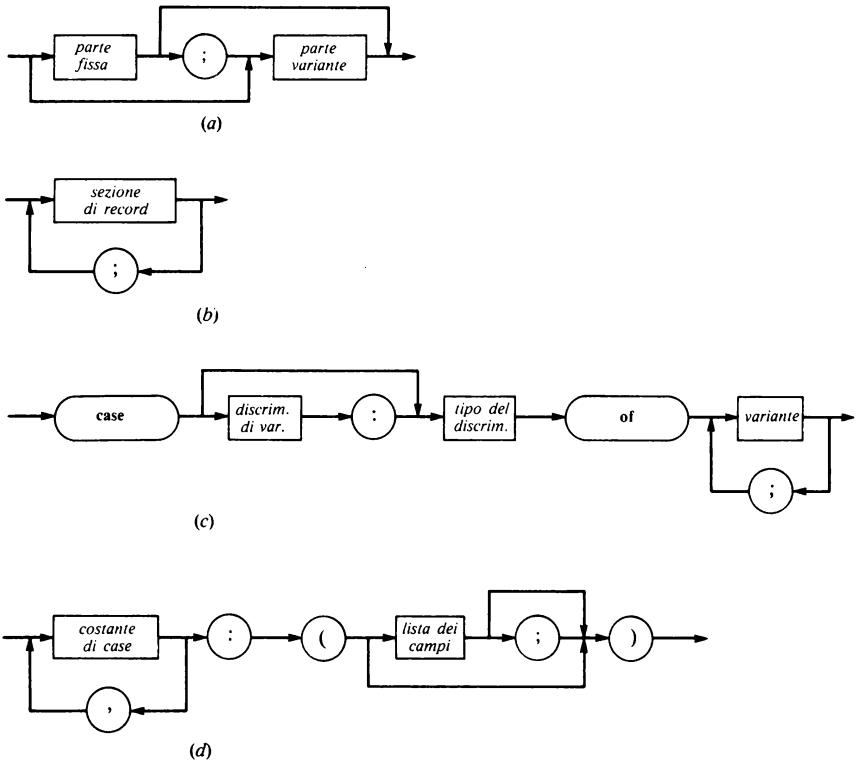


Figura 11.5 Diagramma sintattico di una lista di campi: (*a*) lista dei campi; (*b*) parte fissa; (*c*) parte variante; (*d*) variante

completa di un tipo di record è perciò molto più complicata di quella provvisoria illustrata nel paragrafo 11.1 e la Figura 11.5 dà la sintassi della lista dei campi. In queste regole, il discriminante di variante è un identificatore, il tipo del discriminante è un identificatore di tipo ordinale e la costante di **case** è una costante dello stesso tipo del discriminante. Le similitudini fra queste sintassi e quella della istruzione **case** non devono far dimenticare le differenze. Ecco i commenti più importanti che si devono fare:

a) la parte invariante di un record — che può essere considerata come contenente quei campi presenti in tutte le varianti — deve apparire all'inizio del record, se esiste;
b) c'è soltanto una parte variante in un dato record, ma una data variante è di fatto una lista di campi, che può contenere ulteriori parti varianti. Come conseguenza, si può costruire una gerarchia di varianti;

c) la lista dei campi di una variante è delimitata da parentesi come mezzo per suggerire che non inizia un altro livello di dichiarazione: gli identificatori di campo che compaiono in una variante sono allo stesso livello di quelli che compaiono nella parte invariante. Come conseguenza, tutti gli identificatori di campo devono essere distinti, anche se compaiono in varianti differenti (allo stesso livello);
d) nella gerarchia delle varianti, la variante a livello più alto termina con il simbolo **end** che chiude la definizione del tipo di record; una parte variante a livello più basso termina con la parentesi destra che chiude la variante in cui è annidata;
e) è possibile che più di un valore del discriminante di variante sia identico alla stessa variante. Inoltre, tutti i valori possibili del discriminante devono comparire come costanti in un'istruzione **case**;
f) una variante è descritta sintatticamente come una lista di costanti di un'istruzione **case**, seguita da una lista di campi racchiusa fra parentesi. Tuttavia, la lista di campi può essere omessa, come nella definizione di un record completo (si veda il paragrafo 11.1). In quel caso, la possibilità di una variante vuota è utile ed è usata frequentemente: costituisce una giustificazione all'esistenza di record vuoti, per ortogonalità;
g) il tipo del discriminante di variante non può essere anonimo: deve essere definito e gli deve essere dato un nome;
h) il discriminante stesso può essere omesso. La definizione ufficiale del linguaggio suppone che, in tal caso, tutto vada come se fosse presente un discriminante nascosto; l'implementazione deve essere cioè in grado di stabilire a tempo di esecuzione quale discriminante è stato effettivamente scelto fra tutte le varianti possibili di un record con varianti. Contrariamente a questa definizione, molte implementazioni sfruttano l'essenza del discriminante per consentire conversioni «pericolose», cioè per considerare una zona di memoria da punti di vista differenti, in modo del tutto simile all'opzione EQUIVALENCE in Fortran o RENAMES in PL/1.

ESEMPIO 11.5: DEFINIZIONE DI TIPI DI RECORD CON VARIANTI

```
type registro = 0..15;
   modo = (immediato, normale, shift, salto);
   ventibit = -1048576..1048575;
   dodicibit = -2048..2047;
   diecibit = -0..1023;
   seibit = -32..31;
   quindicibit = 0..32767;
   unbit = 0..1;
   istruzione = packed record {descrizione di un'istruzione macchina}
      indiretto: Boolean;
      operatore: 0..127;
```

```
    reg1: registro;
    case modo of {nell'istruzione non compare un descrittore di variante}
      immediato: (opimmediato: ventibit);
      normale: (indice, base: registro;
        offset: dodicibit);
      shift: (varianteshift: registro;
        filler: diecibit;
        numshift: seibit);
      salto: (variantesalto: registro;
        fillersalto: unbit;
        indirizzo: quindicibit)
    end {tipo istruzione}
  modocoordinate = (cartesiano, polare);
  coordinate = record
    case modo: modocoordinate of
      cartesiano: (x, y: real);
      polare: (ro, theta: real)
    end {tipo coordinate};
  figura = record {descrizione di una figura geometrica}
    posizione: coordinate;
    case circonferenza: Boolean of
      true: (diametro: real);
      false: (orientamento, lato: real;
        case rettangolare: Boolean of
        true: (larghezza: real);
        false: (angolo1, angolo2: real)
        )
    end {tipo figura};
```

COMMENTI

1. La descrizione del tipo *istruzione* presuppone che il compilatore compatti esattamente quanti più bit sia possibile in una parola, e che il calcolatore ospite abbia parole di trentadue bit. Naturalmente, questo è un esempio di descrizione di tipo non portabile.
2. In questo tipo, i campi *varianteshift* e *variantesalto* devono avere nomi differenti, perché appaiono allo stesso livello. Per evitare questo fatto sarebbero necessarie varianti annidate, come nel tipo *figura*.
3. Non ci devono essere discriminanti di varianti nel tipo *istruzione*, perché non ne esistono nell'istruzione del calcolatore che deve essere descritta. Tuttavia, c'è probabilmente una certa relazione tra il valore dell'operatore e la forma dell'istruzione. La legalità dei riferimenti non è verificata dalla maggior parte dei compilatori.
4. Il tipo *coordinate* è un esempio di record senza parte fissa.

Un tipo di record con varianti descrive numerosi record differenti, tutti uniti in uno stesso tipo. Tuttavia, un oggetto di tale tipo può avere una ed una sola variante per volta, cioè sono definiti solo i campi che appaiono nella parte fissa e nella variante corrente. La variante corrente è determinata dal valore corrente del discriminante di variante, se presente. Quando tale discriminante non c'è, ci si deve ricordare che il discriminante «nascosto» è assegnato implicitamente ogni volta che si fa riferimento a un campo di una particolare variante che non sia quella corrente. Poiché è molto difficile e costoso, per una data implementazione, verificare la correttezza di tutti i riferimenti ai campi delle varianti, i programmatori devono prestare la massima attenzione. Un utile suggerimento è quello di far riferimento a campi di varianti soltanto all'interno di istruzioni **case**, seguendo lo stesso modello della parte variante del record (se possibile). Nei prossimi capitoli verranno dati parecchi esempi.

11.4 STRUTTURE COMPOSITE

I record costituiscono il terzo metodo di strutturazione dei dati in Pascal ad essere descritto, e possono essere usati in combinazione con i due metodi precedenti, con la restrizione (già vista) che i file non possono avere degli elementi che contengono altri file. Per esempio, si possono non soltanto definire ed usare tipi strutturati che sono record con array o campi di record (si veda l'Esempio 11.1), ma anche array o file i cui elementi siano record, che a loro volta possono contenere array o campi di record e così via. Questo paragrafo illustrerà quest'aspetto con alcuni esempi di strutture composite, revisioni di esempi tratti dai capitoli precedenti.

ESEMPIO 11.6: VENDITE MASSIME (MODIFICATO)

```
{questo esempio è già stato proposto nell'Esempio 6.6, ma l'idea astratta di una successione di numeri interi con significati diversi è qui sostituita da un file reale di record con varianti}
type tipoprodotto = 1000..99999 {è una codifica con interi, ma una codifica alfanumerica sarebbe andata altrettanto bene};
  tipomese = 1..12 {non c'è più bisogno di un mese zero};
  tipoanno = 1970..2000;
  {il file da processare è una successione di anni, ciascuno dei quali inizia con un record che dà il numero dell'anno seguito da una successione di mesi; una sequenza di mesi inizia con un record che dà il numero del mese seguito da una sequenza di record di prodotti, ordinati per numero di codice crescente}
  tiporecord = (recanno, recmese, recprodotto);
  tiporecordvendite = packed record
    case categoria: tiporecord of
      recanno: (anno: tipoanno);
      recmese: (mese: tipomese);
```

```
            recprodotto: (prodotto: tipoprodotto;
                qta, ric: integer)
        end;
    filevendite = file of tiporecordvendite;
procedure VenditeMassime
        (var f: filevendite {il file da esaminare};
        y: tipoanno {l'anno da esaminare};
        i: tipoprodotto {il prodotto d'interesse};
        var q, r: integer {massima quantità e riciclo corrispondente};
        var m: tipomese {mese corrispondente}
        );
    var mesecor: tipomese {il mese corrente};
        qtacor, riccor: integer {massima qta e ric corrente};
        stato: (inricerca, trovato, assente, fineanno);
    procedure RicercaAnno
        {inizializza f e si posiziona in testa all'anno d'interesse};
    begin {RicercaAnno}
        reset(f); stato := inricerca;
        repeat {ricerca l'anno d'interesse}
            if eof(f) then stato := fineanno else
            begin
                if f↑.categoria = recanno then
                    if f↑.anno = y then stato := trovato;
                get(f) {passa il record corrente}
            end { ¬eof(f)}
        until stato ≠ inricerca
    end {RicercaAnno};
    procedure RicercaProssimoMese
    {posiziona f in testa alla successione relativa al mese successivo, se esiste}
    begin {RicercaProssimoMese}
        stato := inricerca;
        repeat {ricerca un mese o la fine dell'anno corrente}
            if eof(f) then stato := fineanno else
            with f↑ do
                case categoria of
                    recanno: stato := fineanno;
                    recmese:
                        begin stato := trovato; mesecor := mese;
                            get(f) {passa il record mese}
                        end;
                    recprodotto: get(f)
        end {case}
        until stato ≠ inricerca
    end {RicercaProssimoMese};
    procedure RicercaIlProdotto
```

```
        {posiziona f sul record del mese corrente che si riferisce al prodotto i, se
        esiste};
    begin {RicercaIlProdotto}
        stato := inricerca;
        repeat {ricerca il record del prodotto d'interesse}
            if eof(f) then stato := fineanno else
            with f↑ do
                case categoria of
                    recanno: stato := recanno;
                    recmese: stato := assente;
                    recprodotto:
                        if prodotto = i then stato := trovato
                        else if prodotto > i then stato := assente
                        else get(f) {passa il record}
                end {case}
        until stato ≠ inricerca
    end {RicercaIlProdotto};
begin {VenditeMassime}
    RicercaAnno; riccor := 0;
    qtacor := 0; RicercaProssimoMese {inizializza mesecor};
    repeat {processa il prodotto d'interesse in ciascun mese dell'anno}
        RicercaIlProdotto;
        case stato of
            trovato:
                with f↑ do
                begin if qta > qtacor then
                    begin qtacor := qta; riccor := ric;
                        m := mesecor
                    end;
                    RicercaProssimoMese
                end;
            assente: RicercaProssimoMese;
            fineanno: {non fa niente}
        end {case}
    until stato = fineanno;
    q := qtacor; r := riccor {passa il risultato}
end {VenditeMassime}
```

COMMENTI

1. Può essere interessante effettuare un paragone con l'Esempio 6.6 e considerare come un cambiamento nella struttura nel file di input si sia riflesso in un cambiamento di struttura nella procedura stessa. Il comportamento di questa procedura, specialmente in condizioni limite, è esattamente lo stesso di quello dell'Esempio 6.6. Tuttavia le tre procedure ausiliarie, tutte confor-

mi allo stesso modello, procedono in modo completamente differente, a causa della struttura del file.

2. Si è cercato, in modo particolare, di non fare mai riferimento ad un campo inesistente. Per esempio, nella procedura *RicercaAnno*, sarebbe illegale scrivere

if (*f*↑.*categoria* = *recanno*) **and** (*f*↑.*anno* = *y*) **then** *stato* := *trovato*

poiché *f*↑.*anno* esiste soltanto se *f*↑.*categoria* = *recanno* e non si può presumere che la seconda parte dell'espressione booleana sia valutata soltanto se è vera la prima parte. In altre circostanze, i campi di varianti vengono referenziati soltanto nelle istruzioni **case**, dove la discriminazione è fatta dal discriminante di variante del record.

ESEMPIO 11.7: AGGIORNAMENTO SEQUENZIALE

```
program AggiornamentoSequenziale (filevecchio, filenuovo, comandi,
        messaggi)
  {questo programma aggiorna un file di record che descrive modelli di automobili,
      identificati da una chiave numerica, secondo quanto specificato in un
      file di comandi di aggiornamento; produce un file aggiornato ed eventualmente
      dei messaggi di errore; i comandi di aggiornamento consistono
      nell'aggiunta, cancellazione e sostituzione di record; questo file non è di
      tipo text, così deve essere preparato da un altro programma; l'Esempio
      8.5 era simile ma faceva uso solamente di file di tipo text};
  const
    lunghezzaalfa = 30;
  type
    tipoalfa = packed array [1..lunghezzaalfa] of char;
    descrizione = record
      chiave: 1..99999;
      modello, colore: tipoalfa;
      anno: 0..99;
    end;
  var
    filevecchio, filenuovo: file of descrizione;
    messaggi: text;
    comandi: file of
      record tipocomandi: char; elemento: descrizione end;
  reccorrente: descrizione;
  trovato, copia: Boolean;
  procedure CopiaRecord
    {avanza di un record in filevecchio e filenuovo; se la variabile globale copia
      vale true, allora si deve copiare il record corrente in filenuovo};
  begin {CopiaRecord}
```

```
        if copia then write (filenuovo, reccorrente);
        copia := true; get(filevecchio);
        if not eof (filevecchio) then reccorrente := filevecchio↑
    end {CopiaRecord};
    procedure Errore (i: integer) {stampa un messaggio di errore};
    begin {Errore}
        with comandi↑.elemento do
            writeln (messaggi, 'Errore numero', i: 1, 'sul comando',
            chiave: 5, modello: lunghezzaalfa + 1,
            colore: lunghezzaalfa + 1, anno: 3)
    end {Errore};
begin {programma AggiornamentoSequenziale}
    reset(filevecchio); reset(comandi);
    rewrite(filenuovo); rewrite(messaggi);
    reccorrente := filevecchio↑; copia := true;
    while not eof(comandi) do
        with comandi↑ do {elabora un comando}
        begin trovato := false;
            {copia filevecchio in filenuovo fino alla chiave desiderata}
        while not eof(filevecchio) and (elemento.chiave > reccorrente.chiave)
            do CopiaRecord;
            if eof(filevecchio) then trovato := false
            else trovato := elemento.chiave = reccorrente.chiave;
            {chiave desiderata trovata o assente}
            if tipocomandi = 'A' then {aggiunge un record}
                if trovato then Errore(1) {duplica un record}
                else begin filenuovo↑ := elemento; put(filenuovo) end
            else if tipocomandi = 'C' then {cancella un record}
                if trovato then begin copia := false; CopiaRecord end
                else Errore(2) {record mancante}
            else if tipocomandi = 'S' then {sostituisce un record}
                if found then
                    begin reccorrente := elemento; copia := true; CopiaRecord
                    end
                else Errore(3) {record mancante}
            else Errore(4) {comando illegale};
            get(comandi) {esamina il comando successivo}
        end {fine dell'elaborazione di un comando};
        while not eof(filevecchio) do Copia Record {copia i record rimanenti}
    end {AggiornamentoSequenziale}.
```

COMMENTI

1. Questo tipo di programma dovrebbe essere familiare ad un programmatore Cobol, che noterà con interesse alcune differenze di programmazione, specialmente la maggiore flessibilità fornita in Pascal dalla funzione *eof*.
2. La sola convalida fatta nel programma riguarda il carattere che denota il comando. La preparazione del file *comandi* dovrebbe garantire che siano scritti soltanto i record validi. Perciò questo programma è più semplice di quello dell'Esempio 8.5.
3. Questo esempio non intende dimostrare che con il Pascal si possa fare tutto quel che si fa con il Cobol. Esso dimostra semplicemente che un programma di tipo gestionale non è necessariamente lungo, verboso e non strutturato, e che si può considerare seriamente il Pascal per applicazioni semplici e dirette in questa area.

ESEMPIO 11.8: CONCORDANZA DI TESTI (MODIFICATO)

```
program ConcordanzaDiTesti (input, output)
  {questo programma è una revisione dell'Esempio 9.5; viene effettuato un conteggio delle occorrenze di ciascuna parola, che viene stampato al termine del programma};
  const lunghparola = 20 {massima lunghezza utile per le parole};
    lunghdizionario = 1000 {massima lunghezza del dizionario};
  type tipoparola = packed array[1..lunghparola] of char {una stringa};
    descrittoreparola = record d: tipoparola; contatore: 1..maxint end;
  var dizionario: array[1..lunghdizionario] of descrittoreparola;
    parola: tipoparola {la parola corrente};
    dimdizionario, indice: 0..lunghdizionario {due indici nel dizionario};
  function EUnaLettera (ch:char): Boolean;...{si veda l'Esempio 9.3};
  procedure LeggeUnaParola;...{si veda l'Esempio 9.5};
  procedure RicercaEInserisce
    {in aggiunta alla RicercaEInserisce dell'Esempio 9.5, qui viene calcolato anche il numero delle occorrenze di ciascuna parola};
    var sinistra, centro, destra: 0..lunghdizionario;
      vocecorrente: descrittoreparola;
  begin {RicercaEInserisce}
    if dimdizionario = 0 then {dizionario vuoto, si inserisce la parola}
    begin dizionario[1].d := parola; dimdizionario := 1;
      dizionario[1].contatore := 1 .
    end
    else begin {caso generale}
      sinistra := 1; destra := dimdizionario;
      while sinistra ≤ destra do {ricerca binaria}
      begin centro := (sinistra + destra) div 2;
```

```
            vocecorrente := dizionario[centro];
            if vocecorrente.d ≤ parola then sinistra := centro + 1
            if vocecorrente.d ≥ parola then destra := centro - 1
         end;
         if vocecorrente.d = parola then {parola già incontrata}
               dizionario[centro].contatore := vocecorrente.contatore + 1
            else {parola non trovata; se possibile la si inserisce}
            if dimdizionario < lunghdizionario then {ci sta}
            begin dimdizionario := dimdizionario + 1;
               destra := dimdizionario;
               if dizionario[centro].d < parola then
                  {ma dizionario[centro + 1].d >parola}
                  centro := centro + 1;
               for destra := dimdizionario downto centro + 1 do
                  dizionario[destra] := dizionario[destra - 1];
               with dizionario[centro] do
               begin d := parola; contatore := 1 end
            end
      end {caso generale}
   end {RicercaEInserisce};
begin {programma ConcordanzaDiTesti}
   ... {si veda l'Esempio 9.5} ...
   for indice := 1 to dimdizionario do
      with dizionario[indice] do
         writeln(output, d, contatore: 3)
end {ConcordanzaDiTesti}.
```

COMMENTI

Le sole differenze tra questo programma e l'Esempio 9.5 sono l'aggiunta del numero di ricorrenze per ogni parola, l'uso di istruzioni **for** e **with** tutte le volte che sono applicabili e l'aggiunta di una variabile ausiliaria *vocecorrente* per evitare, quando è possibile, consultazioni ripetute a *dizionario*[*centro*].

ESERCIZI

11.1 Gestione del magazzino

Un negozio ha un archivio dei prodotti in magazzino. Un record in questo archivio contiene le informazioni seguenti:

a) codice del prodotto rappresentato come un intero a sei cifre;
b) designazione del prodotto come una stringa di 30 caratteri alfanumerici;
c) quantità a disposizione come un intero;

d) I.V.A. come numero reale;
e) prezzo unitario come numero reale se il prodotto è venduto a pezzi, oppure numero di pezzi per lotto e suo prezzo se invece il prodotto è venduto a lotti; se infine il prodotto è venduto in confezioni di varie misure (massimo 5), il numero di prodotti in ogni confezione, il suo prezzo, i possibili sconti (già inclusi nel prezzo) e il tempo di consegna.

Si descriva il file corrispondente in Pascal. Si scriva una procedura che stampi tutti i codici dei prodotti venduti in confezioni di cento unità, con lo sconto possibile.

11.2 Elaborazione di tabelle

Come nell'Esercizio 9.2, una tabella *tb*, di lunghezza prefissata, con un numero di elementi pari a *lunghezzatb*, contiene dei dati costituiti da una chiave alfanumerica e da informazioni di tipo *T*. Per migliorare l'accesso a questa tabella, un'entrata con indice *i* è tale che $i = f(chiavei)$; *f* è una cosiddetta funzione hash, che dà un risultato compreso fra 1 e *maxhash* (con $maxhash < lunghezzatb$), data una possibile chiave nella tabella. La funzione *f* è tale che due chiavi differenti possono causare collisioni, cioè $f(chiaven) = f(chiavem) = i$. In questo caso, tutte le entrate che collidono nello stesso punto sono concatenate in una lista di overflow; questa forma un terzo campo in ogni entrata necessaria.
L'intestazione di *f* è **function** *F*(*chiave*: *tipochiave*): *accessoveloce*, con *tipochiave* = **packed array**[*1..lunghezzachiave*] **of** *char*, e *accessoveloce* = *1..maxhash*. Si descriva la struttura dati della tabella *tb*. In caso di collisione, la lista di overflow è scandita linearmente. Si scriva la funzione:

function *Ricerca*(*chiave*: *tipochiave*; **var** *info*: *T*): *Boolean*

che restituisce in *info* l'informazione corrispondente alla chiave, se esiste (e quindi *Ricerca* = *true*, altrimenti *Ricerca* = *false*).

*11.3 Solitario

Questo gioco usa un mazzo di 32 carte (dal 7 all'asso). Ventotto carte sono sistemate sul tavolo, coperte, in quattro file (una per ogni seme) di sette carte (corrispondenti ai valori: 7, 8, ...donna, re); quattro carte sono lasciate nel mazzo. Il gioco inizia quando il giocatore prende una carta dal mazzo:

a) se la carta in mano è un asso, si scarta, e si prende un'altra carta dal mazzo;
b) se la carta in mano non è un asso, si mette sul tavolo nella posizione che corrisponde al suo seme e valore, e si prende la carta coperta che è in quella posizione;
c) si ripetono le mosse a) e b) fino a che non ci sono più carte nel mazzo. Per terminare il solitario occorre girare tutte le carte sul tavolo.

Dati i tipi seguenti:

tipovalore = (*sette, otto, nove, dieci, fante, donna, re, asso*);
tiposeme = (*fiori, quadri, cuori, picche*);

si definiscano le rappresentazioni per una carta, il tavolo da gioco con le sue 28 carte e il mazzo. Si scriva un programma che simuli questo gioco, considerando che esistano già una procedura *LeggeTavolo* che inizializza il tavolo da gioco e una procedura *LeggeMazzo* che inizializza il mazzo. Si stampi lo stato del gioco dopo aver messo ogni nuova carta sul tavolo.
Suggerimenti: (1) Le due procedure di lettura assegnino alla variabile booleana globale *errore* il valore *true* in caso di errore. (2) In output, si codifichi ogni carta con due caratteri che ne rappresentano il seme e il valore (per esempio, '8p' = otto di picche, e 'Dc' = donna di cuori).

11.4 Ordinamento di un file Cobol

L'algoritmo di ordinamento di file, usato nel Capitolo 7, è stato programmato nell'Esempio 7.7 per il caso di file Pascal. Si vogliono definire le modifiche da fare a questo esempio per ordinare i file in Cobol, cioè textfile strutturati su linee (chiamati «record» in Cobol). Il programma sarà parametrizzato dalla costante *lineamassima*, che definisce la massima lunghezza possibile per un record del file in Cobol, e da due variabili (lette in input) che danno lo spiazzamento del campo, usato come chiave dal primo carattere del record per la lunghezza di questa chiave.
Si suggerisce di simulare il comportamento di un file in Pascal (procedure *reset, get, put* e la variabile buffer) su un textfile strutturato in linee associato al file.

12
I set

12.1 IL CONCETTO DI INSIEME E IL TIPO SET

Il concetto di insieme è uno dei principali e più potenti della matematica. È così essenziale che per esso non esiste alcuna definizione soddisfacente che non sia lasciata, fino ad un certo punto, all'intuizione del lettore. È così potente che spesso è considerato la base della «matematica moderna». Nella programmazione sarebbe utile poter definire e processare insiemi di oggetti qualsiasi, anche insiemi i cui elementi siano di tipi differenti. Una tale idea, tuttavia, è generalmente considerata troppo complicata per consentire un'implementazione efficiente, e il concetto di insieme (*set*) in Pascal è molto più ristretto e meno potente. Il lettore, infatti, può essere in qualche modo deluso scoprendo la differenza tra l'astrazione dell'insieme e la sua realizzazione in Pascal. Inoltre, alcune implementazioni aumentano questa delusione, imponendo restrizioni più vincolanti di quanto non sia necessario, per una maggiore efficienza e facilità di programmazione.

Un set in Pascal non è esattamente un oggetto strutturato, poiché non è un oggetto costituito da elementi singoli. Piuttosto, un set è una collezione di *rappresentanti* (*modelli*) di oggetti tutti dello stesso tipo: ad esempio, un insieme di caratteri non contiene i caratteri di per sé, ma soltanto i loro rappresentanti. Si discuterà il concetto come se gli oggetti stessi fossero contenuti nell'insieme, ma quest'idea fondamentale dei rappresentanti-modelli è la giustificazione alle restrizioni più importanti imposte ai set in Pascal.

Un tipo set è definito dalla sua struttura e dal tipo base del set, cioè il tipo dal quale sono presi gli elementi del set. La sua sintassi è molto semplice, come è mostrato in Figura 12.1. Il tipo base deve essere un tipo ordinale. Un oggetto di un dato tipo set può essere considerato come un insieme di oggetti distinti del tipo base; ogni oggetto del tipo base può comparire o non comparire nell'insieme (cioè, nell'insieme può esserci o non esserci un suo rappresentante); due oggetti con lo stesso valore hanno lo stesso rappresentante. Conseguente-

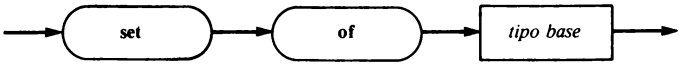


Figura 12.1 Diagramma sintattico di un tipo set

mente, nessun oggetto può comparire più di una volta; questa è una delle principali differenze fra gli insiemi matematici astratti e la loro implementazione in Pascal. L'altra è che soltanto i tipi ordinali possono essere usati come tipi base. Non si possono definire, per esempio, un set di array o un set di record, perché in tal caso sarebbe necessario definire i rappresentanti differenti per ogni valore possibile nel tipo array o nel tipo record.

Come conseguenza di queste restrizioni, non sarebbe utile definire prima una struttura astratta e poi descrivere la sua realizzazione in Pascal, come si è fatto per gli altri tipi strutturati. Descriveremo invece subito tutte le capacità dell'implementazione di questo nuovo concetto. La caratteristica più necessaria è il *costruttore di set*, cioè un modo per costruire i set che non è necessario per gli altri tipi strutturati, perché i file, gli array e i record *contengono* oggetti ed esistono quindi strumenti per aggiungere elementi al file o per aggiornare i singoli elementi di array e record.

Un costruttore di set ha la sintassi mostrata in Figura 12.2. Un problema importante è che il tipo del costruttore non è specificato in questa rappresentazione. Tutte le espressioni devono essere dello stesso tipo ordinale (si ricordi che un'espressione non è mai un tipo sottocampo, essendo qualsiasi variabile di tipo sottocampo estesa automaticamente al suo tipo ospite; si vedano i paragrafi 2.1 e 3.1) e il set così costruito è un tipo set *canonico*, il cui tipo base è questo tipo ordinale. Un costruttore di set contiene una lista di *designatori di membri*, separati da virgole e posti fra parentesi quadre. Un designatore costituito da un'unica espressione specifica il valore di questa espressione come un elemento del set. Due espressioni separate da due punti affiancati specificano che tutti i valori compresi nell'intervallo indicato sono elementi del set. Se l'intervallo è vuoto (essendo la prima espressione maggiore della seconda) non viene specificato alcun elemento. Un dato elemento appare una volta soltanto nel set, anche se esso è specificato parecchie volte. Il costruttore di set che non specifica alcun elemento (non contenendo alcun designatore di membri) ha il valore dell'insieme vuoto, di tipo indeterminato, compatibile con qualsiasi tipo di set.

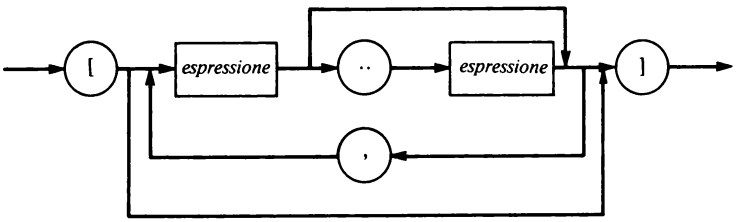


Figura 12.2 Diagramma sintattico di un costruttore di set

ESEMPIO 12.1: TIPI E COSTRUTTORI DI SET

```
type coloreprimario = (rosso, giallo, blu);
   piano = (seminterrato, pianoterra, rialzato, primopiano);
   errore = (spento, vuoto, parita, default, intasato);
   colore = set of coloreprimario;
   chiamataascensore = set of piano;
   statolettura = set of errore;
   {[rosso, giallo] è del tipo canonico coloreprimario, compatibile con il tipo colore;
      probabilmente rappresenta il colore arancio. [rosso] rappresenta il colore ros-
      so ed è esattamente dello stesso colore di [rosso, rosso, rosso]. Il tipo chiama-
      taascensore può rappresentare l'insieme dei piani dai quali si chiama un ascen-
      sore. [vuoto, parita, intasato] è un descrittore di stato per un lettore di sche-
      de, che evidentemente richiede l'intervento dell'operatore. [ ] rappresenta un
      insieme vuoto, di un tipo set non determinato, compatibile con i tre set defini-
      ti}
```

Non esistono delle vere costanti per un dato tipo di set, ma un costruttore di set per cui le espressioni che lo definiscono si riducono a costanti può essere valutato a tempo di compilazione da un compilatore ragionevolmente efficiente. Tuttavia, il costruttore di set non può comparire in una definizione di costante. Le operazioni sui set sono definite come segue:

1. *Assegnamento*: funziona come al solito. Un'espressione di tipo set è sempre del tipo set canonico di qualche tipo base. Può essere assegnata a una variabile (di tipo set) di un certo set di un dato tipo base, o di un sottocampo di questo tipo base, purché tutti gli elementi del set restino entro i limiti del tipo base della variabile di tipo set a cui è stata assegnata, in modo simile alle regole per l'assegnamento fra variabili di tipo sottocampo; ciò è una conseguenza del concetto di compatibilità in un'istruzione di assegnamento (si veda il paragrafo 1.6).
2. *Appartenenza* a un set: l'operatore diadico **in** ha per operando destro un tipo (canonico) di set e per operando sinistro un valore del corrispondente tipo base. Esso restituisce un valore di tipo *Boolean* e ha la stessa priorità degli operatori relazionali.
3. *Confronto*: gli operatori sono definiti per uguaglianza o disuguaglianza di due operandi di tipi di set compatibili. Due set sono uguali se contengono gli stessi elementi.
4. *Inclusione:* la verifica di inclusione è possibile per due operandi di tipi set compatibili; sono definiti soltanto due operatori per verificare se un insieme è contenuto nell'altro e cioè $\leq$ e $\geq$. L'espressione *set1* $<$ *set2* deve essere scritta (*set1* $\leq$ *set2*) **and** (*set1* $\neq$ *set2*), perché un operatore come $<$ sarebbe costoso da implementare e abbastanza inutile.
5. *Unione, intersezione e differenza* di set: sono indicati rispettivamente dagli

operatori +, *, −. Questi operatori mantengono la stessa priorità che hanno quando indicano le operazioni aritmetiche (si veda il paragrafo 3.1).

Queste sono le sole operazioni definite sui set nel Pascal standard e, naturalmente, sono state scelte perché sono utili, ma anche perché possono essere implementate efficientemente sui calcolatori odierni. Infatti, una delle più naturali implementazioni del concetto di set in Pascal, se si considerano accettabili alcune restrizioni, è di rappresentarlo come la sua funzione caratteristica, nella forma di una stringa di bit che sia di una dimensione naturale per il calcolatore, cioè un piccolo numero di celle di memoria contigue. Alcune implementazioni Pascal considerano il set solo come un'astrazione del concetto di parola di memoria e, conseguentemente, impongono limitazioni molto severe sul tipo base di qualsiasi set: se la parola di memoria ha *n* bit, il tipo base deve essere un tipo ordinale, il cui valore minimo abbia numero ordinale zero e il cui valore massimo abbia un numero ordinale inferiore a *n*; poiché i valori più comuni per *n* variano da 16 a 64, una sfortunata conseguenza è che molte applicazioni utili del concetto di set scompaiono (per esempio, l'idea di set di caratteri).
Lo Standard ISO non menziona la possibilità di restringere il dominio dell'insieme di base, e alcune altre implementazioni accettano un set di qualsiasi tipo ordinale, anche numeri interi molto grandi. Il problema di natura implementativa riguarda il tipo base usato dal costruttore di set: se si usano valori interi, il costruttore di set deve poter prevedere qualsiasi valore di un elemento intero, ricorrendo probabilmente ad una lista dinamica di elementi, sebbene una stringa di bit sarebbe molto più efficiente e forse anche sufficiente. Questo problema può essere evitato dando un significato speciale al compattamento del tipo e distinguendo due tipi di set canonici per lo stesso tipo base, uno compattato, l'altro no (questa possibilità è esplicitamente menzionata nello Standard ISO). Il tipo compattato ricorre ad una stringa di bit, ottimizzando il tempo di accesso e l'occupazione di memoria, ma il suo uso è ragionevole solo per elementi molto piccoli, mentre il tipo canonico non compattato usa, per esempio, una rappresentazione a lista. Si noti che questo è un caso in cui la richiesta di compattamento riduce, in circostanze ordinarie, sia le dimensioni che il tempo, contrariamente a quanto accade con array o record.
Se sono possibili sia i set compattati che quelli non compattati, la natura esatta del tipo di set canonico prodotto da un costruttore di set è stabilita implicitamente dal contesto in cui appare: per esempio, nell'assegnamento o nelle trasmissioni di parametri. In certe situazioni questa natura esatta è irrilevante, come ad esempio se il costruttore di set è usato come operando destro di un operatore di appartenenza ad un set.

ESEMPIO 12.2: IL CRIVELLO DI ERATOSTENE CON SET NON LIMITATI

```
program NumeriPrimi (output)
  {questo programma calcola i numeri primi inferiori ad un dato intero n, usando
    un vecchio algoritmo noto come «il crivello di Eratostene»; il crivello (setac-
    cio) è rappresentato come un insieme di interi da 1 a n};
```

```
    const n = ...{un intero molto grande (ad esempio 10 000)};
    var crivello, primi: set of 1..n {il crivello e il set di numeri primi};
        numero, successivo: 1..n;
begin {NumeriPrimi}
    crivello := [2..n-1] {inizialmente il crivello è pieno};
    primi := [1] {e il set di numeri primi è vuoto, eccetto che per 1, che è un caso
          speciale};
    successivo := 2 {numero minimo nel crivello};
    repeat {fino a che il crivello è vuoto}
        {in primi sono ora contenuti tutti i numeri primi inferiori a successivo}
        while not (successivo in crivello) do {trova il numero minimo}
            successivo := succ(successivo);
        primi := primi + [successivo] {questo numero è primo};
        writeln(output, successivo) {lo stampa};
        numero := successivo;
        while numero < n do
        begin {toglie dal crivello tutti i multipli di successivo}
            crivello := crivello - [numero];
            numero := numero + successivo
        end
    until crivello = [ ]
    {in primi sono ora contenuti tutti i numeri primi minori di n; possono essere
        quindi usati in una parte successiva del programma}
end {NumeriPrimi}.
```

COMMENTI

1. Questo programma, da considerare in ogni caso un esercizio scolastico, ha senso solo per piccoli valori di *n*: l'implementazione Pascal usata non consentirà set di più di 32, 64, o anche 256 elementi, oppure utilizzerà una rappresentazione inefficiente per set di dimensioni maggiori, perdendo così tutti i benefici di questo algoritmo, che evita le moltiplicazioni ripetute.
2. Sebbene il «setaccio» possa «contenere» *n*, non si userà questo valore, poiché non lo si può togliere. Se lo togliessimo, *numero* assumerebbe il valore $n + 1$, che non è contenuto nel tipo del suo sottocampo. Questo è un altro esempio frustrante dei problemi del Pascal Standard, già menzionati nel paragrafo 1.4, dovuto alla mancanza di «espressioni costanti»: bisogna che il tipo base di entrambi i set sia compreso nell'intervallo $1..n-1$, oppure che *successivo* e *numero* siano dichiarati nell'intervallo $1..n+1$.

ESEMPIO 12.3: IL CRIVELLO DI ERATOSTENE CON SET LIMITATI

```
program NumeriPrimiEfficiente(output)
    {questo programma risolve lo stesso problema dell'Esempio 12.2, ma ricorre ad
```

```
    una conoscenza a priori del supporto implementativo; si fa l'ipotesi che i set
    compattati, sui quali le operazioni sono molto efficienti, siano consentiti solo
    su tipi base i cui numeri ordinali stiano nel sottoinsieme 0..maxset, dove max-
    set è una costante predefinita che dipende dall'implementazione. Rappresen-
    tiamo un set di grandi dimensioni come array di set di dimensioni inferiori.
    Ciascun set «piccolo» è compattato ed ha la massima dimensione accettabile;
    il numero di elementi dell'array è m = (n + 1) div maxset + 1. Gli indici nu-
    mero e successivo diventano compositi, con una parte che seleziona il set ri-
    chiesto e un'altra che sceglie il numero richiesto al suo interno};
  const n = ...; maxset = ...; maxsetpiu1 = ... {maxset + 1};
    m = ...{(n + 1) div maxset + 1};
  var
    crivello, primi: array[0..m] of packed set of 0..maxset;
    numeroset, successivoset: 0..m {indici nell'array};
    numeronum, successivonum: 0..maxsetpiu1 {numeri nei set};
    finito: Boolean {vale true se il crivello è vuoto};
begin {NumeriPrimiEfficiente}
  for numeroset := 0 to m do
  begin {inizializza ambedue gli array}
    crivello[numeroset] := [0..maxset];
    primi[numeroset] := [ ]
  end;
  successivoset := 0; crivello[0] := crivello[0] - [0, 1];
  finito := false;
  repeat {fino a che il crivello è vuoto}
    successivonum := 0;
    while not (successivonum in crivello[successivoset] do
      {trova il primo numero presente}
      successivonum := succ(successivonum);
    {(successivoset, successivonum) rappresenta il numero primo successivo}
    primi[successivoset] := primi[successivoset] + [succesivonum]
    {il valore del numero primo appena trovato è:
      primocorrente = successivoset * maxsetpiu1 + successivonum; non serve
      più nel seguito}
    numeroset := successivoset; numeronum := successivonum;
    while numeroset < m do
    begin {elimina tutti i multipli}
      crivello[numeroset] := crivello[numeroset] - [numeronum];
      {il prossimo valore di numeroset è:
        numeroset + (primocorrente + numeronum) mod maxsetpiu1, che è
        calcolato in modo più efficiente dalle due istruzioni seguenti}
      numeroset := numeroset + successivoset;
      if numeronum + successivonum > maxset then
        numeroset := succ(numeroset);
      {il valore successivo di numeronum è:
        (numeronum + primocorrente) mod maxsetpiu1, che equivale all'istru-
```

```
            zione seguente}
          numeronum := (numeronum + successivonum) mod maxsetpiu1
        end {while numeroset < m};
        while crivello[successivoset] = [ ] do {trova il primo set non vuoto, oppure si ferma}
          if successivoset < m then successivoset := successivoset + 1
          else finito := true
      until finito
      {in primi sono ora contenuti tutti i numeri primi minori di n}
    end {NumeriPrimiEfficiente}.
```

COMMENTI

1. Questo programma è un buon esempio di una regola generale: l'implementazione efficiente di un algoritmo semplice può essere molto più complicata di quanto ci si aspetti.
2. Il programma illustra anche un altro problema: è difficile predire il sottocampo esatto che sarà necessario per alcune variabili. Il programma infatti non è corretto, perché *numeroset* assume valori al di fuori del sottocampo in cui è definito (si noti che è incrementato in passi successivi). Una soluzione semplice consisterebbe nel dichiararlo come un intero, ma non essere in grado di anticipare il comportamento del programma è un'ammissione di impotenza. Un'altra soluzione sarebbe quella di sostituire l'istruzione di assegnamento che incrementa *numeroset* con la seguente istruzione:

   ```
   if numeroset + successivoset > m then
     numeroset := m
   else numeroset := numeroset + successivoset
   ```

   Una terza soluzione, la più difficile, potrebbe essere quella di predire esattamente il valore massimo che può assumere questa variabile e di usarlo nella dichiarazione di *numeroset*. Il programma calcola probabilmente più numeri primi di quanti richiesti, poiché si ferma soltanto quando l'ultimo set (*crivello*[*m*]) è vuoto, e non quando *primocorrente* vale *n*.
3. Qui, ancora, la mancanza di espressioni costanti nel Pascal standard è un ostacolo. I commenti stabiliscono le relazioni che devono esistere fra le varie costanti definite, ma nulla può garantire la correttezza di queste asserzioni.
4. L'idea di set può anche essere implementata usando un array compattato di booleani; la presenza o l'assenza di un dato elemento è indicata dal valore booleano corrispondente a questo elemento, preso come indice nell'array. Usando quest'idea, il programma precedente potrebbe essere scritto completamente senza set e, infatti, sarebbe più breve. Tuttavia, sarebbe anche meno leggibile e molto meno efficiente, perché le operazioni di unione e differenza di set (specialmente efficienti quando uno degli operandi è un insieme

costituito da un solo elemento) sarebbero sostituite da accessi ripetuti a singoli elementi di array compattati.

5. Il programma precedente è stato scritto in modo tale da essere facilmente leggibile, supponendo che un compilatore ragionevolmente efficiente possa riconoscere l'uso di espressioni costanti all'interno dei cicli. Per esempio, il costruttore di set [*0..maxset*] è usato *m* volte nel ciclo **for** all'inizio del programma, e *crivello*[*successivoset*] è usato nel secondo ciclo **while**, che non modifica *successivoset*.
6. Sebbene sia stato spiegato, nel commento all'inizio del programma, che *numero* e *successivo* sono ora indici compositi, essi sono sostituiti da quattro nuove variabili, poste in relazione solo dai loro nomi. Una soluzione più chiara consisterebbe nel dichiarare:

   numero, successivo: **record** *parola*: *0..n*; *bit*: *0..maxsetpiu1* **end**

   Tuttavia, ciò renderebbe il programma più difficile da scrivere e da leggere, ottenendo per contro un vantaggio marginale.

I due esempi precedenti di uso di set possono essere considerati artificiosi. È difficile infatti dimostrare l'uso di set in esempi semplici e si vedranno casi più significativi negli esercizi. Tuttavia, l'uso dei set è molto frequente e ha la notevole proprietà di comparire anche in programmi che non dichiarano alcuna variabile di tipo set. L'operatore di appartenenza può essere usato come valido sostituto in molti confronti complessi. Per esempio:

1. Se i valori coinvolti sono piccoli valori ordinali, l'espressione booleana $(a \leq x)$ **and** $(x \leq b)$ può essere sostituita con x **in** $[a..b]$, che è più chiara e probabilmente più efficiente se a e b sono costanti. Questo è esattamente il caso in cui si vuol sapere se un certo carattere è una cifra: si scrive quindi

   if *c* **in** ['0'..'9']

   invece di

   if ('0' ≤ *c*) **and** (*c* ≤ '9')

   oppure, ancora meglio, si dichiara una variabile

   cifre: **set of** '0'..'9'

   la si inizializza con

   cifre := ['0'..'9']

   e si scrive

   if *c* **in** *cifre* **then**...

Naturalmente, *cifre* deve essere considerata nel programma come una costante.

2. Il caso è ancora più evidente con un confronto più complesso:

   ```
   ('a' ≤ c) and (c ≤ 'i') or ('j' ≤ c) and
   (c ≤ 'r') or ('s' ≤ c) and (c ≤ 'z')
   ```

   è vantaggiosamente sostituito da

   ```
   c in ['a'..'i', 'j'..'r', 's'..'z']
   ```

   (si veda l'Esempio 9.3), o meglio ancora da

   ```
   c in lettere
   ```

   con *lettere* definita, inizializzata e usata come *cifre* nel caso precedente. Se si vuole scrivere un programma che funzioni sicuramente con qualsiasi insieme di caratteri, si può perfino scrivere:

   ```
   lettere := ['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'] {!}
   ```

Questo uso semplice dei set, cui non si è fatto ricorso nei capitoli precedenti perché il concetto non era ancora disponibile, sarà usato liberamente nei rimanenti capitoli.

ESERCIZI

*12.1 Solitario (continuazione)

Nell'Esercizio 11.3 si supponeva fossero già scritte le procedure *LeggeTavolo* e *LeggeMazzo*. È tempo ora di scriverle, usando la seguente specifica:

```
procedure LeggeTavolo
    {inizializza il tavolo da gioco leggendo 28 triple di caratteri: un seme, codificato come 'f' 'q'
     'c' o 'p', rispettivamente per fiori, quadri, cuori, picche; un valore codificato come '7',
     '8', '9', 'X', 'J', 'D', 'K' o 'A'; uno spazio separa una carta dall'altra. La variabile boo-
     leana globale errore assume il valore true se si riscontra un errore durante la lettura}
procedure LeggeMazzo
    {inizializza il mazzo di carte, leggendo quattro triple di caratteri, codificati come sopra; può
     anche segnalare un errore};
```

12.2 **Master Mind** (migliorato)

Nell'Esercizio 10.3, avrete probabilmente usato un array booleano, che memorizzava i colori eliminati dalla soluzione. Si ripeta l'esercizio, usando un set al posto dell'array. Esiste un altro set che può essere introdotto nel programma per renderlo più semplice?

12.3 **Prenotazione ferroviaria**

Le Ferrovie dello Stato impiegano carrozze divise in 10 scompartimenti. Uno scompartimento di prima classe contiene due posti vicino al finestrino, due posti vicino al corridoio e due posti al centro. Negli scompartimenti di seconda classe i posti centrali sono quattro. Il sistema di prenotazione automatizzata permette al cliente di chiedere:

a) prima o seconda classe;
b) tipo di vettura (fumatori, non fumatori);
c) tipo di posto (finestrino, corridoio, centrale);
d) direzione del posto (senso di marcia o senso contrario).

Si descriva la rappresentazione di una carrozza nel file di prenotazione. Si scriva una funzione booleana che ricerchi in una data carrozza se c'è uno scompartimento libero che possa contenere un gruppo di *n* persone che desiderano viaggiare insieme e che esprimono richieste precise per i loro posti. I parametri di input sono:

a) la carrozza da esaminare;
b) la classe scelta;
c) il tipo di carrozza scelto;
d) il numero di persone nel gruppo;
e) il tipo e la direzione dei posti richiesti.

I parametri di output sono:

a) il numero dello scompartimento trovato, se la funzione restituisce il valore *true*;
b) l'insieme di errori riscontrati sui parametri di input, se la funzione è *false*.

Si suggerisce di considerare che, anche se la funzione è *false*, possano non esserci errori rilevati, cioè che non vi sia alcuno scompartimento libero.

13

Puntatori e variabili dinamiche

13.1 TIPI RECURSIVI

Fra le quattro strutture di dati fin qui considerate, solo una, la successione (o la sua implementazione, il file sequenziale) consente la definizione di oggetti strutturati con un numero variabile di elementi. Inoltre, per realizzare questa struttura di dati si è dovuto ricorrere a un dispositivo di memoria esterno. Nel caso di trasformazioni, insiemi o prodotti cartesiani, la struttura dell'oggetto è fissata nel momento in cui viene dichiarata e non può variare durante l'esecuzione del programma e neppure da un'esecuzione all'altra. Una variabile dichiarata di un dato tipo strutturato può far riferimento a vari oggetti, ma tutti questi oggetti devono avere la stessa struttura. Anche nel caso di tipi aggregati, che possono essere costruiti a partire da strutture differenti, queste devono essere scelte in un dato insieme, che non può cambiare durante l'esecuzione del programma. È a causa di questa proprietà fondamentale che un'area di memoria fissa può essere assegnata ad ogni variabile dichiarata, evitando così di dover ripetere la costruzione dei singoli oggetti e consentendone l'aggiornamento individuale. Questi oggetti hanno una *struttura statica* e sono associati alle cosiddette *variabili statiche*.

In molte circostanze, gli oggetti strutturati in modo statico non sono sufficienti e si deve ricorrere a *strutture dinamiche*, che cambiano durante l'esecuzione del programma. Non è più possibile assegnare un'area di memoria a tali oggetti dinamici, poiché la loro dimensione non è nota in fase di compilazione e varia durante l'esecuzione. Le variabili statiche non sono più utilizzabili e occorre fornire un sistema con cui poter fare riferimento ad oggetti dinamici.

Si noti, tuttavia, che una struttura di dati dinamica può essere scomposta in oggetti elementari a struttura statica.

Il modo più naturale e disciplinato di definire strutture di dati dinamiche ad alto livello è di permettere la definizione di *tipi recursivi*, tipi cioè la cui definizione contiene uno o più riferimenti a loro stessi, sia diretti che indiretti. Natu-

ralmente, poiché qualsiasi recursione deve terminare in qualche modo, la definizione stessa di tipo deve contenere una parte condizionale che controlla l'uso recursivo del tipo. Il modo più semplice di permettere la definizione di tipi recursivi è quello di generalizzare la nozione di aggregazione di tipo, o la sua realizzazione, il *record con varianti*. Per esempio, un tipo usato per descrivere la genealogia ascendente (il pedigree) di un dato individuo potrebbe essere definito nel modo seguente (che non è legale in Pascal):

```
type pedigree =
   record nomeproprio, cognome: stringa;
      padre, madre: record case conosciuto: Boolean of
         true: (p: pedigree);
         false: ( )
      end
   end
```

o in quest'altro modo, probabilmente più semplice:

```
type pedigree =
   record case conosciuto: Boolean of
      true: (nomeproprio, cognome: stringa;
            padre, madre: pedigree);
      false: ( )
   end
```

Conseguentemente, il permettere tipi recursivi in Pascal non necessiterebbe di alcuna modifica alla sintassi del linguaggio e risulterebbe una generalizzazione semplice e potente. Tuttavia avrebbe così tanti inconvenienti che può solo essere considerata un'astrazione, la cui implementazione renderebbe necessaria la presenza di determinate funzioni di basso livello. Bisogna considerare alcune delle difficoltà implicite nell'idea di tipo recursivo in un linguaggio come il Pascal, dove è considerato assolutamente fondamentale conoscere il tipo di ogni oggetto a tempo di compilazione e dove il concetto di tipo implica la sua struttura e la sua dimensione, così come molte altre proprietà.

1. Quando si dichiara una variabile di tipo recursivo, sarebbe impossibile conoscerne la struttura, poiché essa dipenderebbe dalla profondità di recursione. Di conseguenza sarebbero necessari dei costruttori di oggetti da poter usare in modo recursivo.
2. Se tali costruttori fossero permessi, l'aggiornamento selettivo degli elementi di un oggetto non dovrebbe essere generalmente permesso, poiché potrebbe cambiare la struttura dell'oggetto aggiornato.
3. I tipi recursivi costituiscono solo un modo limitato per costruire strutture dinamiche: essi possono descrivere soltanto strutture ad albero, senza rami co-

muni. Per esempio, sarebbe impossibile descrivere una situazione abbastanza frequente nei pedigree (specialmente per gli animali di razza selezionati), cioè avi comuni in rami differenti. Sarebbe ugualmente impossibile descrivere una struttura circolare — come una lista di elementi dove il successore dell'ultimo è il primo della lista — e, più in generale, strutture a grafo.

Per queste ed altre ragioni, i tipi recursivi esistono soltanto nei cosiddetti linguaggi senza tipo, dove ogni oggetto è allocato dinamicamente ogni volta che è necessario (come il LISP). Nei linguaggi che consentono la definizione di tipi, come il PL/1 o il Pascal, i tipi recursivi non sono consentiti, mentre esiste un'altra possibilità: il *puntatore*. In BASIC, Fortran o Cobol invece non c'è alcun modo per definire le strutture di dati dinamici, che devono essere simulate in modo inefficiente, o evitate del tutto.

13.2 PUNTATORI E ALLOCAZIONE DINAMICA

In Pascal, gli strumenti per costruire strutture dinamiche (delle quali le strutture recursive costituiscono un caso semplice e chiaro) sono un nuovo costruttore di tipo — il *puntatore* — e l'allocazione dinamica per gli oggetti puntati. Mentre gli oggetti statici possono essere allocati implicitamente ed in modo automatico al momento della dichiarazione delle variabili corrispondenti e deallocati quando queste variabili non esistono più (cioè quando il sottoprogramma in cui sono dichiarate termina la sua esecuzione), gli oggetti dinamici devono essere esplicitamente allocati e deallocati, usando opportune procedure predefinite. Non gli si può dare un nome per mezzo delle variabili ordinarie, poiché il loro numero non è noto a tempo di compilazione e inoltre la loro esistenza non è legata all'annidamento di dichiarazioni e attivazioni di sottoprogrammi. Quando si alloca un oggetto dinamico, si ha a disposizione un valore di un puntatore di tipo adatto, che può essere usato più tardi per far riferimento in modo diretto a questo oggetto. Per implementare una struttura di dati recursiva, si usa quindi un puntatore nella definizione di tipo per sostituire gli autoriferimenti con il riferimento ad un'altra copia dell'oggetto puntato.
Un tipo puntatore ha la forma sintattica mostrata in Figura 13.1. Il tipo del dominio può essere qualsiasi tipo semplice o strutturato, sebbene sia molto spesso un tipo record. Il tipo puntatore è così legato al tipo del suo dominio e un puntatore può far riferimento soltanto agli oggetti di questo tipo. Come conseguenza, viene mantenuta la proprietà più fondamentale dei tipi in Pascal, cioè che il tipo dell'oggetto puntato può essere noto a tempo di compilazione.

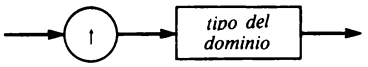


Figura 13.1 Diagramma sintattico di un tipo puntatore

Il tipo del dominio deve essere un identificatore di tipo; conseguentemente, il tipo degli oggetti puntati deve essere definito altrove. Sorge però un problema nel caso dei tipi recursivi, quando il tipo del dominio deve contenere un elemento del tipo del puntatore. La regola generale, presentata nei Capitoli 1 e 4, dice che ogni identificatore deve essere definito testualmente, o dichiarato prima della sua prima utilizzazione. Questo è evidentemente impossibile, se il tipo del dominio di un tipo puntatore deve contenere un riferimento al tipo del puntatore. Di conseguenza è necessario fare un'eccezione alla regola, solo per questo caso: l'identificatore del tipo che compare nella definizione di un tipo puntatore deve essere stato definito quando si raggiunge la fine della sezione di definizione del tipo.
Nel caso di un riferimento recursivo, il tipo puntatore deve comparire prima, per essere poi usato nella definizione del tipo del dominio (si veda l'Esempio 13.1).
Esistono solo due modi per generare un valore di tipo puntatore:

a) la costante universale **nil** è un valore di qualsiasi tipo puntatore e non punta nulla: ogni puntatore può assumere il valore **nil**;
b) la procedura predefinita *new*(*p*), dato un parametro *p* di tipo puntatore, alloca dinamicamente un oggetto del tipo puntato e assegna a *p* un puntatore all'oggetto creato: conseguentemente, non c'è modo di avere un puntatore ad un oggetto statico; questa proprietà fondamentale dei puntatori in Pascal evita i gravi pericoli che sorgerebbero se fosse possibile accedere ad un oggetto statico usando nomi differenti.

Dovrebbe essere chiaro che il concetto di puntatore in Pascal non è equivalente al concetto di indirizzo di memoria, anche se i puntatori sono normalmente implementati usando degli indirizzi. Mentre i programmi in linguaggio macchina permettono di ottenere l'indirizzo di qualsiasi oggetto, compreso l'indirizzo di una parte di un oggetto, o perfino di un'istruzione del programma, un puntatore Pascal può fare riferimento solo ad oggetti dinamici di tipi adatti. I puntatori in PL/1 sono più vicini al linguaggio macchina, con tutte le insicurezze inerenti, poiché il compilatore non può controllare che un oggetto sia veramente del tipo corretto.
Ci sono pochissimi operatori che trattano i puntatori in Pascal. L'assegnamento e il confronto di uguaglianza o ineguaglianza sono permessi soltanto fra due puntatori dello stesso tipo. Non esistono operazioni aritmetiche sui puntatori, né relazioni di ordinamento. I soli altri strumenti che hanno a che fare con i puntatori sono l'accesso all'oggetto puntato e le due procedure predefinite *new* e *dispose*. Tuttavia, una funzione può essere di tipo puntatore, cioè può restituire un valore di tipo puntatore.
Una variabile puntata è un altro caso di variabile la cui descrizione può essere finalmente completata (si veda la Figura 13.2). Una variabile puntatore è una variabile di qualche tipo puntatore. Una variabile puntata è una variabile il cui tipo è del tipo del puntatore. È un errore se la variabile puntatore non ha un

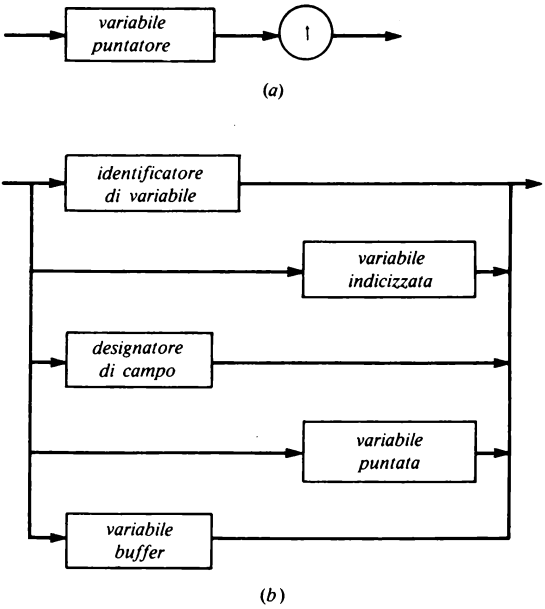


Figura 13.2 Diagramma sintattico completo per le variabili: (*a*) variabili puntate e (*b*) variabili

valore definito o è uguale a **nil**. Sebbene quest'ultimo errore sia generalmente scoperto durante l'esecuzione nella maggior parte delle implementazioni, il primo è più complicato e costoso da individuare ed è di conseguenza il pericolo più grave associato al concetto di puntatore. Un valore illegale di un puntatore può mettere in pericolo un programma completo.
La procedura predefinita *new* può essere chiamata in due forme:

1. Qualunque sia il tipo del dominio al quale *p* è legato, la forma *new*(*p*) crea un nuovo oggetto di questo tipo e assegna a *p* un puntatore che fa riferimento a questo oggetto. Questo nuovo oggetto è indefinito; in particolare, se il tipo del dominio è un record con varianti, nessuna particolare variante viene scelta, e tutte sono possibili (cioè *new* alloca un'area di memoria della dimensione necessaria per la variante più grande).

2. Se il tipo del dominio è un record con varianti (eventualmente annidate) la forma *new*(*p*, *c1*, ..., *cn*) crea un oggetto di questo tipo, con varianti corrispondenti alle costanti di selezione *c1*, ..., *cn*. Queste costanti sono enumerate nell'ordine di annidamento crescente delle varianti, ma possono mancare delle costanti alla fine della lista. Per tutte quelle varianti per cui è prevista una costante di selezione, si sceglie la particolare variante specificata nel-

la definizione del tipo del record. Tuttavia, la chiamata alla procedura *new* non assegna alcun valore all'indicatore di selezione, se esiste, e quest'operazione deve pertanto essere fatta esplicitamente nel programma. Le varianti per le quali non esistono costanti sono assimilabili al primo caso visto per la chiamata alla procedura *new*.

Lo Standard ISO non stabilisce che cosa succede se non vi è più memoria disponibile, quando viene chiamata la procedura *new*, così come non specifica che cosa succede se il programma è troppo lungo, o troppo complicato e così via. In alcune implementazioni, in tale situazione *new* restituisce un valore **nil**, mentre in altre implementazioni provoca semplicemente l'interruzione del programma con segnalazione di errore.
La procedura predefinita *dispose* può anch'essa essere chiamata in due forme:

1. *dispose*(*p*) distrugge (dealloca) l'oggetto dinamico referenziato da *p*. Naturalmente, produce un errore se *p* non è definito o è uguale a **nil**. Dopo questa chiamata, *p* deve essere considerato indefinito, poiché punta qualcosa che non esiste più.

2. *dispose*(*p*,*c1*, ..., *cn*) è l'opposto della corrispondente chiamata a *new*. Questa forma deve essere usata se l'oggetto puntato da *p* è stato creato usando la seconda forma di *new* e il numero di costanti di selezione deve essere uguale al numero usato nella chiamata a *new*.
   Questa necessità si spiega facilmente: la seconda forma di *new* è usata per allocare un oggetto esattamente della dimensione richiesta, generalmente più piccolo della dimensione della massima variante possibile del record; conseguentemente, sarebbe un disastro specificare nella *dispose* una dimensione maggiore di quella allocata realmente. Inoltre, poiché la dimensione esatta di un oggetto, creato usando la seconda forma di *new*, non può essere nota al momento della compilazione, non è legale usare questo oggetto in qualsiasi contesto che richieda la conoscenza della sua dimensione: ciò include il suo uso come variabile puntata su ambedue le parti di un'istruzione di assegnamento e come parametro di procedura, in qualunque modo venga passato. Infatti, gli oggetti creati usando la seconda forma di *new* non possono essere trattati come entità unica, ma soltanto elemento per elemento.

Poiché *dispose* fa sparire l'oggetto dinamico puntato, tutti i riferimenti a questo oggetto che ancora esistono (per mezzo di altre variabili di tipo puntatore che hanno lo stesso valore del parametro di *dispose*) spariscono e conseguentemente non sono mai definite. Sarebbe un grave errore se questa situazione si verificasse mentre è in uso l'oggetto dinamico, perciò non è legale effettuare una chiamata ad un oggetto dinamico se esso è un parametro attuale di tipo variabile, o se è all'interno di un'istruzione **with** in cui compare come variabile. Così, il seguente frammento di programma non è corretto:

```
with pvar↑ do
begin
   ... {istruzioni che usano campi dell'oggetto puntato da pvar}...
   dispose(pvar);
   ...
end
```

I puntatori sono uno strumento molto potente, poiché danno al programmatore la possibilità di costruire strutture dinamiche di qualsiasi forma e complessità. Come si è già notato, ciò è molto più potente dell'idea di tipo recursivo e quindi molto più utile. Tuttavia, i puntatori sono anche uno strumento poco disciplinato, di un livello molto più basso di quello di tutte le altre strutture dati in Pascal, anche se sono un concetto di più alto livello di quello degli indirizzi. Il concetto di puntatore ha una certa somiglianza con l'istruzione **goto**, nel senso che permette di costruire strutture di complessità non controllata, sulle quali non si può fare alcuna asserzione utile. Non a caso le istruzioni **goto** e i puntatori sono le due caratteristiche del Pascal per le quali non abbiamo regole di verifica o assiomi. Tuttavia si possono dare alcune regole semantiche formali (Alagič e Arbib, 1978) ma queste sono così complicate che non possono essere usate facilmente, e sono più di ostacolo che di aiuto. Inoltre, il fatto che un puntatore possa essere valutato da una funzione è un'ulteriore evidenza delle proprietà molto particolari dei puntatori: possono venir usati per costruire oggetti strutturati anche senza essere essi stessi oggetti strutturati.
Abbiamo affermato nelle pagine precedenti che alcuni usi dei puntatori possono causare errori o sono proibiti. Il grosso problema è che queste situazioni non possono essere individuate a tempo di compilazione (tranne in casi assai rari, usando un compilatore in grado di fare un'analisi completa del flusso del programma) e la loro individuazione è molto difficile e costosa, anche a tempo di esecuzione. Come conseguenza, il numero di errori che è possibile commettere e che non viene individuato nella maggior parte delle implementazioni è molto ampio e i puntatori costituiscono un fattore cruciale per l'affidabilità dei programmi. Ciò significa che dovrebbero essere usati soltanto se sono veramente necessari, e con la massima attenzione.

13.3 ESEMPI

ESEMPIO 13.1: STAMPA DELL'ALBERO GENEALOGICO

```
type
   nome = packed array[1..lunghezzamax] of char;
   genitore = ↑genealogia {un tipo puntatore, legato al tipo genealogia};
   genealogia = {non è necessario ricorrere a varianti}
      record nomeproprio, cognome: nome;
         padre, madre: genitore
      end;
```

```
procedure StampaAlbero(persona: genitore; incolonnamento: integer)
   {data una persona, questa procedura ne stampa l'albero genealogico, elencando
      sotto il nome di ogni persona il nome e la genealogia dei suoi genitori, inco-
      lonnati su uno spazio più a destra; il ramo maschile, se esiste, viene stampato
      prima del femminile; mentre è in stampa quello maschile, si fa uso di uno
      stack};
const lunghezzastack = 20 {sufficiente per 20 generazioni, cioè almeno 400 anni};
var cimastack: 0..lunghezzastack;
   nonpiupadrinelramo: Boolean;
   stack: array[1..lunghezzastack] of
      record g: genitore; i: integer end;
begin {StampaAlbero}
   {inizializza dapprima lo stack con la data persona}
   cimastack := 1
   with stack[cimastack] do
   begin g := persona; i := incolonnamento end;
   repeat {fino a che lo stack è vuoto}
      with stack[cimastack] do {elimina la radice del ramo da stampare}
      begin persona := g; incolonnamento := i end;
      cimastack := cimastack - 1;
      nonpiupadrinelramo := false {poiché c'è almeno una persona};
      repeat {fino a che nonpiupadrinelramo}
         with persona↑ do
         begin
            writeln(output, ' ': incolonnamento, nomeproprio, ' ',
               cognome);
            if madre ≠ nil then
            begin {ricorda il ramo femminile}
               cimastack := cimastack + 1;
               with stack[cimastack] do
               begin g := madre; i := incolonnamento + 1 end
            end;
            if padre ≠ nil then
            begin {elabora il ramo maschile}
               persona := padre; incolonnamento := incolonnamento + 4
            end
            else nonpiupadrinelramo := true
         end {with persona}
      until nonpiupadrinelramo
   until cimastack = 0
end {StampaAlbero}
```

COMMENTI

1. Si noti l'ordine nel quale sono fatte le definizioni di tipi: *genitore* precede *genealogia*, sebbene ne faccia uso, perché questa è la sola eccezione ammes-

sa alla regola di definizione prima dell'uso, come è spiegato nel paragrafo 13.2. Il tipo *genitore* è necessario perché è usato nella lista dei parametri della procedura *StampaAlbero* ed anche perché le due variabili dichiarate in posti diversi con l'indicatore di tipo ↑*genealogia* non sarebbero dello stesso tipo e conseguentemente non sarebbero compatibili, neppure in un assegnamento.

2. Nel Capitolo 14 apparirà un'altra versione di questa procedura, molto più semplice, che usa la recursione. La presente versione rende esplicito il meccanismo dello stack usato nelle procedure recursive, ma evita anche una recursione inutile nel ramo maschile di ogni persona stampata.
3. Si dovrebbe fare un controllo quando *cimastack* viene incrementata, per prevenire un overflow di memoria. Tuttavia, questo errore verrebbe identificato dai normali meccanismi di controllo dei tipi e, normalmente, il programma si fermerà subito; quindi è probabilmente questa l'azione più sensata che si possa fare. Questo è un argomento in più a favore dell'importanza dei controlli a tempo di esecuzione.
4. La struttura dati usata in questo esempio è in grado di rappresentare genealogie con antenati comuni. La procedura li stamperà senza problema, ma con i rami ripetuti.

ESEMPIO 13.2: PACKAGE DI GESTIONE DI STRINGHE

```
{per le stringhe sono possibili molte implementazioni, in funzione delle operazioni
   che si considerano utili; la scelta presentata nell'esempio è semplice e flessibile,
   ma inefficiente sia per il tempo di esecuzione che per lo spazio occupato. Una
   stringa è una lista di caratteri}
type naturali = 0..maxint;
   vartipostringa = ↑elementodistringa {stringa di lunghezza variabile};
   elementodistringa = {la testa della lista non contiene informazioni utili}
      packed record ch: char; successivo: vartipostringa end;
{vengono ora definite solo alcune operazioni; nello stesso modo se ne possono pro-
   grammare molte altre}
function Dimensione(varstringa: vartipostringa): naturali
   {restituisce il numero di caratteri della stringa};
   var s: naturali;
begin s := 0;
   while varstringa↑.successivo ≠ nil do
   begin s := s + 1; varstringa := varstringa↑.successivo end;
   Dimensione := s
end {Dimensione};
function ConverteStringa
      (var stringa: packed array[li..ls: naturali] of char
         {la stringa da convertire}
      ): vartipostringa
   {converte una stringa Pascal in una stringa di lunghezza variabile};
```

```
    var i: naturali; varstringa: vartipostringa;
begin {ConverteStringa}
    new(varstringa); ConverteStringa := varstringa; i := li;
    repeat {copia un carattere}
        new(varstringa↑.successivo); varstringa := vastringa↑.successivo;
        vastringa↑.ch := stringa[i]; i := i+1
    until i > ls;
    varstringa↑.successivo := nil
end {ConverteStringa};
function Concatenazione(prima, seconda: vartipostringa): vartipostringa
    {valuta una nuova stringa di lunghezza variabile pari alla concatenazione dei suoi
        due parametri};
    var varstringa: vartipostringa;
begin {Concatenazione}
    new(varstringa); Concatenazione := varstringa;
    while prima↑.successivo ≠ nil do
    begin {copia un carattere dalla prima stringa}
        new(varstringa↑.successivo); varstringa := varstringa↑.successivo;
        prima := prima↑.successivo; varstringa↑.ch := prima↑.ch
    end;
    while seconda↑.successiva ≠ nil do
    begin {copia un carattere dalla seconda stringa}
        new(varstringa↑.successivo); varstringa := varstringa↑.successivo;
        seconda := seconda↑.successivo; varstringa↑.ch := seconda↑.ch
    end;
    varstringa↑.successivo := nil
end {Concatenazione};
function Sezione
        (varstringa: vartipostringa {la stringa in esame};
        i {inizio della sezione},
        j {fine della sezione}: naturali
        ): vartipostringa
    {valuta una sottostringa di varstringa, dal carattere i al carattere j, se esiste; altri-
        menti restituisce una stringa vuota, rappresentata da una lista che contiene so-
        lo la sua testa};
    var sez: vartipostringa; k: naturali;
        stato: (incopia, terminato, finestringa);
begin {Sezione}
    new(sez); Sezione := sez; k := 0;
    while (varstringa↑.successivo ≠ nil) and (k < i) do
    begin varstringa := varstringa↑.successivo; k := k+1 end;
    if k = i then {il carattere i esiste in varstringa}
    begin stato := incopia;
        repeat {fino a che stato ≠ incopia}
            if k > j then stato := terminato
```

```
            else {copia un carattere}
            begin new(sez↑.successivo); sez := sez↑.successivo;
               sez↑.ch := varstringa↑.ch; k := k+1;
               varstringa := varstringa↑.successivo;
               if varstringa = nil then stato := finestringa
            end
         until stato ≠ incopia
      end;
      sez↑.successivo := nil
      {Sezione non è vuota solo se 1 ≤ i ≤ j ≤ Dimensione(varstringa)}
   end {Sezione};
   function Indice
            (soggetto {la variabile di tipo stringa in cui effettuare la ricerca},
            oggetto {la stringa da ricercare in soggetto}: vartipostringa;
            i: naturali {ia ricerca inizia dall'i-esimo carattere di soggetto}
            ): naturali
         {se oggetto compare come sottostringa di soggetto, prima o dopo l'i-esimo
               carattere, Indice valuta la posizione in soggetto relativa al primo carat-
               tere di oggetto; altrimenti Indice vale zero};
      var sogg, ogg: vartipostringa; j: naturali;
         stato: (inricerca, raggiunto, abortito);
         risultato: (inconfronto, trovato, nontrovato, impossibile);
   begin {Indice}
      j := 0; {avanza in soggetto fino al carattere i}
      while (soggetto↑.successivo ≠ nil) and (j < i) do
      begin soggetto := soggetto↑.successivo; j := j+1 end;
      if i = j then {il carattere i esiste}
      begin stato := inricerca;
         repeat {fino a che stato ≠ inricerca}
            sogg := soggetto; ogg := oggetto↑.successivo;
            risultato := inconfronto;
            repeat {fino a che risultato ≠ inconfronto}
               if ogg = nil then risultato := trovato
               else if sogg = nil then risultato := impossibile
               else if sogg↑.ch = ogg↑.ch then
                  begin sogg := sogg↑.successivo; ogg := ogg↑.successivo
                  end
                  else risultato := nontrovato
               until risultato ≠ inconfronto;
               case risultato of
                  trovato: stato := raggiunto;
                  impossibile: stato := abortito {in soggetto rimane un numero
                        insufficiente di caratteri};
                  nontrovato: {avanza di un carattere in soggetto}
                     begin soggetto := soggetto↑.successivo; j := j+1 end
```

```
            end {case risultato}
        until stato ≠ inricerca;
        if stato = raggiunto then Indice := j
        else Indice := 0
    end {if i = j}
    else Indice := 0
end {Indice};
{le operazioni precedenti potrebbero venir usate nella seguente porzione di programma, che sostituisce in alcune stringhe la prima occorrenza di 'computer science' (se esiste) con 'informatica':
    var vecchionome, soggetto: vartipostringa;
        lunghezza, posizione: naturali;...
    vecchionome := ConverteStringa ('computer science');
    lunghezza := Dimensione (vecchionome);
    posizione := Indice (soggetto, vecchionome, 1);
    if posizione ≠ 0 then soggetto :=
        Concatenazione (Concatenazione (Sezione
            (soggetto, 1, posizione-1), ConverteStringa ('informatica')),
            Sezione(soggetto, posizione+lunghezza,
            Dimensione(soggetto)));}
```

COMMENTI

1. L'implementazione particolare del concetto di stringa, scelta nell'esempio precedente, ha delle buone qualità e dei difetti. Da un lato è semplice e facile da processare; con un'implementazione che usa una lista di stringhe di lunghezza fissa (si veda l'Esercizio 13.4), le funzioni *Concatenazione* e *Indice* sarebbero molto più complicate e difficili da capire. Tuttavia, il metodo qui esposto è inefficiente, sia per il tempo richiesto per l'esecuzione, che per lo spazio occupato. Usare un puntatore per ogni carattere di una stringa richiede circa tre volte lo spazio richiesto per una stringa di lunghezza fissa ed anche di più, se l'implementazione del Pascal ignora il compattamento. Scandire ogni elemento della lista per trovare l'ultimo porta via molto tempo e inoltre è molto costoso costruire una nuova stringa per ogni operazione. Se non si intende fare un uso intensivo di queste procedure, i difetti esposti sono compensati dalla semplicità e dalla naturalezza delle funzioni, come si può vedere nell'esempio.
2. Si è aggiunto un ulteriore elemento in testa alla lista per ogni stringa — sebbene esso non contenga informazioni utili — per evitare di dover ricorrere ad una rappresentazione speciale per la stringa vuota e per semplificare tutte le operazioni, presumendo che, benché *varstringa* sia in realtà un puntatore, non assumerà mai il valore **nil**. Il primo elemento della lista potrebbe essere usato per memorizzare informazioni sulla lista (per esempio la sua lunghezza e un puntatore al suo ultimo elemento). Ciò renderebbe più efficiente l'intero insieme di procedure, anche se più difficili da programmare, poiché quest'informazione aggiuntiva dovrebbe essere esaminata ed inserita ogniqualvolta fosse necessaria.

3. Gli elementi della lista sono allocati per mezzo della procedura predefinita *new*, ma non sono mai rilasciati. Un'operazione di deallocazione potrebbe venir definita come una procedura supplementare, ma non potrebbe essere usata nell'ultimo assegnamento di questo esempio, poiché le stringhe sono create ed usate immediatamente, mai assegnate a variabili. In ogni caso, una procedura di deallocazione non farebbe probabilmente uso della procedura standard *dispose*, poiché gli elementi della lista sono della stessa dimensione e, per avere una strategia di allocazione molto efficiente, basterebbe ricorrere ad una semplice lista degli elementi disponibili (si veda l'Esercizio 13.2).
4. L'uso di funzioni che valutano le stringhe permette operazioni molto naturali. Ciò è possibile perché i puntatori possono essere valutati da funzioni, mentre gli oggetti strutturati sono proibiti. Tuttavia, si noti che una funzione che valuta un puntatore non può essere usata in tutti i punti in cui si può usare un puntatore, perché un designatore di funzione è un'espressione e non una variabile. Nella parte destra di un assegnamento o come parametro attuale passato per valore ciò non comporta alcuna differenza, ma non si può utilizzare direttamente il contenuto del puntatore valutato da una funzione e si deve ricorrere ad una variabile ausiliaria. Questa restrizione è stata fatta, probabilmente, per ragioni sintattiche, poiché la natura esatta di un'istruzione di assegnamento, la cui parte sinistra sia l'utilizzo diretto di una chiamata di funzione, potrebbe essere determinata solo molto tardi nell'analisi dell'istruzione. Si consideri come esempio l'istruzione:

   Sezione(*soggetto, posizione* + *lunghezza,*
   Dimensione(*soggetto*))↑.*successivo* := ...

   dove si vorrebbe chiamare la funzione *Sezione* per valutare l'indirizzo di un record di tipo *elementodistringa*, di cui si deve accedere al campo *successivo*.
5. Si potrebbero definire altre operazioni utili per le stringhe, nello stesso modo usato nell'esempio precedente: procedure di lettura e di scrittura (da o in file di tipo *text*), predicati di confronto, operazioni di scansione più complicate o generali di *Indice* e così via.
6. Si noti l'uso dei due indicatori di stato, *stato* e *risultato*, in funzione di *Indice*. Essi rendono la costruzione dei cicli molto più semplice, più efficiente e più facile da capire che non se si fossero usati indicatori booleani.

ESEMPIO 13.3: CROSS-REFERENCE

```
program CrossReference (input, output)
  {questo programma è un miglioramento dell'Esempio 9.5 e stampa ogni parola
    che compare nel testo di input insieme alla lista dei numeri di linea in cui
    compare; il file di input viene stampato all'inizio con questi numeri di linea};
  const
    lunghparola = 20 {massima lunghezza utile per le parole};
```

```
    lunghdizionario = 997 {massima lunghezza del dizionario: deve essere un
      numero primo};
    maxnumlinea = 9999 {massimo numero di linea};
  type
    tipoparola = packed array[1..lunghparola] of char;
    indice = 0..lunghdizionario;
    tiponumlinea = 0..maxnumlinea;
    puntaelemento = ↑elemento;
    elemento {della lista di riferimento di una parola} =
      packed record
        numlinea: tiponumlinea; successivo: puntaelemento
      end;
  var
    dizionario: array[indice] of
      packed record
        nome: tipoparola {la parola};
        testalista, codalista: puntalemento {la sua lista di riferimento};
        vuota: Boolean {stato di occupazione dell'elemento}
      end;
    numelemento: indice;
  procedure CostruisceIlDizionario
    {legge il testo di input, lo stampa con i numeri di linea e costruisce il dizionario
      delle parole con i riferimenti};
    var lettere: set of char;
      parola: tipoparola {la parola appena letta};
      numerolinea: tiponumlinea {il numero di linea corrente};
      iniziolinea: Boolean {vale true se si è all'inizio di una nuova linea};
      charletto: char {il carattere appena letto};
      elemcorrente: indice {per la scansione del dizionario};
    procedure CarattereSuccessivo
      {legge il carattere successivo, lo stampa, ed elabora nuove linee};
    begin {CarattereSuccessivo}
      if iniziolinea then
      begin {siamo all'inizio di una nuova linea}
        writeln(output) {termina la linea precedente};
        numerolinea := numerolinea + 1; iniziolinea := false;
        write(output, numerolinea: 5, ' ')
      end;
      read(input, charletto);
      if not eof(input) then
      begin {altrimenti charletto ed eoln sono indefiniti}
        write(output, charletto);
        if eoln(input) then iniziolinea := true
      end
    end {CarattereSuccessivo};
```

```
procedure LeggeUnaParola
   {molto simile alla procedura omonima dell'Esempio 9.5 ad eccezione
      dell'uso di CarattereSuccessivo e del set lettere};
   var parolaletta: array[1..lunghparola] of char;
      k: 0..lunghparola;
begin {LeggeUnaParola}
   k := 0;
   repeat {fino a che charletto in lettere}
      if k < lunghparola then
      begin k := k+1; parolaletta[k] := charletto end;
      CarattereSuccessivo
   until not (charletto in lettere);
   for k := k+1 to lunghparola do parolaletta[k] := ' ';
   pack(parolaletta,1,parola)
end {LeggeUnaParola};
procedure RicercaEInserisce
   {ricerca la parola nel dizionario, la inserisce se non è presente, aggiun-
      ge il numero di linea corrente nella lista di riferimento; il dizionario è
      organizzato come una tabella hash con ricerca quadratica per la gestio-
      ne delle collisioni};
   var hash, passo: indice;
      ricorrenza: puntaelemento;
      trovato: Boolean;
      i: 1..lunghparola;
begin {RicercaEInserisce}
   {calcola l'indice hash}
   hash := 1;
   for i := 1 to lunghparola do
      if parola[i] ≠ ' ' then
         {evita che gli spazi influenzino il calcolo}
         hash := hash * ord(parola[i]) mod lunghdizionario;
   trovato := false; passo := 1;
   new(ricorrenza);
   with ricorrenza↑ do {inizializza il nuovo elemento della lista}
   begin numlinea := numerolinea; successivo := nil end;
   repeat {fino a che non si trova un elemento}
      with dizionario[hash] do
      if vuota then
      begin {la parola può essere inserita qui}
         trovato := true; nome := parola; vuota := false;
         testalista := ricorrenza; codalista := ricorrenza
      end else
      if nome = parola then
      begin {la parola cercata è stata trovata}
         trovato := true;
```

```
            codalista↑.successivo := ricorrenza;
            codalista := ricorrenza
        end {parola già presente}
        else begin {collisione; cerca altrove}
            hash := (hash+passo) mod lunghdizionario;
            if passo = lunghdizionario then Errore('Tabella piena')
                {la procedura Errore non viene definita qui}
            else passo := passo+2 {tentativi successivi sono fatti a distanze
                che aumentano in maniera quadratica}
        end {collisione}
    until trovato
  end {RicercaEInserisce};
begin {CostruisceIlDizionario}
  {inizializzazione}
  for elemcorrente := 0 to lunghdizionario do
    dizionario[elemcorrente].vuota := true;
  lettere := ['a'..'i', 'j'..'r', 's'..'z', 'A'..'I', 'J'..'R', 'S'..'Z'];
  numerolinea := 0; iniziolinea := true;
  CarattereSuccessivo;
  {elabora il testo (non vuoto)}
  repeat {fino a che eof(input)}
    if charletto in lettere then
      begin {inizio di una parola}
        LeggeUnaParola {¬(charletto in lettere)};
        RicercaEInserisce
      end else CarattereSuccessivo
  until eof(input)
end {CostruisceIlDizionario};
procedure OrdinaIlDizionario
  {ordina il dizionario secondo un ordine alfabetico; lascia tutte le locazioni oc-
    cupate all'inizio del dizionario e pone numelemento uguale al valore delle
    locazioni usate};
begin {il corpo di questa procedura non è definito qui: si veda l'Esempio 14.4
    per una procedura di ordinamento molto efficiente}
end {OrdinaIlDizionario};
procedure StampaIlDizionario
  {dal momento che il dizionario è già ordinato, la sua stampa è banale};
  const lunghlinea = 120 {lunghezza di una linea stampata};
    numerocifre = 6 {dimensione di un numero di riferimento, con gli spazi
      di allineamento, in relazione a maxnumlinea};
  var poslinea: 0..lunghlinea {posizione corrente nella linea di stampa};
  ricorrenza: puntalemento {per scandire la lista dei riferimenti};
  elemcorrente: indice {per scandire il dizionario};
begin {StampaIlDizionario}
  page(output);
  writeln(output, 'Cross-reference del testo precedente');
```

```
        writeln(output); writeln(output);
        for elemcorrente := 0 to numelemento do
          with dizionario[elemcorrente] do
          begin {stampa un elemento e la sua lista di riferimenti}
            write(output, nome) {nome è una stringa di caratteri};
            poslinea := lunghparola; ricorrenza := testalista;
            repeat {poslinea = posizione corrente nella linea; ricorrenza punta al
                    riferimento corrente}
              if poslinea + numerocifre > lunghlinea then
              begin {il prossimo numero di riferimento non va bene}
                writeln(output); write(output, ' ': lunghparola);
                poslinea := lunghparola
              end {il prossimo numero di riferimento ora va bene};
              write(output, ricorrenza↑.numlinea: numerocifre);
              poslinea := poslinea + numerocifre;
              ricorrenza := ricorrenza↑.successivo
            until ricorrenza = nil;
            writeln(output)
          end {fine dell'elaborazione di un elemento}
      end {StampaIlDizionario};
    begin {programma CrossReference}
      CostruisceIlDizionario;
      OrdinaIlDizionario;
      StampaIlDizionario
    end {CrossReference}.
```

COMMENTI

1. Il programma precedente è un esempio di come la struttura possa rappresentare la gerarchia dei suoi processi differenti e suggerire il modo in cui costruirlo. I nomi globali (costanti, tipi o variabili) sono nomi che vengono usati in più parti; se una variabile è usata in procedure differenti, ma non ha significato al di fuori di queste procedure, è dichiarata localmente due volte, invece di essere globale (si veda per esempio, *ricorrenza* o *elemcorrente*). Tale metodo è del tutto generale e consente una facile sostituzione di ogni procedura con una completamente differente che svolge la stessa funzione. Come caso limite, se *CostruisceIlDizionario* ordinasse già il dizionario, *OrdinaIlDizionario* sarebbe un blocco vuoto. Il costo della definizione e l'uso delle procedure chiamate una sola volta è del tutto trascurabile in un'implementazione abbastanza efficiente.
2. Poiché le collisioni dei nomi sono manipolate mettendo altrove nel dizionario i nomi in conflitto, la dimensione del dizionario stabilisce un limite definito al numero di parole differenti che possono essere processate e per questo motivo è necessaria la chiamata a *Errore* in *RicercaEInserisce*. *Errore* dovrebbe probabilmente porre fine al programma immediatamente, usando

per esempio un **goto** che salti ad una label, immediatamente prima della fine del programma. Un altro modo per trattare le collisioni sarebbe quello di inserire ogni nome nel dizionario in testa alla lista di tutti i nomi con lo stesso indice hash.

3. I due puntatori ad una lista di ricorrenza forniscono una via facile per costruire la lista e stamparla nello stesso ordine (come una coda). Con un solo puntatore la lista sarebbe uno stack e dovrebbe essere invertita prima di venir stampata.
4. L'uso di un insieme «fisso» di lettere è un modo semplice, efficiente e leggibile per evitare l'uso della funzione *EUnaLettera* dell'Esempio 9.5. Il costruttore di set è conforme all'insieme di caratteri EBCDIC, ma va bene anche per lo Standard ISO-ASCII.
5. La funzione di hashing usata in *RicercaEInserisce* è piuttosto inefficiente, ma è indipendente dalla particolare implementazione. In linguaggio macchina si può considerare una nuova parola da inserire nel dizionario come un gruppo di poche parole macchina, riducendo così il numero delle moltiplicazioni. Gli spazi sono tralasciati per evitare di dar loro troppa importanza nelle parole brevi.
6. *OrdinaIlDizionario* è necessaria, perché il principio su cui si basa una tabella hash è di inserire le chiavi in modo pseudo-casuale, non adatto per la stampa finale. Si sarebbero potuti scegliere numerosi algoritmi di ordinamento, ma si è preferito posticipare la scelta fino al Capitolo 14, dove si potrà usare il miglior algoritmo conosciuto, valido in condizioni normali e che richieda solo una quantità limitata di memoria supplementare.
7. Nella stampa finale, i numeri di riferimento sono stampati ad intervalli regolari, formando delle colonne in corrispondenza di una parola usata parecchie volte. Una scelta differente sarebbe stata quella di stampare i numeri separandoli semplicemente con delle virgole. L'estensione di campo automatica, fatta dalla procedura standard *write*, sarebbe andata bene, ma sarebbe stato difficile conservare una traccia della posizione corrente sulla linea.
8. Come negli esempi precedenti, non abbiamo usato la procedura predefinita *dispose*, benché questo sia un esempio di programma completo. Questa è una situazione frequente che spiega perché *dispose*, a volte, non venga implementata affatto: in molti programmi, la locazione dinamica è usata per permettere una ripartizione ottimale della memoria fra strutture differenti di dimensioni non prevedibili, ma in cui la memoria richiesta può solo aumentare durante l'esecuzione del programma.

ESERCIZI

*13.1 Elaborazione di messaggi

Una stazione meteorologica è collegata con numerose stazioni trasmittenti, sia fisse e permanenti (stazioni di rilevamento con personale di servizio), che fisse ma non perma-

nenti (rilevatori automatici), che mobili ed intermittenti (navi che transitano nella regione). Ad intervalli di tempo prefissati, ogni trasmittente invia alla stazione un messaggio contenente la lettura dei principali parametri, rilevanti ai fini delle previsioni meteorologiche. Un messaggio è diviso in pacchetti con un numero fisso di caratteri, preceduti da un numero che identifica la stazione trasmittente. Il carattere '.' chiude il messaggio, che può essere di qualsiasi lunghezza. Si presume che tutti i messaggi arrivino alla stazione sulla stessa linea di trasmissione (che può essere collegata ad un file sequenziale). L'ordine di arrivo dei diversi pacchetti da una data stazione trasmittente è lo stesso dell'ordine in cui sono trasmessi. Tuttavia, i pacchetti di stazioni differenti possono sovrapporsi. Si descriva una procedura per trattare il file dei messaggi, che ricostruisca i messaggi trasmessi. Poiché il numero delle stazioni trasmittenti è variabile, quelle attive sono organizzate in una lista, ordinata dal loro codice d'identificazione. Ogni elemento della lista contiene, fra le altre cose, il puntatore al primo e all'ultimo pacchetto della lista dei messaggi trasmessi. Non appena si arriva alla fine di un messaggio, questo viene stampato e lo spazio di memoria — necessario per la lista dei pacchetti e per il descrittore della stazione trasmittente — viene rilasciato.

13.2 Package per la manipolazione di stack e code

Un package per la manipolazione di stack e code di elementi di un dato tipo *T* deve poter svolgere le seguenti funzioni: creare e distruggere uno stack o una coda, inserire ed estrarre un elemento da uno stack, aggiungerlo o toglierlo da una coda. Per ridurre al minimo lo spazio di memoria necessario per queste strutture e il tempo necessario per elaborarle, le si organizzi a liste e le si costruisca usando il metodo seguente:

a) inizialmente il package non deve allocare alcuna struttura se non la testa di una lista (si veda (d));
b) quando si costruisce una struttura, il package prepara dinamicamente un opportuno descrittore;
c) quando si inserisce un elemento in uno stack o lo si aggiunge ad una coda, il package lo costruisce dinamicamente;
d) quando si estrae un elemento da uno stack o lo si rimuove da una coda, o si distrugge un'intera struttura, il package inserisce tutti questi elementi in una lista libera. Allocazioni successive, che richiederebbero la costruzione di un nuovo elemento, usano questa lista fino a che non è vuota, prima di riprendere il processo di allocazione dinamica.

Seguono le specifiche del package:

```
type puntatoredato = ↑dato
   dato = ... {descrittore per un elemento di stack o di coda};
procedure CreaStack(var s: puntatoredato) {crea uno stack chiamato s};
procedure CancellaStack (s: puntatoredato) {cancella lo stack s};
procedure Push (var s: puntatoredato; elem: T) {pone elem in cima a s};
procedure Pop (var s: puntatoredato; var elem: T; var vuota: Boolean)
   {toglie elem dalla cima di s, o pone vuota a true se s è vuota};
procedure CreaCoda(var c: puntatoredato) {crea una coda chiamata c};
procedure CancellaCoda(c: puntatoredato) {cancella la coda c};
procedure AggiungeA(var c: puntatoredato; elem: T) {aggiunge elem alla fine di c};
```

procedure *ToglieDa*(**var** *c*: *puntatoredato*; **var** *elem: T*; **var** *vuota*: *Boolean*)
{toglie elem dalla testa di c, o pone vuota a true se c è vuota};

13.3 Matrici sparse

Rappresentare una matrice sparsa di grosse dimensioni (cioè una matrice in cui la maggior parte degli elementi non contiene informazioni) come un array, porta ad un grande spreco di memoria. Sono necessari altri metodi che fanno uso di liste dei soli elementi significativi. Si progetti una rappresentazione per tali matrici che soddisfi le richieste seguenti:

a) le matrici sono quadrate ma di dimensioni variabili;
b) le operazioni sulle matrici sono sempre fatte su tutti gli elementi di una riga per indice di colonna crescente, a partire da un indice di colonna dato, oppure su tutti gli elementi di una colonna per indici di colonna crescenti, a partire da un dato indice di riga;
c) l'elemento di una matrice è di tipo *T*; un elemento di tale tipo può essere letto da un file di tipo text, usando la procedura *LeggeT* (**var** *x*: *T*).

Dato che si desidera accelerare l'accesso agli elementi della matrice — quando vengono verificate le ipotesi descritte sopra — si progettino le strutture dati necessarie e si programmino le procedure seguenti.

1. *ScandisceLaRiga* e *ScandisceLaColonna* che, data una matrice *m*, una riga *i* o una colonna *j*, un indice di partenza *pi* o *pj*, predispongano la scansione di una riga o di una colonna di *m* (tre parametri: *m*, *i* o *j*, *pi*, o *pj*).
2. *SuccessivoDiRiga* e *SuccessivoDiColonna* che, data una matrice *m*, forniscano il valore *v* dell'elemento successivo nella riga o colonna inizializzata, insieme ai suoi indici di colonna o di riga; le variabili booleane *fineriga* o *finecolonna* assumano valore *true* se non esistono tali elementi (quattro parametri: *m*, *v*, *i*, o *j*, *fineriga* o *finecolonna*).
3. *Inizmatrice* che inizializzi una matrice *m* dalle triple (*i*, *j*, *v*), lette in un textfile.

13.4 Package di gestione di stringhe (soluzione alternativa)

Nell'Esempio 13.2, le stringhe di lunghezza variabile sono state rappresentate come liste i cui elementi contenevano soltanto un carattere. Nell'Esercizio 13.1, i messaggi (che sono in realtà delle stringhe) sono trasmessi come lista di pacchetti, dove ogni pacchetto è una stringa di lunghezza fissa. Combinando queste due idee, è possibile rappresentare una stringa di lunghezza variabile come una lista di pacchetti, di cui l'ultimo può essere incompleto. Si progettino le strutture dati necessarie per questa rappresentazione, e si riscrivano le procedure e le funzioni dell'Esempio 13.2, cioè, *Dimensione*, *ConverteStringa*, *Concatenazione*, *Sezione* e *Indice* e inoltre la procedura *StampaStringa* (*vs*: *varstringa*) che stampa sul file *output* la stringa di lunghezza variabile *vs*.

14

Procedure recursive

Un concetto è recursivo se contiene un riferimento a se stesso, o se viene parzialmente definito dal suo stesso uso. La sintassi del Pascal è un esempio di funzioni recursive, molto comuni in matematica, come si può osservare, ad esempio, nella definizione di un'espressione (Capitolo 3), o di un'istruzione (Appendice B). Abbiamo già preso in considerazione l'uso del concetto recursivo per descrivere strutture di dati dinamici. In questo capitolo applicheremo il concetto della recursione ad azioni, usando procedure o funzioni recursive.

Una definizione recursiva consente di definire un numero infinito di oggetti con un numero finito di regole. Ad esempio, si può definire un insieme N di numeri naturali con le due regole: 1 è un numero naturale, e il successore di un numero naturale è un numero naturale. Allo stesso modo un programma recursivo (cioè un programma che fa uso del concetto di recursione) permette di definire un numero infinito di processi con un numero finito di costrutti linguistici, anche senza dover usare costrutti iterativi o ripetitivi. Di fatto, quest'ultimo tipo di istruzioni può sempre essere descritto in maniera recursiva. Per contro, un'istruzione recursiva non può essere sempre descritta in termini completamente generali per mezzo di costrutti iterativi o ripetitivi.

Ciò significa che la programmazione che si avvale del concetto di recursione aumenta enormemente la potenza dell'insieme di strumenti messi a disposizione dal linguaggio di programmazione.

Ciononostante, come vedremo, è possibile fare un uso improprio della recursione: essa può portare a programmi estremamente inefficienti quando una soluzione iterativa potrebbe essere molto più semplice. Come regola generale, possiamo assumere il fatto che una soluzione recursiva può essere interessante e utile solo quando il problema da risolvere, o la funzione da definire, o i dati da elaborare siano essi stessi definiti in un modo recursivo.

Introdurre la recursione in un linguaggio di programmazione non richiede alcun meccanismo sintattico addizionale, purché esista la possibilità di dare un nome ad un dato costrutto linguistico (che in generale è composto). Questa facoltà è

intrinseca al concetto di sottoprogramma (sia come procedura che come funzione) e un sottoprogramma è recursivo se chiama se stesso, direttamente o indirettamente (cioè richiamando un altro sottoprogramma). Se esiste questa possibilità, allora restano solo problemi semantici o implementativi.
La semantica deve fornire un mezzo per immagazzinare i dati usati in un'attivazione di un sottoprogramma indipendentemente dai dati relativi ad altre attivazioni. Nel Pascal il meccanismo corrispondente è garantito da variabili locali, create all'attivazione del sottoprogramma e presenti durante l'intero processo di attivazione. Se diverse attivazioni dello stesso sottoprogramma possono coesistere durante una chiamata recursiva, a ciascuna attivazione viene applicato un insieme di variabili locali. Il contesto di ciascuna attivazione viene perciò salvato e ristabilito in punti adeguati e non esiste possibilità di comunicazione fra le diverse attivazioni di uno stesso sottoprogramma.
L'implementazione deve assicurare che un simile meccanismo operi in maniera corretta ed efficiente. Il problema dell'implementazione dei sottoprogrammi recursivi infatti non è ancora perfettamente controllato da chi produce compilatori, ed è incluso nella soluzione generale data al problema dell'implementazione delle variabili locali e delle chiamate di sottoprogrammi. La recursione non aggiunge costi ulteriori all'implementazione di sottoprogrammi che vengono implementati in maniera molto efficiente sui calcolatori attualmente diffusi. Per queste ragioni, la scarsa efficienza dell'implementazione non dovrebbe mai essere addotta come scusa per eliminare la recursione; la scelta fra una soluzione iterativa e una recursiva, in Pascal, è dovuta soltanto alle proprietà dell'algoritmo che viene programmato.
Le stesse cose non si possono sempre dire per i vecchi linguaggi di programmazione; la definizione ufficiale del Fortran non prescrive alcuna scelta per la relazione fra l'esistenza di una variabile e l'attivazione del sottoprogramma in cui viene definita. Tutti i compilatori sono concordi nella stessa scelta, che viene ora considerata parte del Fortran: tutte le variabili esistono per tutta la durata del programma, non sono correlate ad una particolare attivazione e, di conseguenza, la recursione è impossibile. Anche il Cobol, ufficialmente, ha operato la stessa scelta, e per molto tempo la recursione è stata considerata un puro esercizio accademico. Gli autori del PL/1 hanno riconosciuto finalmente l'importanza della recursione, che era nota da lungo tempo (era usata nell'Algol 60) ma, spaventati dalla possibile inefficienza implementativa, l'hanno resa facoltativa. Di fatto, l'uso della recursione nel PL/1 è generalmente scoraggiato a causa dei suoi presunti alti costi.
Dicevamo prima che non è necessario alcun meccanismo sintattico particolare per l'applicazione delle procedure recursive. L'unico problema, anche se di secondaria importanza, si incontra nel caso della mutua recursione, cioè quando due procedure si richiamano a vicenda. Poiché la dichiarazione di qualsiasi identificatore deve precedere tutti i suoi usi, è necessario ricorrere ad un trucco quando la procedura A richiama la procedura B, che a sua volta richiama A. La soluzione è data dalla direttiva standard *forward*, di cui si è già trattato nel paragrafo 4.1. La procedura A, ad esempio, viene predichiarata per prima; ap-

pare cioè solo la sua intestazione, e il suo blocco è sostituito da *forward*. A questo punto la procedura B può essere dichiarata completamente e può usare A che è già stata dichiarata. Il corpo di A appare in un secondo tempo, completando la dichiarazione di A, e può servirsi di B, già dichiarata. Si ricordi che, quando il corpo di una procedura è separato dalla sua intestazione, è preceduto da una semplice identificazione di procedura. I parametri della procedura, se ne esistono, sono specificati solo una volta, nell'intestazione. Nel caso di diverse procedure mutuamente recursive, inoltre, un sistema più preciso consiste nel predichiarare dapprima tutte le procedure aventi questa relazione reciproca con commenti adeguati, e dichiarare poi i rispettivi blocchi in qualsiasi ordine.
La restante parte di questo capitolo sarà dedicata ad esempi di programmi recursivi; anzitutto, due esempi mostrano situazioni in cui la recursione non è la scelta migliore, in quanto una soluzione iterativa sarebbe più facile ed efficiente. In seguito, esempi più validi dimostrano come la recursione possa essere usata come strumento semplice, elegante ed efficiente in numerose situazioni.

ESEMPIO 14.1: FATTORIALI RECURSIVI E ITERATIVI

```
function FattorialeRecursivo(n: naturali): naturali;
begin {FattorialeRecursivo}
   if n = 0 then FattorialeRecursivo := 1
   else FattorialeRecursivo := n * FattorialeRecursivo(n-1)
end {FattorialeRecursivo};
function Fattoriale(n: naturali): naturali;
   var f, i: naturali;
begin {Fattoriale}
   i := 0; f := 1;
   while i ≠ n do {f = i!}
   begin i := i+1; f := i * f end {f = n!};
   Fattoriale := f
end {Fattoriale}
```

COMMENTI

1. La prima versione di questa funzione di calcolo del fattoriale è ricavata direttamente dalla definizione recursiva data dai matematici. Invertendo l'ordine in cui sono calcolati i successivi fattoriali, si giunge alla seconda versione. La prima è apparentemente più semplice perché non deve ricorrere a variabili locali e non contiene iterazioni. L'iterazione è infatti nascosta dalla chiamata recursiva e sono necessarie n attivazioni al fine di calcolare il fattoriale n-esimo. Se le chiamate di sottoprogrammi sono implementate in modo corretto, il costo in termini di tempo non è probabilmente proibitivo;

ma lo è il costo in termini di spazio occupato, poiché lo stack di attivazione deve memorizzare il contesto della funzione *n* volte.

2. La versione recursiva mette in risalto una situazione in cui la recursione si può evitare molto semplicemente, cioè in tutti i casi in cui si ha una sola chiamata recursiva nel sottoprogramma e questa non è seguita da altre operazioni. In simili casi una versione iterativa è molto semplice da ottenere, e riduce enormemente i costi del sottoprogramma.
3. Si notino i diversi significati che ha l'occorrenza del nome della funzione nel suo stesso corpo, nel caso di *FattorialeRecursivo* e di *Fattoriale*. In due sole situazioni questo fatto non implica una chiamata alla funzione: nell'intestazione della funzione (ovviamente) e quando il nome compare nella parte sinistra di un'istruzione di assegnamento. Ciò spiega la necessità della variabile locale *f* nella funzione *Fattoriale*.

ESEMPIO 14.2: NUMERI DI FIBONACCI RECURSIVI E ITERATIVI

```
function FibonacciRecursivo(n: naturali): naturali
  {questa funzione calcola l'n-esimo numero di Fibonacci, definito dalla seguente
      relazione recursiva: F(i+1) = F(i)+F(i-1) per i > 0; F(1) = 1;
      F(0) = 0};
begin {FibonacciRecursivo}
  if n = 0 then FibonacciRecursivo := 0
  else if n = 1 then FibonacciRecursivo := 1
  else FibonacciRecursivo :=
    FibonacciRecursivo(n-1)+FinonacciRecursivo(n-2)
end {FibonacciRecursivo};
function Fibonacci(n: naturali): naturali;
  var fib, fibprecedente, temp, i: naturali;
begin {Fibonacci}
  i := 1; fib := 1; fibprecedente := 0;
  if n = 0 then fib := 0 else
  while i ≠ n do
  begin {fib = Fibonacci(i); fibprecedente = Fibonacci(i-1)}
    temp := fib; i := i+1;
    fib := fib+fibprecedente; fibprecedente := temp
  end {fib = Fibonacci(n); fibprecedente = Fibonacci(n-1)};
  Fibonacci := fib
end {Fibonacci}
```

COMMENTI

1. Anche qui la versione recursiva si deduce immediatamente dalle relazioni matematiche. La versione iterativa è, al contrario, meno evidente, poiché ci

sono due chiamate recursive della funzione, una dopo l'altra. È comunque sufficiente invertire l'ordine in cui i valori sono calcolati.

2. La versione recursiva è estremamente inadeguata, a causa del numero di chiamate recursive necessarie e anche per via delle ripetizioni di calcoli già eseguiti. Ad esempio, *F5* ha bisogno di *F4*, poi di *F3* e *F2* e così via, per un totale di 15 chiamate per poter calcolare solo quattro valori differenti. La massima profondità possibile delle chiamate di funzioni è n e il numero di chiamate recursive è circa k^n, con $k \cong 1.67$ per i primi valori interi di n.

L'insegnamento da trarre dai due esempi precedenti si può sintetizzare nella seguente regola: dato un problema per il quale è applicabile una soluzione recursiva, questa soluzione deve essere scartata a favore di una soluzione iterativa se si verifica una delle seguenti situazioni:

a) la soluzione iterativa è evidente;
b) uno studio della soluzione recursiva dimostra che la massima profondità della recursione è nell'ordine di n o più, oppure il numero di chiamate recursive è nell'ordine di $n \log_2 n$ o più, e si trova una soluzione iterativa.

In certe situazioni, una soluzione recursiva e una iterativa allo stesso problema possono rispettivamente mostrare vantaggi e svantaggi; è difficile poter affermare con certezza che una delle due soluzioni è in assoluto più semplice, chiara, efficiente, elegante e comprensibile dell'altra e la decisione fra queste possibilità è soprattutto una questione di gusti. È il caso del seguente esempio.

ESEMPIO 14.3: SCRITTURA DI INTERI ITERATIVA E RECURSIVA

```
procedure ScritturaInteri (var f: text; n: integer)
   {questa procedura scrive su f una rappresentazione decimale del numero n, utilizzando il minimo numero di caratteri};
   const maxcifre = 7 {numero massimo di cifre per un intero:
         maxcifre = log10 maxint + 1};
      base = 10;
   var cifre: array[1..maxcifre] of '0'..'9';
      indice: 0..maxcifre;
begin {ScritturaInteri}
   indice := maxcifre;
   if n < 0 then {scrive il segno meno}
   begin write (f, '-'); n := -n end;
   {0 ≤ n < 10^maxcifre}
   repeat {fino a che non ci sono più cifre da calcolare}
      cifre[indice] := chr(n mod base + ord('0'));
      n := n div base; indice := indice - 1
   until n = 0;
   repeat {fino a che non ci sono più cifre da scrivere}
```

```
        indice := indice + 1;
        write(f, cifre[indice])
    until indice = maxcifre
end {ScritturaInteri};
procedure ScritturaInteriRecursiva (var f: text; n: integer);
    const base = 10;
begin {ScritturaInteriRecursiva}
    if n < 0 then {scrive il segno meno}
    begin write (f, '-'); n := -n end;
    if n ≥ base then {scrive la prima cifra}
    ScritturaInteriRecursiva (f, n div base);
    write(f, chr(n mod base + ord('0'))) {l'ultima cifra}
end {ScritturaInteriRecursiva}
```

COMMENTI

1. Un parametro passato per valore è locale alla procedura, mentre un parametro di tipo variabile fa riferimento al corrispondente parametro attuale. Infatti, nella procedura *ScritturaInteriRecursiva* c'è una copia di *n* per ciascuna attivazione, mentre il riferimento al parametro attuale di tipo textfile è lo stesso a tutti i livelli di recursione.
2. La versione recursiva richiede una profondità di recursione pari al numero di cifre da scrivere, mentre la versione iterativa richiede un array locale, perché non si devono scrivere; inoltre, la versione iterativa è chiaramente più complicata. Tuttavia, la procedura recursiva sarebbe molto meno attraente se la specifica della procedura avesse richiesto di allineare a destra l'intero in un campo di una data lunghezza, come accade con la procedura predefinita *write*.

I tre esempi che seguono mostrano situazioni in cui una soluzione recursiva non è solo concettualmente semplice, ma anche efficiente. Ciò accade in molti contesti differenti e altri casi saranno trattati negli esercizi.

ESEMPIO 14.4: ORDINAMENTO VELOCE

```
procedure OrdinamentoVeloce(var tabella: array[basso..alto: integer] of T)
    {questa procedura ha le stesse specifiche della OrdinamentoDiShell dell'Esempio
        10.3, ma fa uso del miglior algoritmo noto a tutt'oggi per l'ordinamento
        interno, dovuto a C.A.R. Hoare};
    procedure Partizione(sinistro, destro: integer {sinistro < destro};
            var primo, secondo: integer)
        {questa procedura locale esegue una partizione del sottoarray tabella[sinistro..
            destro] attorno alle posizioni primo e secondo, che costituiscono il risulta-
```

```
        to, cioè permuta gli elementi della tabella fino a che non sono valide le se-
        guenti relazioni:

            primo < secondo
            tabella[i] ≤ tabella[j] per i in sinistro..destro − 1 e j in primo + 1..destro

        Ciò implica che esiste un x tale che

            tabella[i] < x con i in sinistro..primo
            tabella[i] = x con i in primo + 1..secondo − 1
            tabella[i] > x con i in secondo..destro

        Di conseguenza gli elementi in tabella, compresi fra primo e secondo, so-
        no già nella loro posizione finale};
      var x, z: tipoelemento;
    begin {Partizione}
      x := tabella[(sinistro + destro) div 2] {sceglie l'elemento centrale come as-
      se della partizione}
      primo := destro; secondo := sinistro;
      repeat {fino a che secondo > primo}
        {tabella[i] ≤ x per i in sinistro..secondo − 1,
        tabella[i] ≥ x per i in primo + 1..destro,
        secondo < primo}
        while MinoreDi(tabella[secondo], x) do secondo := secondo + 1;
        while MinoreDi(x, tabella[primo]) do primo := primo − 1;
        if secondo ≤ primo then {tabella[secondo] ≥ x, tabella[primo] ≤ x}
          begin {scambia tabella[secondo] e tabella[primo]}
            z := tabella[secondo]; tabella[secondo] := tabella[primo];
            tabella[primo] := z;
            secondo := secondo + 1; primo := primo − 1
          end
      until secondo > primo
    end {Partizione};
    procedure Ordinamento(sinistro, destro: integer {sinistro < destro})
      {questa procedura locale, che è recursiva, ordina il sottoarray tabella[sinistro..
        destro] chiamando ripetutamente Partizione};
      var nuovosinistro, nuovodestro: integer;
    begin {Ordina}
      Partizione(sinistro, destro, nuovodestro, nuovosinistro)
        {tabella[sinistro..nuovodestro] e tabella[nuovosinistro..destro], se non sono
          vuote, possono essere ordinate separatamente;
          tabella[nuovodestro + 1..nuovosinistro − 1] è già nello stato finale};
      if sinistro < nuovodestro then Ordinamento(sinistro, nuovodestro);
      if nuovosinistro < destro then Ordinamento(nuovosinistro, destro)
      {tabella[k] ≤ tabella[l] per ogni k < l tali che k ed l siano in sinistro..destro}
    end {Ordinamento}
  begin {OrdinamentoVeloce}
```

```
    Ordinamento(basso, alto)
  end {OrdinamentoVeloce}
```

COMMENTI

1. Uno studio accurato di questo algoritmo va al di là degli intenti del libro. Si osservi comunque come *OrdinamentoVeloce* implichi due importanti svantaggi:
   a) la media dei suoi risultati è eccellente ($n \log_2 n$ confronti), ma decresce fino a livelli del tutto insoddisfacenti se l'array di partenza è già ordinato o quasi ordinato;
   b) l'ordinamento risultante non è stabile, cioè non si è in grado di garantire che due componenti uguali compaiono fra i dati ordinati nello stesso ordine che avevano inizialmente. Ciò può creare degli inconvenienti, se il tipo *T* contiene campi che non intervengono nella relazione di ordinamento data da *MinoreDi*, ma devono essere ordinati con un criterio secondario.
2. Un importante miglioramento si può ottenere con facilità usando la recursione solo per il sottoarray più corto data da *Partizione* e usando un'iterazione per il più lungo. Ciò riduce la massima profondità di recursione che altrimenti è, nel peggiore dei casi, uguale alla lunghezza dell'array. La procedura *Ordinamento* diventa quindi:

```
procedure Ordinamento2(sinistro, destro: integer);
  var nuovosinistro, nuovodestro: integer; finito: Boolean;
begin {Ordinamento2}
  finito := false;
  repeat
    Partizione(sinistro, destro, nuovodestro, nuovosinistro);
    if nuovosinistro - destro ≤ destro - nuovosinistro then
    begin {il sottoarray di sinistra è il più corto}
      if sinistro < nuovodestro
        then Ordinamento2(sinistro, nuovodestro);
      if nuovosinistro < destro then sinistro := nuovosinistro
      else finito := true
    end else
    begin {il sottoarray di destra è il più corto}
      if nuovosinistro < destro
        then Ordinamento2(nuovosinistro, destro);
      if sinistro < nuovodestro then destro := nuovodestro
      else finito := true
    end
  until finito
end {Ordinamento2}
```

Utilizzando questo espediente, la massima profondità di recursione è pari al $\log_2$ della lunghezza dell'array, il che è perfettamente accettabile.

ESEMPIO 14.5: STAMPA DELL'ALBERO GENEALOGICO (MIGLIORATO)

```
{date le definizioni di tipo dell'Esercizio 13.1, la seguente procedura è uno strumento
   comunemente usato per applicare alcuni processi ad ogni persona di una data ge-
   nealogia; vengono elaborati anzitutto i dati relativi alla persona, seguiti da quelli
   del ramo paterno e del ramo materno}
procedure ScandisceAlbero
      (persona: genitore {la genealogia da elaborare};
      procedure ElaboraUnNodo {il processo da applicare}
         (nodo: genitore {il nodo da elaborare};
         profondita: naturali {la profondità di recursione, forse utile}
         )
      );
   procedure Attraversa {nell'ordine nodo-padre-madre}
         (nodo: genitore; profondita: naturali);
   begin {Attraversa}
      if nodo ≠ nil then
      begin {il nodo esiste}
         ElaboraUnNodo(nodo, profondita);
         with nodo↑ do
         begin {chiamata recursiva per i rami padre e madre}
            Attraversa(padre, profondita+4);
            Attraversa(madre, profondita+4)
         end
      end
   end {Attraversa};
begin {ScandisceAlbero}
   Attraversa(persona, 4)
end {ScandisceAlbero};
{la procedura dell'Esempio 13.1 può ora essere scritta in modo molto semplice; non
   c'è più bisogno del secondo parametro}
procedure StampaAlbero(persona: genitore);
   procedure StampaUnNodo(nodo: genitore; incolonnamento: naturali)
      {questa procedura ausiliaria esegue l'elaborazione necessaria per ogni dato
         nodo};
   begin {StampaUnNodo}
      with nodo↑ do {nodo ≠ nil, grazie ad Attraversa}
         writeln(output, ' ': incolonnamento, nomeproprio, ' ', cognome)
   end {StampaUnNodo};
begin {StampaAlbero}
   ScandisceAlbero(persona, StampaUnNodo)
end {StampaAlbero}
```

COMMENTI

1. La procedura *ScandisceAlbero* è completamente generale, nel senso che può effettuare qualsiasi azione sulla genealogia, tranne una che potrebbe cancellarla. Benché il primo parametro sia un parametro passato per valore, esso può essere usato per modificare la struttura dell'albero genealogico, il che non è raccomandabile. Cambiando l'ordine rispettivo della chiamata *ElaboraUnNodo* o delle due chiamate recursive di *Attraversa* nella procedura *Attraversa*, si può cambiare l'ordine in cui l'albero viene visitato.
2. La procedura *ScandisceAlbero* presenta una caratteristica generale che si incontra in molte soluzioni recursive, comparsa anche nell'esempio precedente: la procedura principale serve solo per richiamare una procedura recursiva locale, che esegue tutto il lavoro. Questa è la maniera più semplice di fare le necessarie inizializzazioni.

ESEMPIO 14.6: CALCOLATORE DA TAVOLO

```
{Il seguente esempio incompleto simula un calcolatore da tavolo, leggendo e interpretando direttamente particolari istruzioni di assegnamento con sintassi semplificata. Si assume che siano disponibili le seguenti procedure:
   — SimboloSuccessivo legge il successivo simbolo significativo digitato dall'utente, ignorando gli spazi, le tabulazioni, i «return» e così via. Questo simbolo è assegnato alla variabile globale di tipo carattere denominata simbolo.
   — Costante (var valore: real) legge una costante reale e l'assegna al suo parametro.
   — Errore stampa un messaggio di errore e sospende l'operazione.
   Tutte le variabili sono di tipo reale, e hanno per nome una delle 26 lettere minuscole. La variabile globale varvalore: array['a'..'z'] of real contiene i valori correnti di tutte le variabili. La sintassi accettata da ciascuna delle seguenti procedure è indicata nel commento che segue la relativa predichiarazione. Le parentesi quadre racchiudono un'entità opzionale, mentre le parentesi angolari racchiudono un'entità che può essere ripetuta zero o più volte}
procedure Assegnamento
   {assegnamento = variabile '=' espressione ';'}; forward;
procedure Espressione (var valore: real)
   {espressione = ['+'|'-'] fattore < ('+'|'-') fattore >}; forward;
procedure Fattore (var valore: real)
   {fattore = termine < ('*'|'/') termine >}; forward;
procedure Termine (var valore: real)
   {termine = variabile|costante|'(' espressione ')'}; forward;
procedure Assegnamento;
   var nome: char;
begin {Assegnamento}
   if not (simbolo in lettere) then Errore
```

```
    else begin nome := simbolo; SimboloSuccessivo end;
    if simbolo ≠ '=' then Errore
    else SimboloSuccessivo;
    Espressione(varvalore[nome]);
    if simbolo ≠ ';' then Errore
    else begin writeln(output, varvalore[nome]: 8: 2);
      SimboloSuccessivo end
end {Assegnamento};
procedure Espressione {si veda la predichiarazione};
    var operandodestro: real; operatore: char;
begin {Espressione}
    if simbolo in ['+', '-'] then {segno iniziale}
    begin operatore := simbolo; SimboloSuccessivo end
    else operatore := '+';
    Fattore(valore);
    if operatore = '-' then valore := -valore;
    while simbolo in ['+', '-'] do {operatore di somma}
    begin operatore := simbolo; SimboloSuccessivo;
      Fattore(operandodestro);
      if operatore = '+' then valore := valore+operandodestro
      else valore := valore-operandodestro
    end {operatore di somma}
end {Espressione};
procedure Fattore {si veda la predichiarazione};
    var operandodestro: real; operatore: char;
begin {Fattore}
    Termine(valore);
    while simbolo in ['*', '/'] do {operatore di moltiplicazione}
    begin operatore := simbolo; SimboloSuccessivo;
      Termine(operandodestro);
      if operatore = '*' then valore := valore * operandodestro
      else valore := valore/operandodestro
    end {operatore di moltiplicazione}
end {Fattore};
procedure Termine {si veda la predichiarazione};
begin {Termine}
    if simbolo in lettere then
    begin valore := varvalore[simbolo]; SimboloSuccessivo end
    else if simbolo in cifre then
    begin Costante (valore); SimboloSuccessivo end
    else if simbolo := '(' then
    begin SimboloSuccessivo; Espressione (valore);
      if simbolo ≠ ')' then Errore
      else SimboloSuccessivo
      end
```

```
    else Errore
end {Termine};
```

COMMENTI

Scrivere un simile insieme di procedure mutuamente recursive è decisamente semplice. Il solo problema è il procedimento richiesto in caso di errore. A confronto, un programma equivalente, che non si avvalga della recursione, sarebbe molto meno leggibile e molto più complicato, per ottenere risultati probabilmente equivalenti. Infatti, buona parte dei compilatori del Pascal sono scritti in Pascal e usano una generalizzazione di questo metodo.

ESERCIZI

14.1 Cross-reference (modificato)

Il problema da risolvere è lo stesso dell'Esempio 13.3, ma il dizionario delle parole è organizzato in modo tale che viene riordinato dopo l'aggiunta di ogni nuova parola. La tabella a lunghezza fissa (dizionario variabile) e la sua organizzazione come tabella hash viene abbandonata. La tabella è costruita dinamicamente e strutturata come un albero binario. Un elemento della tabella è del seguente tipo:

```
type puntaoggetto = ↑oggetto;
    oggetto = record
        nome: tipoparola {la parola};
        testalista, codalista: puntaelemento {la sua lista di riferimento};
        prima, dopo: puntaoggetto {sottoalbero che precede o segue nome in ordine lessicale}
    end;
```

Tutti gli oggetti che appaiono nel sottoalbero con radice *prima* (*dopo*) contengono nomi che precedono (seguono) in ordine alfabetico il nome in esame.
Si scrivano le procedure *RicercaEInserisce* e *StampaIlDizionario*, con queste nuove specifiche, modificando di conseguenza il programma *CrossReference* (si noti che la procedura *OrdinaIlDizionario* non è più necessaria). Questa soluzione, che assomiglia ad una ricerca binaria in un array, dà un risultato altrettanto efficiente di una ricerca binaria vera e propria?

14.2 Miglior scelta

Un negoziante vuole dare a tutti i suoi clienti lo stesso pacco-dono, come regalo per il nuovo anno. Il pacco-dono ha numerosi oggetti, presi da un insieme di oggetti possibili. Ogni oggetto ha tre caratteristiche: il suo costo, il suo fascino come dono e una lista di oggetti che sono incompatibili, cioè oggetti che non possono essere aggiunti al pacco, se l'oggetto in esame è già presente.
Si scriva un programma che, dai dati sull'insieme degli oggetti possibili, numerati da *1* a

n e dalle loro caratteristiche, trovi l'insieme ideale per il negoziante, cioè quell'insieme con un prezzo inferiore o uguale a un dato *prezzomassimo*, che abbia massima attrazione per i clienti.

*14.3 Invio di un messaggio

Nell'Esercizio 13.1, una serie di stazioni meteorologiche inviavano, ad una stazione principale ricevente, messaggi di lunghezza variabile, nella forma di pacchetti di lunghezza fissa *l*. Si è interessati alla trasmissione dei pacchetti e alla gestione di possibili errori di trasmissione, sulla linea di comunicazione. Il protocollo scelto è molto semplice: una trasmittente invia un pacchetto sotto forma di sequenza di caratteri, ma non invia il carattere successivo, fino a che dalla stazione ricevente non viene rinviata un'eco del carattere. Se l'eco di rimando è sbagliata, siamo in presenza di un errore e la stazione trasmittente invia un carattere di cancellazione (per esempio '#'), poi ripete il carattere giusto. Date le seguenti procedure:

```
procedure InviaCarattere (ch: char) {invia ch sulla linea}
procedure RiceveCarattere (var ch: char) {riceve ch dalla linea}
```

si scriva la procedura

```
procedure InviaPacchetto (p: pacchetto)
```

che trasmette il pacchetto *p*, secondo il protocollo precedente, data la definizione di tipo:

```
type pacchetto: array[1..l] of char
```

A

Soluzione di esercizi scelti

5.2 Stampa di un titolo

```
procedure StampaTitolo (titolo: char)
    {stampa un titolo personale, identificato da una lettera};
begin {StampaTitolo}
    case titolo of
        'A': write ('Architetto');
        'V': write ('Avvocato');
        'I': write ('Ingegnere');
        'D': write ('Dottore');
        'P': write ('Professore');
        'G': write ('Geometra');
        'R': write ('Ragioniere')
    end {case titolo}
end {StampaTitolo}
```

COMMENTI

Se non si può garantire che il parametro *titolo* assuma solo valori legali, bisogna proteggere l'istituzione **case** da assegnamenti illegali: in caso contrario, il programma abortirà. Per fare ciò si può utilizzare un'istruzione **if**:

```
if (titolo = 'A') or (titolo = 'V') or ... or (titolo = 'R') then
    case titolo of ...
    end {case titolo}
else write ('titolo errato')
```

Sarebbe meglio però usare un costruttore di set costante (si veda il Capitolo 12), come nella seguente istruzione **if**:

```
if titolo in ['A', 'V', 'I', 'D', 'P', 'G', 'R'] then
    case titolo of...
```

```
    end {case titolo}
else write('titolo errato')
```

6.3 Radici di un'equazione

```
procedure RadiciDiErone
        (xzero: real {approssimazione iniziale della radice};
        function F(x: real): real {la funzione vera e propria};
        function Fprimo(x: real): real {la sua derivata};
        epsilon: real {la precisione relativa desiderata};
        var convergenza: Boolean {vale true se la precisione desiderata è ottenuta prima del
            massimo numero di iterazioni previsto};
        var soluzione: real {la radice desiderata, se convergenza è true}
        )
    {Calcola la radice di f(x) = 0 usando il metodo di Erone di Alessandria};
    const maxiterazioni = 20 {numero massimo di iterazioni};
    var xi, xipiu1: real {due valori consecutivi della radice calcolata};
        stato: (incalcolo, precisioneraggiunta, troppeiterazioni);
        numiterazioni: integer;
begin {RadiciDiErone}
    xipiu1 := xzero; numiterazioni := 0;
    stato := incalcolo;
    repeat {fino a che stato ≠ incalcolo}
        xi := xipiu1; numiterazioni := numiterazioni + 1;
        xipiu1 := xi − (F(xi)/Fprimo(xi));
        if abs((xipiu1 − xi)/xi) < epsilon then
            stato := precisioneraggiunta
        else if numiterazioni = maxiterazioni then
            stato := troppeiterazioni
    until stato ≠ incalcolo;
    case stato of
        precisioneraggiunta: convergenza := true;
        troppoeiterazioni: convergenza := false
    end {case stato};
    soluzione := xipiu1
end {RadiciDiErone};
```

COMMENTI

Si noti l'uso della variabile *stato* come nell'Esempio 6.6 per discriminare le due possibili condizioni di uscita alla fine del ciclo.

7.1 Elenco di elementi di un file

```
procedure Elenco (var f: filediinteri; mese, anno: integer)
    {fornisce l'elenco di tutti i codici dei prodotti i cui pezzi unitari non sono più stati aggiornati
        da (mese, anno)};
```

```
    var codiceprodotto, prezzounitario, quantita, m, a: integer
        {le cinque informazioni che descrivono il prodotto};
    function NonAggiornato: Boolean
        {vale true se (m, a) precede (mese, anno)};
    begin {NonAggiornato}
        NonAggiornato := (a < anno) or (a = anno) and (m < mese)
    end {NonAggiornato};
begin {Elenco}
    reset(f);
    {f1 è il primo codice di prodotto se ¬eof(f)}
    writeln ('***Elenco di tutti i codici di prodotto il cui prezzo',
            'unitario non è stato aggiornato da', anno, '/',
            mese, '***');
    while not eof(f) do
    begin {elabora un prodotto}
        read (f, codiceprodotto, prezzounitario, quantita, m, a);
        if NonAggiornato then {lo inserisce nell'elenco}
        write (codiceprodotto)
        {se a era l'ultimo intero di f, allora eof(f) vale true}
    end;
    writeln
end {Elenco};
```

COMMENTI

1. Nel caso di dati non validi (gli interi in *f* non sono logicamente raggruppati a gruppi di 5) questa procedura causa un errore, poiché la verifica sull'end-of-file viene effettuata soltanto dopo aver letto una serie completa di cinque interi. Una soluzione migliore sarebbe quella di strutturare *f* come un file di record (si veda il Capitolo 11).
2. La stampa dei risultati è brutta perché non si possono ancora usare gli strumenti che saranno presentati nel Capitolo 8.

8.5 Confronto di file

```
procedure ConfrontaFile
        (var rif, verif: text;
        var nuovorif: text
        )
    {confronta il file di riferimento rif con verif e produce un nuovo file di riferimento nuovorif};
    var numrif, numnuovorif, numverif: integer
        {numero di linea sui tre file};
    procedure Inizializza
        {inizializza tutte le variabili, file compresi};
    begin {Inizializza}
        reset(rif); reset(verif); rewrite(nuovorif);
        numnuovorif := 0;
        writeln (output, '***Confronto di due file',
            'eseguito dalla procedura ConfrontaFile***')
```

```
end {Inizializza};
procedure LeggeNum (var f: text; var numlinea: integer)
    {legge il numero di linea numlinea sul file f, se esiste};
begin {LeggeNum}
    {siamo all'inizio di una nuova linea, o alla fine del file}
    if not eof(f) then read (f, numlinea)
end {LeggeNum};
procedure Cancella
    {cancella una linea dal file rif e la lista in output con un appropriato messaggio};
begin {Cancella}
    writeln (output, 'la linea numero', numrif: 1, 'e" stata cancellata',
        'Questa linea e":');
    while not eoln(rif) do
    begin output↑ := rif↑; put(output); get(rif) end;
    readln (rif); writeln (output)
end {Cancella};
procedure Inserisce
    {inserisce una nuova linea in nuovorif e la scrive in output};
begin {Inserisce}
    writeln (output, 'Alla linea', numrif: 1, 'del file di riferimento',
        'e" stata inserita la linea seguente:');
    while not eoln(verif) do
    begin output↑ := verif↑; put(output);
        nuovorif↑ := verif↑; put(nuovorif);
        get(verif)
    end;
    readln (verif); writeln (output)
end {Inserisce};
procedure ScriveNum
    {aggiorna e scrive il numero di linea nel file nuovorif};
begin {ScriveNum}
    numnuovorif := numnuovorif + 1;
    write (nuovorif, numnuovorif: 5, ' ')
end {ScriveNum};
procedure ConfrontaLinee
    {confronta i caratteri di rif e di verif e copia quelli di verif in nuovorif; dopo la prima
        discrepanza, i caratteri restanti di ciascun file vengono listati in output};
    var contacarrif, contacarverif: integer
        {contatori dei caratteri di rif e verif};
        stato: (uguali, diverse, finelinee);
begin {ConfrontaLinee}
    contacarrif := 1;
    {rif↑ e verif↑ sono i primi caratteri da confrontare}
    stato := uguali;
    while stato = uguali do
        {fino ad ora le linee sono uguali}
        if eoln(rif) then
            if eoln(verif) then stato := finelinee
            else stato := diverse
        else if rif↑ ≠ verif↑ then stato := diverse
```

```
                else begin {ancora un carattere uguale}
                    nuovorif↑ := verif↑; put(nuovorif);
                    get(rif); contacarrif := contacarrif + 1;
                    get(verif)
                end;
            if stato = diverse then
            begin
                writeln (output, 'Differenza alla linea', numrif: 1, 'a partire dal',
                    contacarrif: 1, '-esimo carattere');
                contacarverif := contacarrif;
                writeln (output, 'Il resto della linea di riferimento e'':');
                while not eoln(rif) do
                begin output↑ := rif↑; put(output);
                    get(rif); contacarrif := contacarrif + 1
                end;
                readln(rif); writeln (output);
                writeln (output, 'Il resto della linea da verificare e'':');
                while not eoln(verif) do
                begin {copia in nuovorif e in output}
                    output↑ := verif↑; put(output);
                    nuovorif↑ := verif↑; put (nuovorif);
                    get(verif); contacarverif := contacarverif+1
                end;
                readln(verif); writeln(output);
                writeln(nuovorif);
                wirteln(output, 'Differenza di lunghezza fra le due linee = ',
                    abs(contacarverif − contacarrif));
            end {if stato = diverse}
            else {uguaglianza completa}
            begin readln (rif); readln(verif) end
            {un eoln viene scritto in nuovorif e letto da rif e verif}
    end {ConfrontaLinee};
begin {ConfrontaFile}
    Inizializza;
    LeggeNum(rif,numrif); LeggeNum(verif, numverif);
    while (not eof(rif) and not eof(verif)) do
        {inizio di linea dopo il numero di linea in rif e verif e prima del numero di linea
            in nuovorif}
        if numrif = numverif then
        {forse una modifica nella linea corrente}
        begin ScriveNum; ConfrontaLinee;
            LeggeNum(rif, numrif); LeggeNum(verif, numverif);
        end
        else if numrif < numverif then {cancella una linea}
            begin Cancella; LeggeNum(rif, numrif) end
            else if numverif = 0 then {inserisce una linea}
                begin ScriveNum; Inserisce;
                    LeggeNum(verif, numverif)
                end;
        {in tutti i casi, o siamo all'inizio di una linea nei file rif e verif dopo il numero di
            linea, oppure eof vale true}
```

```
    {è possibile che uno dei due file non sia terminato}
    while not eof(rif) do
    begin Cancella; LeggeNum(rif, numrif) end;
    while not eof(verif) do
    begin ScriveNum; Inserisce; LeggeNum(verif, numverif) end;
    writeln(output, '***Fine del confronto***')
end {ConfrontaFile};
```

COMMENTI

1. La procedura precedente è lunga e complessa. Si raccomanda di leggere il corpo della procedura principale prima di passare alle procedure locali che essa chiama.
2. Per risolvere un problema complesso, è molto utile frazionarlo in numerosi sottoproblemi. Al limite, la procedura *LeggeNum* contiene solo un'istruzione, per risolvere il delicato problema della fine del file.
3. Il confronto è abbastanza semplice da fare, a causa della natura dei file che si devono confrontare. Sarebbe molto diverso se le linee non fossero numerate; sarebbe facile scoprire la prima differenza, ma sarebbe impossibile trovare la fine delle sequenze differenti, senza provare parecchie soluzioni, con ritorni al testo per la verifica.

9.2 Analisi di tabelle

La tabella *tb*, che contiene due valori ad ogni entrata (una chiave di tipo *tipochiave* e un elemento di informazione di tipo *T*), è divisa in due tabelle parallele: *tb* contiene informazioni di tipo *T*, e *chiavetb* contiene le chiavi in modo tale che l'informazione con chiave *chiavetb*[*i*] sia in *tb*[*i*].
Per accedere alla tabella *chiavetb* con una chiave di lunghezza *l*, si fa una ricerca lineare fra le entrate con la stessa lunghezza di chiave. Perciò, è necessario conoscere per ogni lunghezza di chiave l'indice della prima entrata corrispondente. Questi indici di lunghezza di chiave sono memorizzati in un array *accessoatb*, tale che, se n_j è il numero di entrate per le chiavi di lunghezza *j*, valgono le seguenti relazioni:

$$accessoatb[1] = 1$$
$$accessoatb[2] = 1 + n_1$$
$$\ldots$$
$$(1)\quad accessoatb[j+1] = accessoatb[j] + n_j$$

Come caso particolare se $n_j = 0$, allora $accessoatb[j+1] = accessoatb[j]$. Per ricercare in *chiavetb* una chiave di lunghezza *j*, si dovrà fare una ricerca lineare fra *chiavetb*[*accessoatb*[*j*]] e *chiave tb*[*accessoatb*[*j* + 1] − 1]. Quando *j* è uguale a *lunghezzachiave*, questo implica *accessoatb*[*lunghezzachiave* + 1], nel qual caso la relazione (1) diventa:

$$accessoatb[lunghezzachiave + 1] = accessoatb[lunghezzachiave] + n_{lunghezzachiave}$$
$$= 1 + \sum_{j=1}^{lunghezzachiave} n_j$$

oppure (2) *accessoatb*[*lunghezzachiave* + *1*] = *lunghezzatb* + *1*
Nel modulo di gestione di *tb* sono specificate le seguenti dichiarazioni:

```
const
    lunghezzachiave = ... {lunghezza massima di una chiave};
    lunghezzaaccesso = ... {lunghezzachiave + 1};
    lunghezzatb = ... {numero di elementi in tb};
    maxaccesso = ... {lunghezzatb + 1};
type
    tipochiave = packed array[1..lunghezzachiave] of char;
    indice = 1..lunghezzatb;
var
    tb: array[indice] of T;
    chiavetb: array[indice] of tipochiave;
    accessoatb: array[1..lunghezzaaccesso] of 1..maxaccesso;
procedure Costruisce (var filetb: text)
    {costruisce tb, chiavetb e accessoatb; si assume che (a) il file di tipo text filetb contenga una
        sequenza lunga lunghezzatb di coppie (chiave, informazioni di tipo T), che sono ordinate
        in funzione della lunghezza della chiave e che (b) la procedura LeggeT legga da filetb un
        elemento di informazione di tipo T; dopo ogni elemento c'è uno spazio di separazione};
    var numerochiavi: array[1..lunghezzaaccesso] of 0..lunghezzatb
            {numero di relazioni di tipo (1)};
        i: 0..lunghezzachiave; j: 0..lunghezzatb;
        chiave: tipochiave;
begin {Costruisce}
    reset (filetb); i := 0;
    repeat {inizializza numerochiavi}
        i := i+1; numerochiavi[i] := 0
    until i = lunghezzachiave;
    j := 0;
    repeat {inizializza tb e chiavetb; calcola numerochiavi}
        j := j + 1; i := 0;
        repeat {legge la chiave}
            i := i + 1; chiave[i] := filetb↑; get(filetb)
        until filetb↑ = ' ';
        numerochiavi[i] := numerochiavi[i] + 1;
        while i < lunghezzachiave do {riempie di spazi}
        begin i := i + 1; chiave[i] := ' ' end;
        chiavetb[j] := chiave; LeggeT(tb[j])
    until j = lunghezzatb;
    {calcola accessoatb con numerochiavi}
    accessoatb[lunghezzaaccesso] := maxaccesso
        {relazione (2)};
    i := lunghezzachiave;
    repeat
        accessoatb[i] := accessoatb[i + 1] - numerochiavi[i]
            {relazione (1)};
        i := i - 1
    until i = 0
```

```
    end {Costruisce};
    function Ricerca (chiave: tipochiave; var k: indice): Boolean
        {se vale true, tb[k] è l'elemento la cui chiave è chiave; altrimenti tb[k] è indefinito};
        type lunghchiave = 1..lunghezzachiave;
        var lungh: lunghchiave; stato: (inricerca, trovato, terminato);
            i, j: 1..maxaccesso;
        function Lunghezza: lunghchiave
            {calcola la lunghezza usata per la chiave; si assume che uno spazio segua l'ultimo carattere
             di una chiave la cui lunghezza sia minore di lunghezzachiave};
            var stato: (inricerca, spazio, terminato); i: 0..lunghezzachiave;
        begin {Lunghezza}
            stato := inricerca; i := 1;
            repeat {chiave[j] ≠ ' ' per j = 1..i}
                if i = lunghezzachiave then stato := terminato
                else if chiave[i+1] = ' ' then stato := spazio
                else i := i+1
            until stato ≠ inricerca;
            Lunghezza := i
        end {Lunghezza};
    begin {Ricerca}
        {accessoatb[1] = 1 e accessoatb[lunghezzaaccesso] = lunghezzatb + 1}
        lungh := Lunghezza {1 ≤ lungh ≤ lunghezzachiave};
        i := accessoatb[lungh] {1 ≤ i ≤ lunghezzatb + 1};
        j := accessoatb[lungh + 1] - 1 {1 ≤ j ≤ lunghezzatb};
        stato := inricerca;
        repeat {fino a che stato ≠ inricerca}
            if i > j then stato := terminato
            else {1 ≤ i ≤ lunghezzatb}
                if chiavetb[i] = chiave then
                    stato := trovato
                else i := i+1
        until stato ≠ inricerca;
        Ricerca := stato = trovato; k := i
    end {Ricerca};
```

COMMENTI

1. La funzione *Lunghezza*, che è chiamata una volta soltanto, non è necessaria. Tuttavia, l'istruzione *lung* := *Lunghezza* nella funzione *Ricerca* evita il calcolo corrispondente, che maschererebbe in qualche modo il problema reale risolto da *Ricerca*. Si ricordi che la chiamata di un sottoprogramma privo di parametri è di solito molto economica in una buona implementazione del Pascal.
2. Si dovrebbero esaminare attentamente le numerose asserzioni in *Ricerca*, che dimostrano che il limite inferiore *i* della ricerca lineare può anche essere maggiore dell'estremo superiore *j*; conseguentemente è necessario un ciclo **while**. Questa situazione si verifica, per esempio, quando la tabella *chiavetb* non contiene alcuna chiave con una lunghezza maggiore di un certo valore *cmax* < *lunghezzachiave* e quando una chiamata a *Ricerca* è fatta con una chiave più lunga di *cmax*. In questo caso

accessoatb[*cmax*] = *accessoatb*[*cmax* + *1*] = ... = *accessoatb*[*lunghezzaaccesso*] = *lunghezzaaccesso*.

3. La procedura *Costruisce* abortisce se il numero di coppie su *filetb* è maggiore di *lunghezzatb* o se una chiave è più lunga di *lunghezzachiave*.
4. Il tipo *tipochiave* avrebbe potuto essere definito in un altro modo, così da associare una lunghezza alla chiave. La funzione *Lunghezza* diventerebbe inutile. Una tale associazione sarebbe facile usando tipi record (si veda il Capitolo 11).

10.1 Valutazione di un polinomio

```
function ValorePolinomio
        (var a: array[li..ls: integer] of real; x: real): real
    {calcola a[n]*x^n + a[n - 1]*x^(n-1) + ... + a[1]*x + a[0]; li = 0, ls = n};
    var i: integer; p: real;
begin {ValorePolinomio}
    p := 0.0;
    for i := ls downto li do p := p*x + a[i];
    ValorePolinomio := p
end {ValorePolinomio}
```

COMMENTI

1. Questa funzione usa la ben nota formula di Horner, che evita il calcolo di x^i, grazie alla seguente formula:

$$P = (...((a[n]*x + a[n - 1])*x + a[n - 2])*x + ... + a[1])*x + a[0]$$

2. Il tipo della variabile di controllo del ciclo deve essere lo stesso del tipo degli identificatori dei limiti. Poiché questi non sono costanti, *i* non può essere dichiarata nell'intervallo *li..ls*, che costituirebbe un identificatore di tipo errato.

11.3 Solitario

```
program Solitario(input, output);
    const numerocarte = 28;
        cartenelmazzo = 4;
    type
        tipovalore = (sette, otto, nove, dieci, fante, donna, re, asso);
        tiposeme = (fiori, quadri, cuori, picche);
        tipocarta = record seme: tiposeme; valore: tipovalore end;
    var
        errore: Boolean;
        tavolo: array[tiposeme, sette..re] of tipocarta;
        tavolodecodificato: array[tiposeme, sette..re] of
            record s, v: char end {tavolo da gioco per la stampa};
        mazzodicarte: array[1..cartenelmazzo] of tipocarta;
        cartacorrente: tipocarta;
```

```
    numcarte: 0..cartenelmazzo {numero di carte rimaste nel mazzo};
    cartescoperte: 0..numerocarte {numero di carte a faccia in su sul tavolo da gioco}
    stato: (mazzofinito, riuscito, giocoincorso);
...
procedure InizTavoloPerStampa
  {inizializza tavolodecodificato per la stampa; tutte le carte sono coperte, cioè i campi di
     tipo char hanno un punto ciascuno};
  var seme: tiposeme; valore: tipovalore;
begin {InizTavoloPerLaStampa}
  for seme := fiori to picche do
    for valore := sette to re do
      with tavolodecodificato[seme, valore] do
      begin c := '.'; v := '.' end
end {InizTavoloPerLaStampa};
procedure DecodificaUnaCarta (carta: tipocarta; var semedecodificato,
    valoredecodificato: char)
  {decodifica una carta restituendo due caratteri};
begin {DecodificaUnaCarta}
  with carta do
  begin
    case seme of
      fiori: semedecodificato := 'f';
      quadri: semedecodificato := 'q';
      cuori: semedecodificato := 'c';
      picche: semedecodificato := 'p'
  end;
  case valore of
    sette: valoredecodificato := '7';
    otto: valoredecodificato := '8';
    nove: valoredecodificato := '9';
    dieci: valoredecodificato := 'X';
    fante: valoredecodificato := 'J';
    donna: valoredecodificato := 'Q';
    re: valoredecodificato := 'K';
    asso: valoredecodificato := 'A'
    end
  end
end {DecodificaUnaCarta};
procedure StampaUnaCarta(carta: tipocarta)
  {stampa due caratteri per carta};
  var s, v: char;
begin {StampaUnaCarta}
  DecodificaUnaCarta(carta, s, v);
  writeln(output, s, v)
end {StampaUnaCarta};
procedure RegistraUnaCarta(carta: tipocarta)
  {registra una carta scoperta in tavolodecodificato};
begin {RegistraUnaCarta}
  with tavolodecodificato[carta.seme, carta.valore] do
    DecodificaUnaCarta(carta, s, v)
```

```
    end {RegistraUnaCarta};
    procedure StampaTavolo
        {stampa tavolodecodificato};
        var seme: tiposeme; valore: tipovalore;
    begin {StampaTavolo}
        for seme := fiori to picche do
        begin
            for valore := sette to re do
                with tavolodecodificato[seme, valore] do
                    write(output, s, v, ' ');
            writeln(output)
        end
    end {StampaTavolo};
begin {programma Solitario}
    InizTavoloPerLaStampa; InizDati;
    LeggeTavolo; writeln(output);
    LeggeMazzo; writeln(output);
    if errore then
        writeln(output, 'Qualcosa non va nei dati')
    else begin
        writeln(output, 'Inizio del solitario');
        writeln(output); StampaTavolo: writeln(output);
        cartescoperte := 0; stato := giocoincorso;
        numcarte := cartenelmazzo;
        repeat {fino a che non giocoincorso}
            cartacorrente := mazzodicarte[numcarte];
            writeln(output, 'La carta estratta dal mazzo e'':');
            StampaUnaCarta(cartacorrente);
            while (cartacorrente.valore ≠ asso) and (cartescoperte ≠ numerocarte)
            do begin
                RegistraUnaCarta (cartacorrente); StampaTavolo;
                writeln(output);
                cartacorrente := tavolo[cartacorrente.seme, cartacorrente.valore];
                cartescoperte := cartescoperte + 1
            end;
            if cartescoperte = numerocarte then stato := riuscito
            else begin {estrae una carta nel mazzo, se ve ne sono}
                numcarte := numcarte - 1
                if numcarte = 0 then stato := mazzofinito
            end
        until stato ≠ giocoincorso;
        case stato of
            riuscito: writeln(output, 'Il solitario e'' riuscito');
            mazzofinito: writeln(output, 'Il solitario non e'' riuscito',
                'e ci sono', numerocarte - cartescoperte: 3, 'carte coperte')
        end
    end {nessun errore nei dati}
end {Solitario}.
```

COMMENTI

1. Il tipo di una carta è molto semplice: è una coppia [*tiposeme, tipovalore*]. Il tavolo da gioco è una matrice le cui linee sono semi e le cui colonne sono sottoinsiemi di valori.
2. Tutte le procedure in questo programma tranne *LeggeTavolo* e *LeggeMazzo* sono procedure ausiliarie per stampare le carte. Tuttavia, in parallelo alla variabile *tavolo*, che contiene il seme e il valore di ogni carta, la variabile *tavolodecodificato* contiene lo stato del gioco ad un dato istante, in una forma stampabile: una coppia di caratteri se la carta è scoperta o due punti se è coperta.
3. La codifica di carte con i due tipi scalari *tiposeme* e *tipovalore* consentono dei controlli veloci con i cicli **for** degli array *tavolo* e *tavolodecodificato*. Ciò non sarebbe stato possibile con la codifica del carattere usato per stampare, poiché i caratteri non sono consecutivi.
4. La procedura *DecodificaUnaCarta* usa un'istruzione **case**, laddove un array sarebbe più naturale (**array**[*tiposeme*] **of** *char* e **array** [*tipovalore*] **of** *char*). Non c'è quasi differenza in termini di tempo e di memoria, fra le due soluzioni, tranne che l'uso di un array di codifica renderebbe necessario inizializzare esplicitamente tutti i componenti.

12.1 **Solitario** (continuazione)

```
var {si aggiunga la seguente variabile alle variabili globali dell'Esercizio 11.3}
dati: array[tiposeme] of set of tipovalore {per verificare i dati};
...
procedure LeggeUnaCarta (var carta: tipocarta)
    {legge (e scrive) triple di caratteri, verifica se questi caratteri sono corretti, codifica la carta e
     segnala, per mezzo della variabile booleana globale errore, se i caratteri non sono corretti
     e se la carta è stata letta};
    var chs, chv, separtore: char {la tripla};
    procedure CodificaUnaCarta
        {genera una carta a partire da chs e chv};
    begin {CodificaUnaCarta}
        with carta do
        begin
            case chs of
                'f': seme := fiori;
                'q': seme := quadri;
                'c': seme := cuori;
                'p': seme := picche
            end;
            case chv of
                '7': valore := sette;
                '8': valore := otto;
                '9': valore := nove;
                'X': valore := dieci;
                'J': valore := fante;
                'Q': valore := donna;
                'K': valore := re;
                'A': valore := asso
```

```
            end
        end
    end {CodificaUnaCarta};
begin {LeggeUnaCarta}
    read(input, chs, chv, separatore);
    write(output, chs, chv, ' ');
    if (chs in ['f', 'q', 'c', 'p']) and
        (chv in ['7', '8', '9', 'X', 'J', 'Q', 'K', 'A']) then
    begin {nessun errore nella tripla}
        CodificaUnaCarta;
        if carta.valore in dati[carta.seme] then {carta duplicata}
            errore := true
        else {registra la carta in dati}
            dati[carta.seme] := dati[carta.seme] + dati[carta.valore]
    end else {errore nella tripla}
        errore := true
end {LeggeUnaCarta};
procedure InizDati
    {inizializza dati con l'insieme vuoto; non si sono lette carte};
    var s: tiposeme;
begin {InizDati}
    errore := false;
    for s := fiori to picche do dati[s] := [ ]
end {InizDati};
procedure LeggeTavolo
    {inizializza tavolo leggendo 28 triple di caratteri: un seme, un valore e uno spazio di separazione; stampa i dati e pone a true la variabile globale errore se i caratteri contengono valori errati o se il mazzo stesso è sbagliato};
    var si: tiposeme; vi: tipovalore;
        carta: tipocarta;
begin {LeggeTavolo}
    writeln(output, 'Carte lette per il tavolo da gioco');
    writeln(output);
    for si := fiori to picche do
    begin
        for vi := sette to re do
            LeggeUnaCarta (tavolo[si, vi]);
        writeln(output)
    end
end {LeggeTavolo};
procedure LeggeMazzo
    {inizializza mazzodicarte leggendo quattro triple di caratteri, come nella procedura LeggeTavolo}
    var si: tiposeme; vi: tipovalore;
        i: 1..cartenelmazzo;
begin {LeggeMazzo}
    writeln(output);
    writeln(output 'Carte lette per il mazzo');
    writeln(output);
    for i := 1 to cartenelmazzo do
```

```
        LeggeUnaCarta(mazzodicarte[i]);
    writeln(output)
end {LeggeMazzo};
```

COMMENTI

1. Non ci sono difficoltà nel leggere i dati: la procedura *LeggeUnaCarta* è la controparte della procedura *DecodificaUnaCarta* dell'Esercizio 11.3. Il rilevamento dell'errore è più critico ed è diviso in due parti:
   a) il rilevamento degli errori di codifica è fatto con due insiemi «costanti» che contengono i soli caratteri legali per codificare i semi e i valori delle carte;
   b) il rilevamento di nomi illegali (carte duplicate) è fatto con l'array *dati*, indicizzato con i quattro semi, i cui elementi contengono ciascuno la serie dei valori delle carte. Inizialmente questi insiemi sono vuoti (**procedure** *InizDati*). Alla lettura di ogni carta (**procedure** *LeggeUnaCarta*), viene esaminata la coppia [*seme, valore*]: se non è già stata letta (cioè, se non è presente in *dati*[*seme*]), viene aggiunta ai dati.
2. In questo caso non si dovrebbe usare un array per la codifica. Un array definito come **array**[*'f'..'p'*] **of** *tiposeme* può occupare più spazio dell'istruzione equivalente **case** (a causa dell'inizializzazione). Inoltre, l'altra codifica non può essere definita con meno di un **array**[*char*] **of** *tipovalore*, se si desidera l'indipendenza del set dei caratteri.

13.1 Elaborazione di messaggi

```
const
    numcaratteri = ...{numero di caratteri in un pacchetto};
    terminatore = '.' {carattere di fine messaggio};
type
    pacchetto = packed array[1..numcaratteri] of char;
    puntamessaggio = ↑partedimessaggio;
    partedimessaggio = record
        dati: pacchetto;
        partesuccessiva: puntamessaggio
        end;
    puntastazione = ↑stazione;
    stazione = record {descrittore della trasmittente}
            idstazione: integer {identifica la trasmittente};
            testamessaggio, codamessaggio: puntamessaggio;
            stazionesuccessiva: puntastazione
        end;
    inviainformazione = record {informazione elementare inviata dalle trasmittenti};
            idtrasmittente: integer;
            informazione: pacchetto
        end;
    messaggi = file of inviainformazione {messaggi mischiati fra di loro};
...
procedure ElaboraMessaggi (var m: messaggi)
```

```
{tratta il file m, che contiene messaggi, mischiati fra di loro, provenienti da parecchie trasmit-
      tenti};
var trasmattive: puntastazione {testa della lista delle trasmittenti attive};
   ps, precedente: puntastazione; esiste: Boolean;
procedure IdentificaTrasmittente
      (n: integer {numero della trasmittente};
      var trovata: Boolean {true se la trasmittente n è ancora attiva};
      var ps, precedente: puntastazione {se trovato è true, ps è il puntatore alla
         trasmittente attiva n e precedente è il suo predecessore nella lista trasmattive; in
         caso contrario la trasmittente n deve venir inserita dopo precedente}
      )
   {ricerca la trasmittente n nella lista trasmattive};
   var stato: (inricerca, trasmtrovata, nessuna);
begin {IdentificaTrasmittente}
   precedente := nil; ps := trasmattive; stato := inricerca;
   repeat {fino a che non inricerca}
      if ps = nil then stato := nessuna
      else if n = ps↑.idstazione then stato := trasmtrovata
      else if n > ps↑.idstazione then stato := nessuna
      else begin precedente := ps; ps := ps↑.stazionesuccessiva end
   until stato ≠ inricerca;
   trovata := stato = trasmtrovata
   {precedente = nil ∧ ps = nil: nessuna trasmittente attiva;
   precedente = nil ∧ ps ≠ nil: la trasmittente n è la prima della lista;
   precedente ≠ nil ∧ ps ≠ nil: la trasmittente n non è la prima}
end {IdentificaTrasmittente};
procedure InserisceTrasmittente
      (n: integer {numero della trasmittente};
      precedente: puntastazione {n deve venir inserito dopo precedente};
      var ps: puntastazione {descrittore della trasmittente creata}
      )
      {crea un descrittore per la trasmittente e lo inserisce nella lista trasmattive};
begin {InserisceTrasmittente}
   new(ps);
   with ps↑ do
   begin
      idstazione := n; testamessaggio := nil; codamessaggio := nil;
      if precedente = nil then {inserisce in testa}
      begin stazionesuccessiva := trasmattive;
         trasmattive := ps
      end
      else begin stazionesuccessiva := precedente↑.stazionesuccessiva;
         precedente↑.stazionesuccessiva := ps
      end
   end
end {InserisceTrasmittente};
procedure InserisciPacchetto (p: pacchetto; ps: puntastazione)
   {inserisce il pacchetto p nel messaggio della trasmittente ps};
   var pm: puntamessaggio;
begin {InserisciPacchetto}
```

```
        new(pm);
        with pm↑ do begin dati := p; partesuccessiva := nil end;
        with ps↑ do
            if testamessaggio = nil then {primo pacchetto}
            begin testamessaggio := pm; codamessaggio := pm end
            else {inserisce in coda}
            begin codamessaggio↑.partesuccessiva := pm;
                codamessaggio := pm
            end
    end {InseriscePacchetto};
    procedure StampaMessaggio (ps: puntastazione)
        {stampa il messaggio ricevuto dalla trasmittente ps};
        var pm: puntamessaggio; i: 0..numcaratteri;
            pi: array[1..numcaratteri] of char;
    begin {StampaMessaggio}
        writeln (output, '**Messaggio dalla trasmittente numero',
            ps↑.idstazione: 4, '**');
        pm := ps↑.testamessaggio;
        if (pm ≠ nil) and (ps↑.codamessaggio ≠ pm) then
            {ci sono pacchetti completi}
            while pm ≠ ps↑.codamessaggio do
            begin write(output, pm↑.dati); pm := pm↑.partesuccessiva end;
        {stampa il pacchetto}
        i := 0; unpack(pm↑.dati, pi, 1);
        repeat {c'è almeno un carattere nell'ultimo pacchetto}
            i := i+1; write(output, pi[i])
        until pi[i] = terminatore;
        writeln(output); writeln(output)
    end {StampaMessaggio};
    procedure Elimina (precedente, ps: puntastazione)
        {elimina il messaggio ed il descrittore della trasmittente ps};
        var pprec, pp: puntamessaggio;
    begin {Elimina}
        pp := ps↑.testamessaggio; pprec := pp;
        while pp ≠ nil do {pprec ≠ nil}
        begin pp := pp↑.partesuccessiva; dispose(pprec);
            pprec := pp
        end {fine della cancellazione di parti del messaggio};
        if precedente = nil then {elimina il descrittore della prima trasmittente}
            trasmattive := ps↑.stazionesuccessiva
        else precedente↑.stazionesuccessiva := ps↑stazionesuccessiva;
            dispose(ps)
    end {Elimina};
    function UltimoPacchetto (p: pacchetto): Boolean
        {vale true se p contiene un terminatore};
        const terminatore = '.';
        var i: 0..numcaratteri; trovato: Boolean;
            pi: array[1..numcaratteri] of char;
    begin {UltimoPacchetto}
```

```
        i := 0; trovato := false; unpack(p, pi, l);
        repeat {c'è almeno un carattere in p}
            i := i+1; trovato := pi[i] = terminatore
        until (i = numcaratteri) or trovato;
        UltimoPacchetto := trovato
    end {UltimoPacchetto};
begin {ElaboraMessaggi}
    reset(m);
    while not eof(m) do
    begin
        IdentificaTrasmittente(m↑.idtrasmittente, esiste, ps, precedente);
        if not esiste then
            InserisceTrasmittente(m↑.idtrasmittente, precedente, ps);
        InseriscePacchetto(m↑.informazione, ps)
        if UltimoPacchetto(m↑.informazione) then
        begin StampaMessaggio(ps); Elimina(precedente, ps) end;
        get(m)
    end
end {ElaboraMessaggio};
```

COMMENTI

1. L'inserimento di un elemento della lista puntata da *p* modifica il valore del precedente elemento nella lista, se l'inserzione non è all'inizio della lista stessa. In *InserisceTrasmittente* il test sul puntatore precedente (l'elemento precedente a quello corrente nell'inserzione) risulta dalle condizioni di uscita di *IdentificaTrasmittente*.
2. In generale, si dovrebbe notare il numero di volte in cui si effettua un confronto con **nil**. La maggior parte delle volte ciò è fatto per evitare di accedere a un elemento inesistente, oppure come in *Elimina*, per evitare di rilasciare una variabile dinamica inesistente.
3. Si potrebbe migliorare l'efficienza di *ElaboraMessaggi* con una gestione di memoria simile a quella dell'Esercizio 13.2 ed evitando il controllo di ogni pacchetto in *UltimoPacchetto*. Trasmettendo, si dovrebbe aggiungere un flag per indicare l'ultimo pacchetto.

14.3 Invio di un messaggio

```
const l = ...;
type pacchetto = array[1..l] of char;
procedure InviaCarattere(ch: char);...;
procedure RiceveCarattere(var ch: char);...;
...
procedure InviaPacchetto(p: pacchetto)
        {invia un pacchetto con un protocollo molto semplice};
    var i: 1..l;
    procedure Trasmette(c: char)
        {invia un carattere e lo ritrasmette se l'eco è sbagliato};
        var eco: char;
```

```
    begin {Trasmette}
        InviaCarattere(c); RiceveCarattere(eco);
        if c ≠ eco then
        begin Trasmette('\') {cancella il carattere};
            Trasmette(c)
        end
    end {Trasmette};
begin {InviaPacchetto}
    for i := 1 to l do Trasmette(p[i])
end {InviaPacchetto};
```

COMMENTI

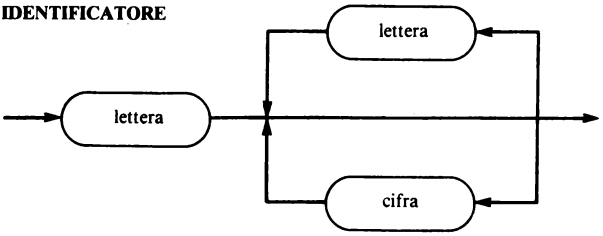
1. La recursione, usata nella procedura *Trasmette*, permette una programmazione elegante e concisa, dove il protocollo della trasmissione è applicato a tutti i caratteri, incluso il carattere di cancellazione.
2. Tuttavia, questo protocollo è rudimentale, poiché non tratta gli errori dovuti all'eco. Se l'errore di trasmissione ha a che fare con la trasmissione, la trasmittente rinvia un carattere ricevuto correttamente. Questo semplice caso porta ad un aumento nel numero dei caratteri trasmessi, ma ci può essere un errore che è impossibile identificare se il carattere di cancellazione inviato dal trasmettitore è trasformato in un carattere normale quando arriva al ricevitore.

B

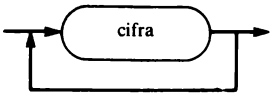
Raccolta di diagrammi sintattici

I seguenti diagrammi descrivono, in modo molto conciso, la sintassi completa del Pascal. Come già spiegato nel paragrafo 1.1, questi diagrammi non costituiscono una pura raccolta di quelli presentati nei vari capitoli del testo, dal momento che il loro scopo è diverso. Essi non devono infatti servire per spiegare la sintassi di una nozione che si sta descrivendo, ma per consentire a chiunque voglia rinfrescare le proprie conoscenze della sintassi del Pascal di avere un comodo riferimento. Le varie istruzioni del linguaggio, per esempio, sono descritte su più diagrammi sparsi nei corrispondenti capitoli, mentre in questa appendice esse sono raccolte in un unico diagramma.
In questa appendice si fa uso delle stesse convenzioni usate nel resto del libro. Un nonterminale chiamato «identificatore di qualcosa» non viene definito in alcun diagramma: semplicemente esso rappresenta un identificatore, con il vincolo aggiuntivo di dover essere definito, da qualche parte nel programma, essere «qualcosa».

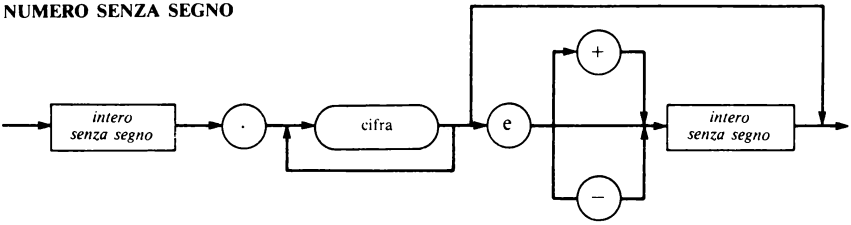
IDENTIFICATORE



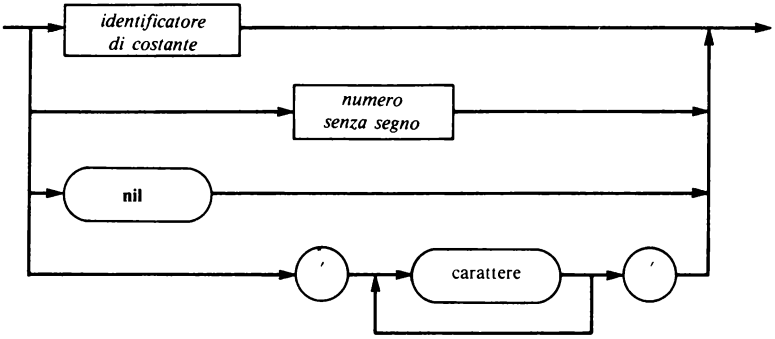
INTERO SENZA SEGNO



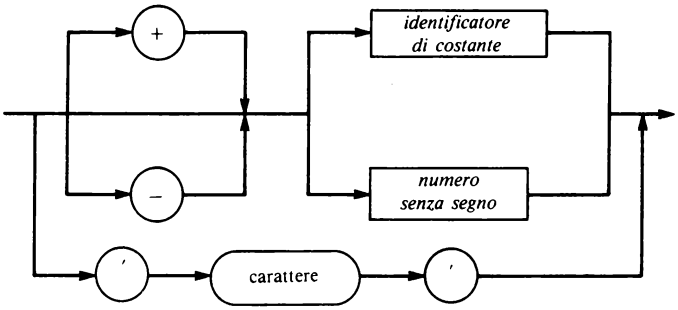
NUMERO SENZA SEGNO



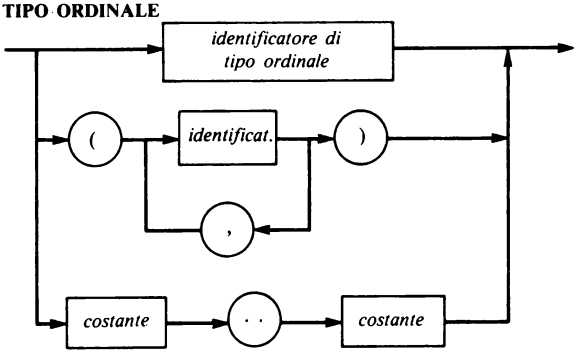
COSTANTE SENZA SEGNO



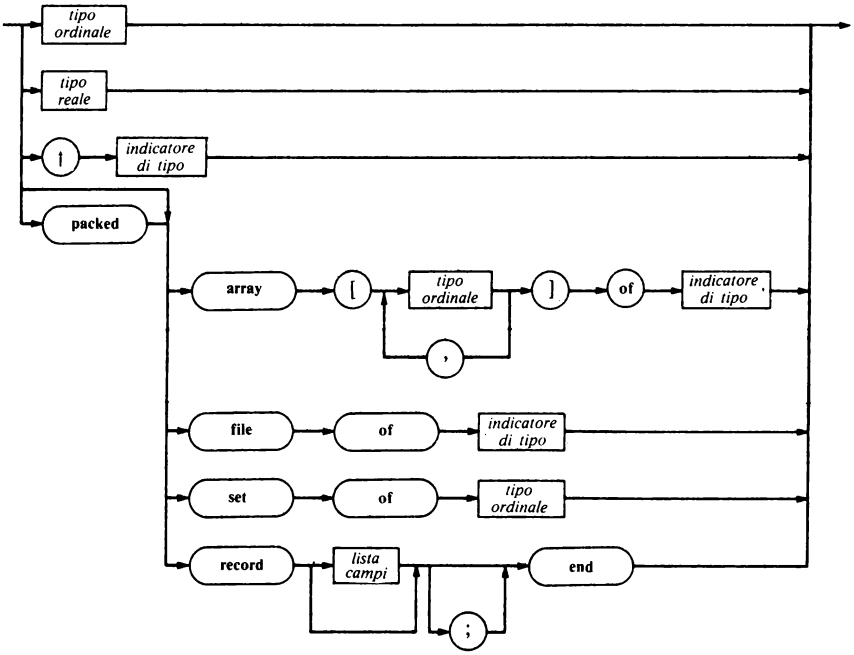
COSTANTE



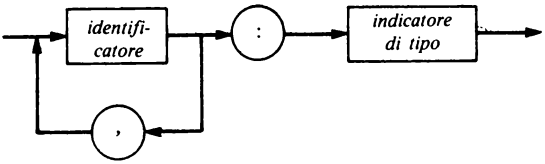
TIPO ORDINALE



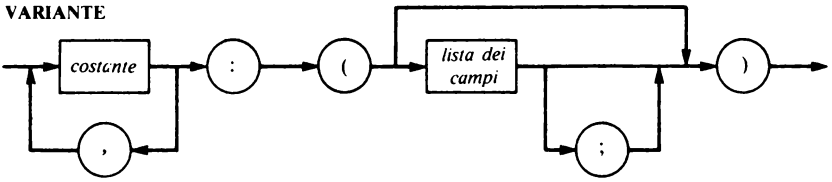
INDICATORE DI TIPO



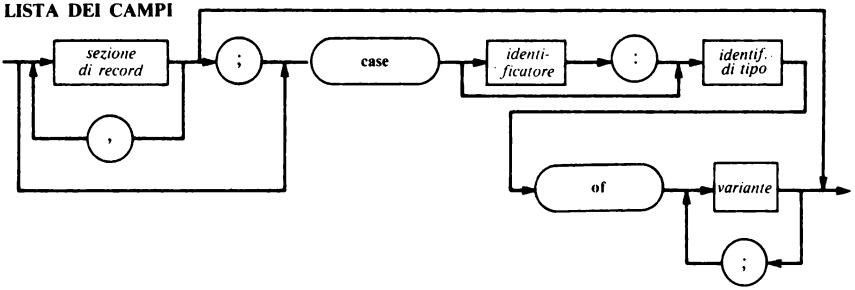
SEZIONE DI RECORD



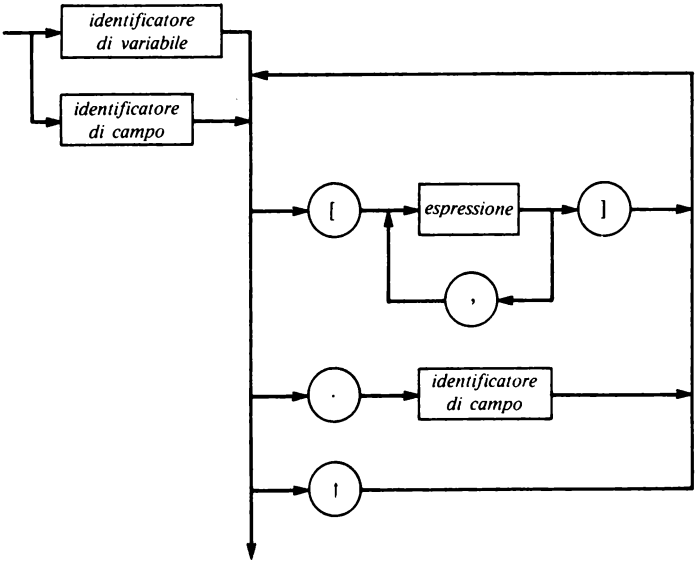
VARIANTE



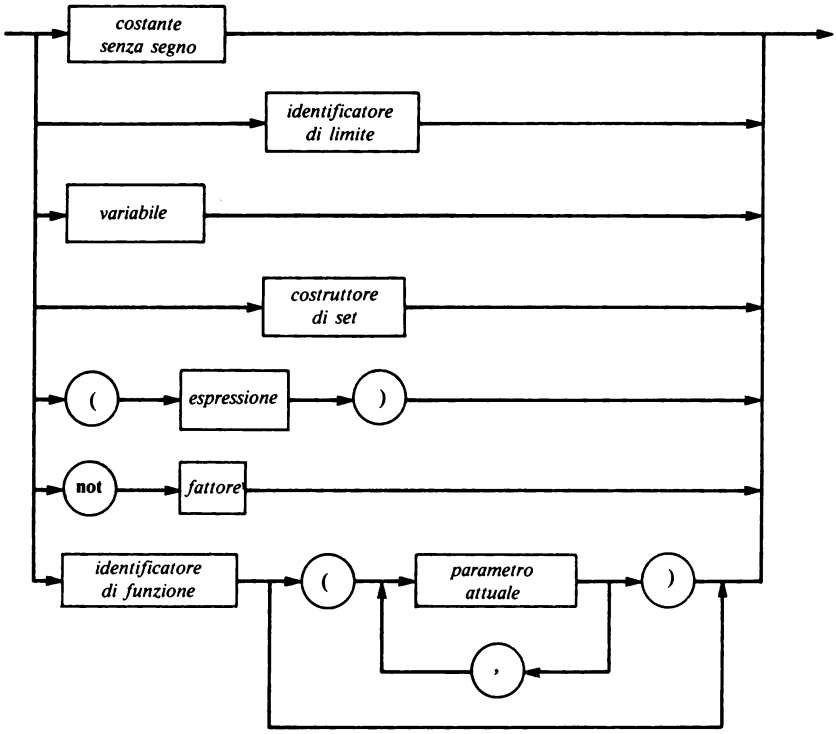
LISTA DEI CAMPI



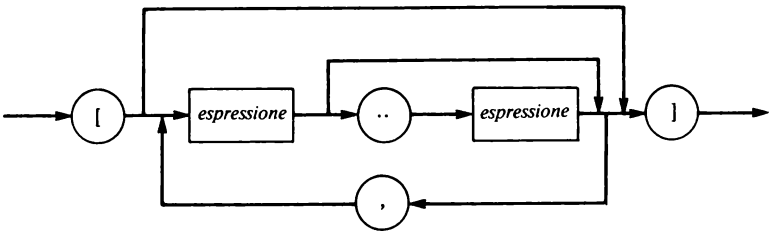
VARIABILE



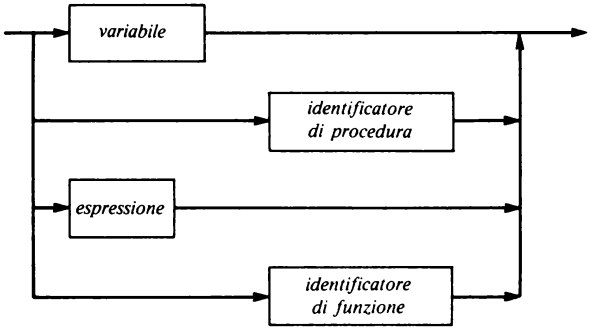
FATTORE



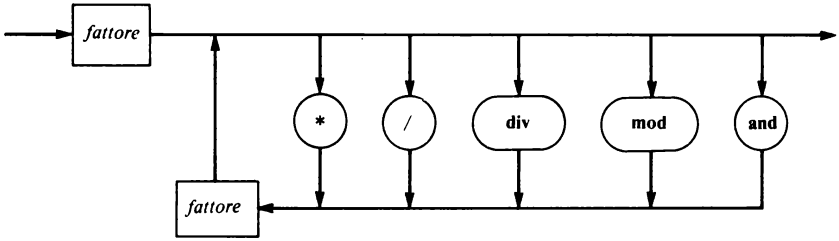
COSTRUTTORE DI SET



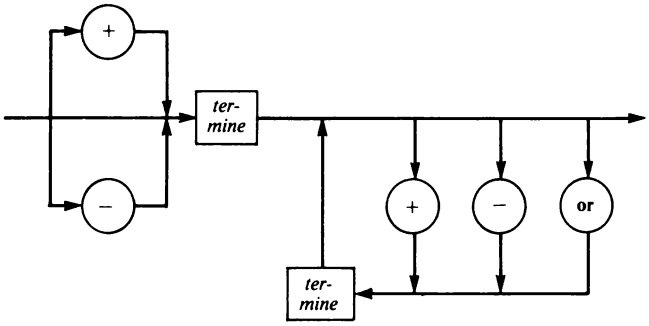
PARAMETRO ATTUALE



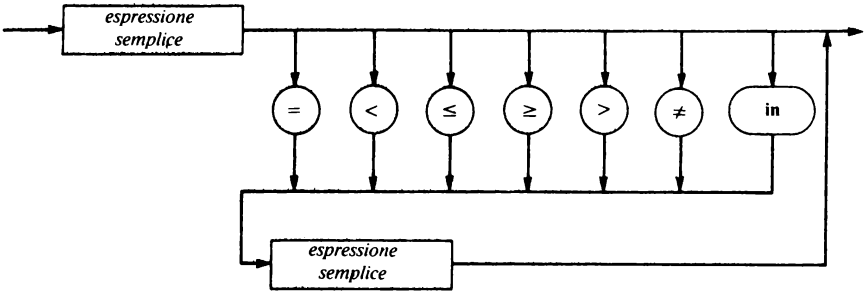
TERMINE



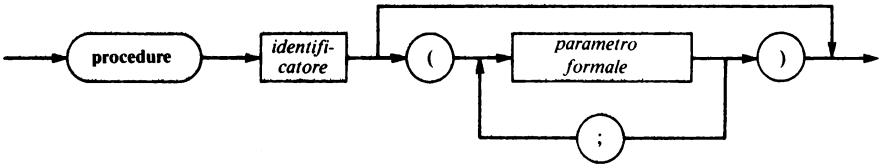
ESPRESSIONE SEMPLICE



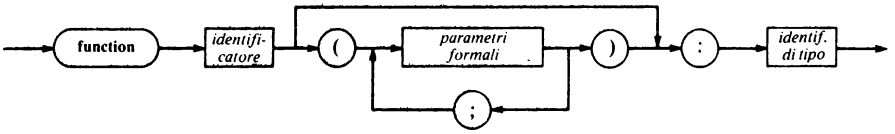
ESPRESSIONE



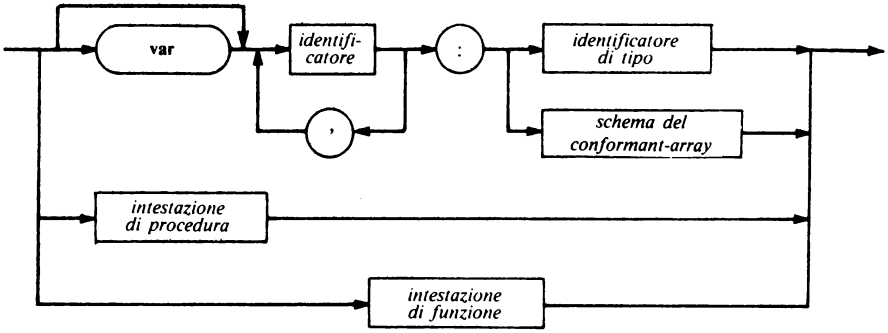
INTESTAZIONE DI PROCEDURA



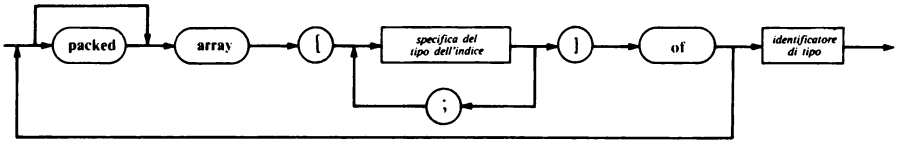
INTESTAZIONE DI FUNZIONE



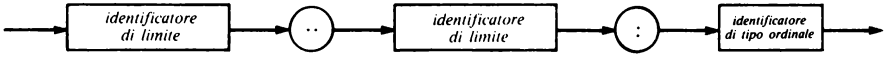
PARAMETRO FORMALE



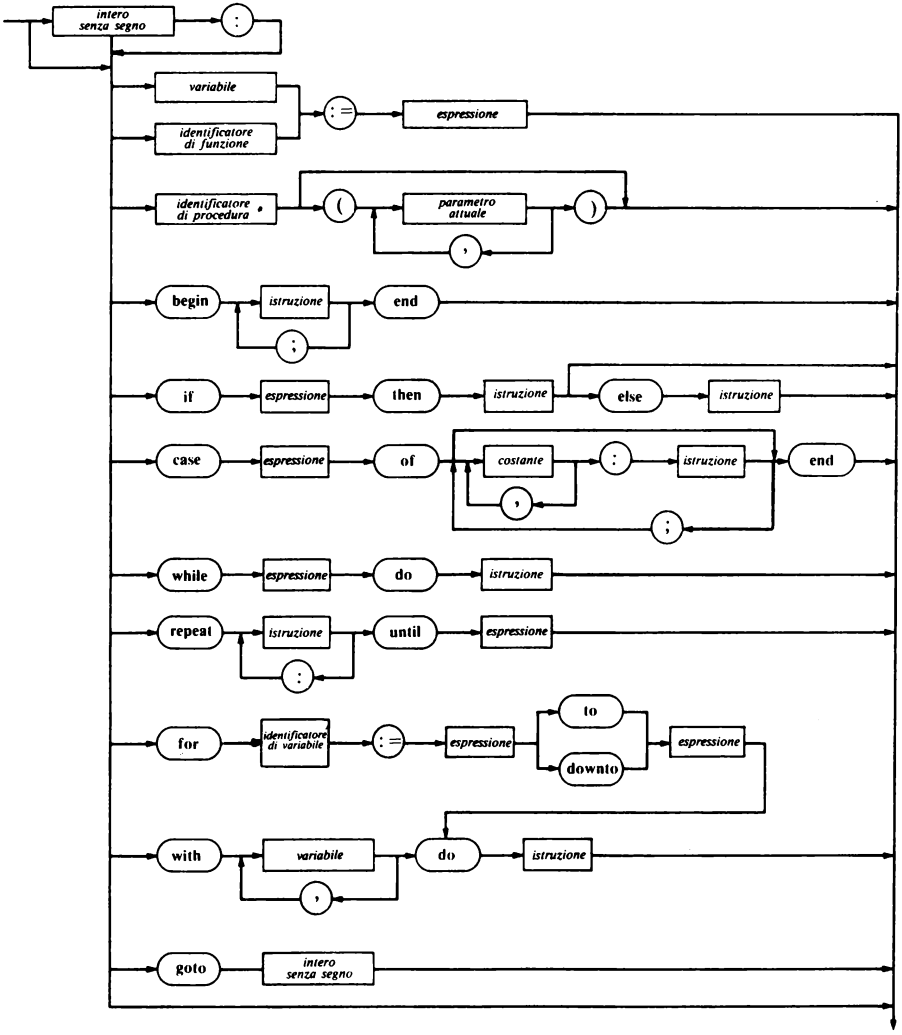
SCHEMA DI UN CONFORMANT-ARRAY



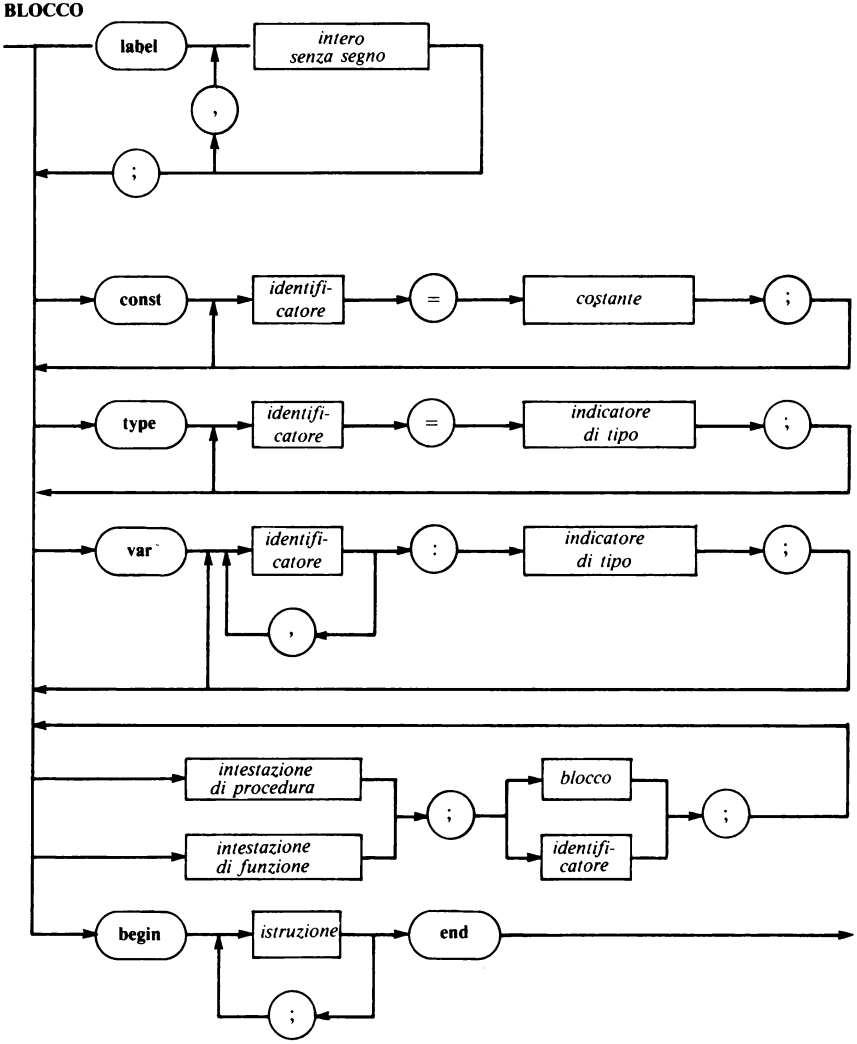
SPECIFICA DEL TIPO DELL'INDICE



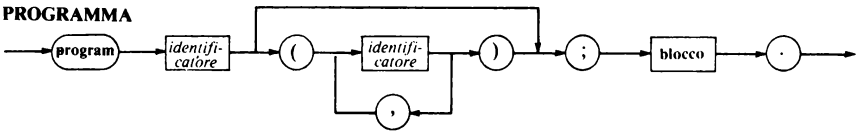
ISTRUZIONE


intero senza segno
:
variabile
identificatore di funzione
:=
espressione
identificatore di procedura
(
parametro attuale
,
)
begin
istruzione
;
end
if
espressione
then
istruzione
else
istruzione
case
espressione
of
costante
,
:
istruzione
;
end
while
espressione
do
istruzione
repeat
istruzione
:
until
espressione
for
identificatore di variabile
:=
espressione
to
downto
espressione
with
variabile
,
do
istruzione
goto
intero senza segno

BLOCCO



PROGRAMMA



C

Vocabolario del Pascal

Il vocabolario completo del Pascal è descritto nel paragrafo 1.2 ed è riportato in questa appendice solo per comodità di riferimento. Sono state usate due differenti classificazioni: nella prima, gli elementi del vocabolario sono ordinati secondo la funzione svolta, mentre nella seconda essi sono classificati in base al loro «aspetto». Dal momento che alcuni simboli hanno più significati, essi compaiono più volte nella prima classificazione. Inoltre, poiché esistono più rappresentazioni per alcuni simboli, la seconda classificazione presenta non solo la rappresentazione usata in questo volume, ma anche le possibili alternative, ed è perciò ridondante.

Simboli base del Pascal

Rappresentazione Standard	Rappresentazioni alternative	Classe
a...z	qualsiasi altro carattere, corpo e stile	lettere
01234 56789		cifre
+ − * / **div** **mod**		operatori aritmetici
or **and** **not**	∨ ∧ ¬	operatori booleani

Simboli base del Pascal (segue)

Rappresentazione Standard	Rappresentazioni alternative	Classe
< <= = <> >= > **in**	≤ ≠ ≥	operatori relazionali

(
)
[
]
{
}
begin
end
	(.
	.)
	(*
	*)
		parentesi

. , : ; ..		separatori
if the else **case of end** **while do repeat until** **for to downto** **with program**		separatori di istruzioni
const type var **procedure function**		identificatori di classi di oggetti
array file of **record set end** **packed**		identificatori di classi di strutture
:= ' ↑ **nil** **goto** **label**	^ @	vari

Simboli base del Pascal (segue)

PAROLE CHIAVE

and	**downto**	**if**	**or**	**then**
array	**else**	**in**	**packed**	**to**
begin	**end**	**label**	**procedure**	**type**
case	**file**	**mod**	**program**	**until**
const	**for**	**nil**	**record**	**var**
div	**function**	**not**	**repeat**	**while**
do	**goto**	**of**	**set**	**with**

SIMBOLI SPECIALI

+	<	≠	>	.)	;	..	¦	^
−	< =	< >	[	.	↑	: =	(*	v
*	≤	≥	(.	,	~	(	¦	¬
/	=	> =	]	:	@	)	*)	

D

Identificatori predefiniti

Dal momento che questi identificatori esistono come se fossero dichiarati o definiti in un ambiente esterno al programma, vengono qui elencati in modo informale, come una successione di dichiarazioni e definizioni.

```
const
   maxint = {il più grande intero positivo};
   {false, true sono definite con il tipo Boolean}

type
   Boolean = (false, true);
   integer = -maxint.. +maxint;
   char = {l'insieme di caratteri descritto nell'Appendice C};
   real = {l'insieme possibile di numeri reali};
   text = file of char {con proprietà aggiuntive};
var
   input, output: text {dichiarati solo se compaiono come parametri del
      programma};

{funzioni aritmetiche}
function abs (x: {integer o real}): {tipo di x}
   {valore assoluto};
function sqr (x: {integer o real}): {tipo di x}
   {quadrato di x};
function sin (x: real): real
   {seno di x espresso in radianti};
function cos (x: real): real
   {coseno di x espresso in radianti};
function exp (x: real): real
   {esponenziale di x};
```

```
function ln (x: real): real
   {logaritmo naturale di x > 0};
function sqrt (x: real): real
   {radice quadrata di x ≥ 0};
function arctan (x: real): real
   {valore principale, in radianti, dell'arcotangente di x};

{funzioni di conversione}
function trunc (x: real): integer
   {0 ≤ x−trunc(x) < 1 se x ≥ 0, oppure
    −1 < x−trunc(x) ≤ 0 se x < 0};
function round (x: real): integer
   {trunc(x+0.5) se x ≥ 0, oppure
   trunc(x−0.5) se x < 0};

{funzioni ordinali}
function ord (x: {qualsiasi tipo ordinale}): integer
   {numero ordinale di x};
function chr (x: integer): char
   {carattere il cui numero ordinale è x, se esiste};
function succ (x: {qualsiasi tipo ordinale}): {tipo di x}
   {successore di x, se esiste};
function pred (x: {qualsiasi tipo ordinale}): {tipo di x}
   {predecessore di x, se esiste};

{funzioni booleane}
function odd (x: integer): Boolean
   {x è dispari};
function eof (var f: {qualsiasi tipo file}): Boolean
   {la parte destra di f è vuota};
function eoln (var f: text): Boolean
   {se ¬eof(f), allora firstof parte destra di f è un simbolo di fine linea};

{procedure di gestione file}
procedure rewrite (var f: {qualsiasi tipo file})
   {inizializza f in generazione};
procedure reset (var f: {qualsiasi tipo file})
   {inizializza f in ispezione};
procedure put (var f: {qualsiasi tipo file})
   {passa all'elemento successivo in generazione};
procedure get (var f: {qualsiasi tipo file})
   {passa all'elemento successivo in ispezione};
procedure write (var f: {qualsiasi tipo file};
          x1,..., xn: {tipo degli elementi di f, o altri tipi se f è un textfile})
   {scrive in f i valori di x1,..., xn};
```

```
procedure read (var f: {qualsiasi tipo file};
        var x1,..., xn: {tipo degli elementi di f, o altri tipi se f è un textfile})
   {legge i valori di x1,..., xn da f};
procedure writeln (var f: text;
        x1,..., xn: {integer, real, char, Boolean o text})
   {scrive in f e passa a una nuova linea};
procedure readln (var f: text;
        var x1,..., xn: {integer, real o char})
   {legge da f e passa alla linea successiva};
procedure page (var f: text)
   {inizia una nuova pagina in f};

{procedure di allocazione dinamica}
procedure new (var p: {qualsiasi tipo puntatore};
        c1,..., cn: {tipi dei rispettivi campi varianti})
   {alloca una variabile dinamica, puntata da p};
procedure dispose (p: {qualsiasi tipo puntatore};
        c1,..., cn: {tipi dei rispettivi campi varianti})
   {dealloca la variabile dinamica puntata da p};

{procedure di conversione}
procedure pack (a: {un tipo array non compattato};
        i: {tipo dell'indice del tipo di a})
        var z: {tipo array compattato i cui elementi sono dello stesso tipo di a})
   {compatta a in z, a partire da a[i]};
procedure unpack (z: {tipo array compattato};
        var a: {tipo array non compattato i cui elementi sono dello stesso tipo
           di z};
        i: {tipo dell'indice del tipo di a})
   {scompatta z in a, a partire da a[i]};
```

E

Aspetti implementativi

Numerose caratteristiche del linguaggio non si possono definire senza fare riferimento ad un'implementazione particolare.
I seguenti argomenti devono venir **definiti in ogni implementazione**:

Tipi predefiniti

1. Il valore della costante predefinita *maxint* (paragrafo 2.4.1).
2. I valori di tipo *real*, il loro campo, la loro precisione, la precisione dei calcoli con valori reali (paragrafi 2.6 e 3.1).
3. I valori di tipo *char* e gli ordinali a loro associati (paragrafo 2.4.3).

File

4. Il collegamento fra parametri di programma di tipo file con l'ambiente esterno (paragrafo 7.4).
5. Quando viene eseguita realmente l'azione fisica implicata da una chiamata alla procedura predichiarata *get* (paragrafo 8.2).
6. I valori impliciti usati per dimensionare i campi di numeri interi, reali o booleani su file di tipo *text* (paragrafo 8.4).
7. Il numero di cifre per l'esponente (e tipo) della lettera *e* usato per rappresentare l'esponente, quando si debbano scrivere numeri reali sui file di tipo *text* (paragrafo 8.4.3).
8. Il tipo della stringa di caratteri quando si devono scrivere valori booleani sui file di tipo *text* (paragrafo 8.4.4).
9. L'effetto sul file di tipo *text* della procedura predichiarata *page* (paragrafo 8.4).
10. L'effetto della chiamata *reset*(*input*) e *rewrite*(*output*) (paragrafo 8.2).

I seguenti punti sono, invece, **dipendenti dall'implementazione**:

Interfaccia con il sistema ospite

1. Direttive diverse da *forward* (paragrafo 4.1.1).
2. La natura e il collegamento con l'ambiente esterno dei parametri del programma che non sono file (paragrafo 1.3).

Calcoli per valore o per indirizzo

3. L'ordine di valutazione e/o l'accesso alle variabili in entrambe le parti dell'istruzione di assegnamento (paragrafo 1.6).
4. L'ordine di valutazione e l'esistenza di questa valutazione, per gli operandi di operatori diadici (paragrafo 3.1).
5. L'ordine di valutazione, l'accesso e la corrispondenza dei parametri attuali in una chiamata di procedura o di funzione (paragrafo 3.3).

Altri argomenti

6. Che cosa accade quando un file di tipo *text*, al quale è stata applicata la procedura predichiarata *page*, è riletto (paragrafo 8.4).
7. L'esistenza di altre rappresentazioni alternative dei simboli del Pascal (paragrafo 1.2 e Appendice C).

Per tutti i punti qui discussi è assolutamente necessario consultare il manuale utente — o una documentazione equivalente — dell'implementazione Pascal usata.

Bibliografia ragionata

Una bibliografia completa sul Pascal includerebbe, probabilmente, parecchie centinaia di articoli e alcune dozzine di libri. Inoltre, sarebbe obsoleta ancor prima di essere completata. Abbiamo scelto qui solo una piccola serie di testi importanti, suddivisi in tre argomenti principali.

Definizioni del linguaggio

Hoare C.A.R. e Wirth N.: *An Axiomatic Definition of the Programming Language Pascal* in *Acta Informatica*, vol. 2, n. 4, 1973, pp. 335-355.
Questo documento, molto conciso, contiene la semantica quasi completa in assiomi e la sintassi completa in diagrammi della versione revisionata del Pascal. Anch'essa è stata ristampata in Wassermann (1980).

ISO (International Organization for Standardization): *Specification for the Computer Programming Language Pascal*, 7185-1983
Questo è lo stato corrente dello Standard internazionale. È un documento di ottanta pagine, piuttosto difficile da leggere e capire, ma estremamente preciso e completo. Gli standard nazionali nei paesi membri della ISO dovrebbero essere le copie esatte di questo standard internazionale.

Jensen K. e Wirth N.: *Pascal: User Manual and Report*, Springer Verlag, Berlin, 1974.
Questo piccolo best-seller (167 pagine) contiene il rapporto revisionato del Pascal di Niklaus Wirth e il manuale per l'utente di Jensen e Wirth. Per molto tempo è stato il solo testo di riferimento ufficiale sul Pascal Standard. Non è pienamente soddisfacente a causa delle numerose ambiguità, contraddizioni e mancanze. Nelle successive ristampe sono state fatte delle leggere revisioni.

Wirth N.: *The Programming Language Pascal* in *Acta Informatica*, vol. 1, n. 1, 1971, pp. 35-63.
Questa è la definizione originale del linguaggio, ristampata più tardi in Wassermann (1980). È interessante leggerla per notare le differenze fra questa versione originale e quella revisionata. Il paragrafo introduttivo è di particolare interesse.

Valutazioni del linguaggio e critiche

I quattro lavori seguenti sono stati ristampati in Wassermann (1980).

Habermann A.N.: *Critical Comments on the Programming Language Pascal*, in *Acta Informatica*, vol. 3, n. 1, 1973, pp. 47-57.
La critica principale di Habermann tratta le ambiguità, le incoerenze e le lacune nella definizione dei testi e nella nozione di tipo. Il rapporto contiene anche critiche che possono scaturire soltanto da una cattiva lettura dei testi, o da differenze di opinioni su ciò che dovrebbe e ciò che non dovrebbe essere incluso in tale linguaggio.

Lecarme O. e Desjardins P.: *More Comments on the Programming Language Pascal*, in *Acta Informatica*, vol. 4, n. 3, 1975, pp. 231-243.
Questo lavoro è una risposta a quello di Habermann. Corregge alcuni errori di quel rapporto e suggerisce alcuni miglioramenti al linguaggio.

Welsh J., Sneeringer W.J. e Hoare C.A.R.: *Ambiguities and Insecurities in Pascal*, in *Software — Practice and Experience*, vol. 7, n. 6, 1977, pp. 685-696.
Questo importante lavoro è stato anche ristampato in Barron (1981). L'attento esame di alcuni aspetti difficili del Pascal è stato molto utile durante la preparazione dello Standard ISO, che fornisce soluzioni alla maggior parte dei problemi evidenziati.

Wirth N.: *An Assessment of the Programming Language Pascal*, in *IEEE Trans. on Software Engineering*, giugno 1975, pp. 192-198.
L'autore del Pascal esamina il suo lavoro sei anni dopo e discute alcune scelte, apparentemente contraddittorie, dimostrando che la maggior parte di esse sono il risultato di un attento bilancio fra semplicità e potenza.

Libri di testo

Alagič S. e Arbib M.A.: *The Design of Well-Structured and Correct Programs*, Springer Verlag, Berlin, 1978.
Questo libro di testo, piuttosto orientato verso la teoria, usa il Pascal come strumento per dimostrare la costruzione di programmi parallelamente alla loro prova formale, usando l'assiomatizzazione di Hoare.

Barron D.W. (a cura di): *Pascal — The Language and Its Implementation*, Wiley, New York, 1981.
È un'altra collezione di lavori, che trattano però solo del Pascal e la maggior parte non era mai stata pubblicata in precedenza. In particolare, questa raccolta contiene dei rapporti sul Pascal-P, l'implementazione portabile che è stata usata nella maggior parte delle implementazioni del Pascal (almeno all'inizio) e sul Pascal-S, un sottoinsieme del Pascal progettato (da Wirth) espressamente per un'implementazione estremamente compatta, descritta nello stesso Pascal.

Dahl O.J., Dijkstra E.W. e Hoare C.A.R.: *Structured Programming*, Academic Press, London, 1972
Questo libro è formato da tre lavori, che costituiscono la pietra miliare della programmazione strutturata. Il secondo, *Notes on Data Structuring* di Hoare, contiene la maggior parte delle idee sui tipi di dati che sono stati implementati nel Pascal.

Wassermann A.I. (a cura di): *Tutorial: Programming Language Design*, IEEE Computer Society Press, New York, 1980.
Questa è una raccolta di lavori pubblicati precedentemente (tranne alcune eccezioni) che contiene, fra gli altri, sei dei più importanti rapporti sul Pascal; tra questi sono assai degni di nota quelli di Hoare e Wirth sulla progettazione dei linguaggi di programmazione.

Wirth N.: *Systematic Programming: an Introduction*, Prentice-Hall, Englewood Cliffs, 1973.
Questo minuscolo libro di introduzione alla programmazione usa il Pascal come strumento di sostegno, senza mai menzionarlo, e copre circa i due terzi del linguaggio.

Wirth N.: *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, 1976.
È la logica continuazione del libro precedente ed è utile ad un secondo corso sulla programmazione; anch'esso presenta il Pascal come uno strumento di sostegno. È un'inestimabile fonte di programmi Pascal di difficoltà medio-alta.

Indice analitico

Nella stessa serie:

88 7700 601 3	S. Harrington, *Computer Graphics - Corso di programmazione*
88 7700 603 X	M. McGilton e R. Morgan, *Il sistema operativo UNIX*
88 7700 604 8	E. Rich, *Intelligenza Artificiale*

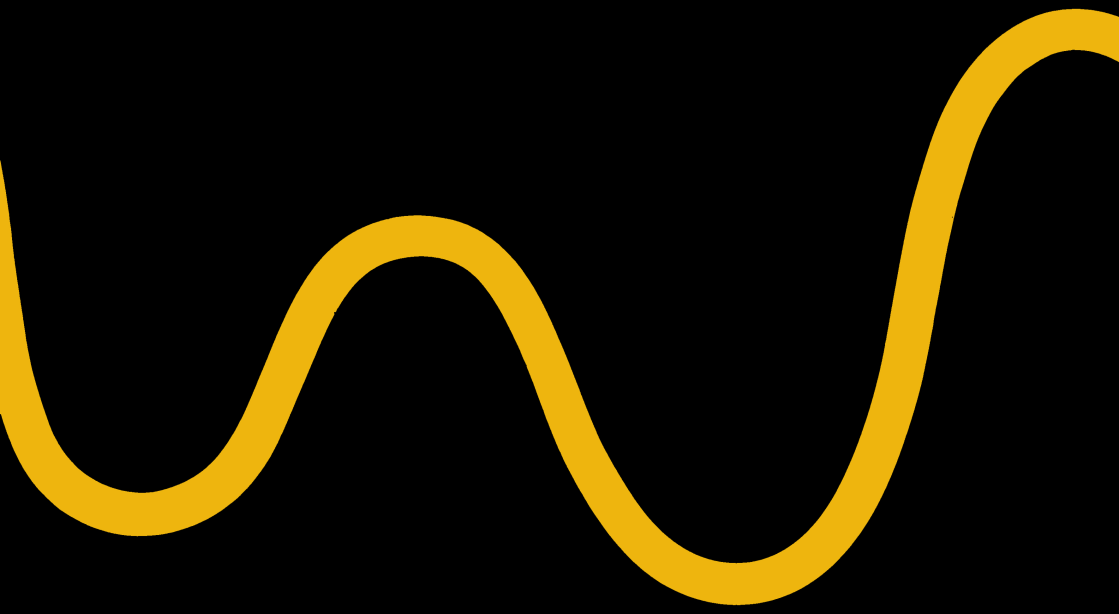
Altri titoli pubblicati in lingua italiana:

88 7700 001 5	J. Heilborn e R. Talbott, *Guida al Commodore 64*
88 7700 002 3	C.A. Street, *La gestione delle informazioni con lo ZX Spectrum*
88 7700 003 1	T. Woods, *L'Assembler per lo ZX Spectrum*
88 7700 004 X	R. Jeffries, G. Fisher e B. Sawyer, *Divertirsi giocando con il Commodore 64*
88 7700 005 8	G. Bishop, *Progetti hardware con lo ZX Spectrum*
88 7700 006 6	H. Mullish e T. Kruger, *Il BASIC Applesoft*
88 7700 007 4	N. Williams, *Progettazione di giochi d'avventura con lo ZX Spectrum*
88 7700 008 2	H. Peckham, *Il BASIC e il PC-IBM in pratica*
88 7700 009 0	H. Peckham, *Il BASIC e il Commodore 64 in pratica*
88 7700 010 4	S. Nicholls, *Tecniche avanzate in Assembler con lo ZX Spectrum*
88 7700 011 2	K. Skier, *L'Assembler per il Commodore 64 e il VIC-20*
88 7700 012 0	S. Kamins e M. Waite, *Programmazione umanizzata in Applesoft*
88 7700 013 9	A. Pennell, *Guida allo ZX Microdrive e all'Interface 1*
88 7700 015 5	P. Cohen, *Grafica e animazione con gli Apple II*
88 7700 016 3	C. Duff, *Guida al Macintosh*
88 7700 017 1	G. Kane, *Il manuale MC68000*
88 7700 018 X	P. Hoffman e T. Nicoloff, *Il manuale MS-DOS*
88 7700 020 1	S. Nicholls, *Grafica avanzata con lo ZX Spectrum*
88 7700 021 X	L.J. Graham e T. Field, *Guida al PC-IBM*
88 7700 022 8	T. Field, *Come usare MacWrite e MacPaint*
88 7700 024 4	H. Peckham, *Il BASIC e gli Apple II in pratica*
88 7700 025 2	C. Morgan e M. Waite, *Il manuale 8086/8088*
88 7700 027 9	G. Mainis, *Il manuale ProDOS*
88 7700 028 7	J. Jones, *Il SuperBASIC del QL*
88 7700 029 5	C. Opie, *L'Assembler per il QL*
88 7700 030 9	W. Ettlin e G. Solberg, *Il BASIC Microsoft*
88 7700 032 5	R. Person, *Le meraviglie dell'animazione con gli Apple II*
88 7700 034 1	P. Hoffman, *Il manuale MSX*
88 7700 035X	R. Person, *Le meraviglie dell'animazione con il PC-IBM*
88 7700 036 8	W. Ettlin, *Il Multiplan per il Macintosh*
88 7700 037 6	D. Kruglinski, *Introduzione al Framework*

Olivier Lecarme
Jean-Louis Nebut

Pascal

Guida per programmatori



Lire 29 000
(IVA 2% inclusa)

ISBN 88 7700 602 1

LECARME NEBUT *PASCAL*