

Niklaus Wirth
PRINCIPI DI
PROGRAMMAZIONE
STRUTTURATA

0100

INV. N°

531-2

9006026

ISEDI

Indice



ISEDI

© 1995 UTET Libreria
Via P. Giuria 20 - 10125 Torino

Titolo originale: *Systematisches Programmieren*
© 1972 B.G. Teubner Verlag, Stuttgart
© 1977 Isedi, Istituto Editoriale Internazionale, Milano
© 1980 Arnoldo Mondadori, Milano
© 1984 Petrini editore, Torino

Traduzione di Mario Ornaghi

I diritti di traduzione, di memorizzazione elettronica, di riproduzione e di adattamento totale o parziale, con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche) sono riservati per tutti i Paesi.

L'Editore potrà concedere a pagamento l'autorizzazione a riprodurre una porzione non superiore a un decimo del presente volume e fino a un massimo di settantacinque pagine.

Le richieste di riproduzione vanno inoltrate all'AIDROS,
via delle Erbe 2 - 20121 Milano.

Tel. e fax 02/809506

Stampa: Edit.ei, Moncalieri (To)

ISBN 88-8008-112-8

Volume di 194 pagine

Ristampe: 2 3 4 5 6 7

1997 1998 1999 2000

Prefazione

1. Introduzione	5
2. Concetti fondamentali	6
3. Cenni sulle parti principali di un calcolatore	12
4. Ausilii alla programmazione e sistemi di programmazione	17
5. Semplici esempi di programmi	20
5.1. Problemi	31
6. Terminazione dei programmi	32
6.1. Problemi	34
7. Notazione lineare e linguaggi di programmazione	35
7.1. Generalità	35
7.2. Espressioni e istruzioni	39
7.2.1. Istruzioni strutturate	40
7.2.2. Regole di verifica in notazione lineare	43
7.3. Semplici esempi di programmi in notazione lineare	44
7.4. Problemi	48
8. Tipi di dati	51
8.1. Il tipo Boolean	54
8.2. Il tipo Integer	55
8.3. Il tipo Char	56
8.4. Il tipo Real	59
8.5. Problemi	64

9.	<i>Programmi basati su relazioni di ricorrenza</i>	66	<i>Appendice 1. Il linguaggio di programmazione PASCAL</i>	170
9.1.	Successioni	66		
9.2.	Serie	70	<i>Appendice 2. L'alfabeto ASCII</i>	179
9.3.	Problemi	74		
10.	<i>Il tipo strutturato "file"</i>	76	<i>Bibliografia</i>	181
10.1.	Il concetto di file	76		
10.2.	Generazione di un file	78	<i>Indice analitico</i>	182
10.3.	Ispezione di un file	79		
10.4.	File costituiti da un testo	82		
10.5.	Problemi	86		
11.	<i>Il tipo strutturato "array"</i>	87		
11.1.	Regole di verifica per l'istruzione for	92		
11.2.	Problemi	96		
12.	<i>Sottoprogrammi - Procedure e funzioni</i>	99		
12.1.	Concetti e terminologia	99		
12.2.	Il concetto di locale	101		
12.3.	Parametri di procedura	103		
12.4.	Funzioni e procedure parametriche	106		
12.5.	Problemi	109		
13.	<i>Trasformazioni della rappresentazione dei numeri</i>	112		
13.1.	Ingresso (lettura) di numeri interi positivi nella rappresentazione decimale	113		
13.2.	Uscita (stampa) di interi positivi nella rappresentazione decimale	114		
13.3.	Uscita (stampa) di numeri razionali fratti nella rappresentazione decimale	114		
13.4.	Trasformazione delle rappresentazioni in virgola mobile	115		
13.5.	Problemi	118		
14.	<i>Elaborazione di testi mediante le strutture "file" e "array"</i>	120		
14.1.	Delimitazione della lunghezza delle linee di un <i>textfile</i>	120		
14.2.	Modifica (editing) di una linea in un testo	123		
14.3.	Riconoscimento di linguaggi regolari	127		
14.4.	Problemi	133		
15.	<i>Sviluppo di un programma per passi successivi</i>	138		
15.1.	Soluzione di un sistema di equazioni lineari	140		
15.2.	Determinazione del più piccolo numero eguale a due somme diverse di numeri naturali elevati alla terza potenza	146		
15.3.	Calcolo dei primi n numeri primi	151		
15.4.	Un esempio di algoritmo euristico	156		
15.5.	Problemi	165		

Sulla programmazione esistono numerosissimi libri, relativi a linguaggi per calcolatori di ogni tipo, e scrivere un testo che tenga conto di quanto già esiste è un'impresa. È perciò opportuno spiegare i motivi che hanno condotto alla pubblicazione di questo libro.

In primo luogo, intendo trattare la programmazione come una disciplina autonoma, riguardante i metodi di formulazione e di costruzione degli algoritmi. Un algoritmo è una legge che governa un'intera classe di processi di elaborazione e controllo di dati; quindi dev'essere costruito a partire da unità concepite logicamente, adeguate e sicure.

Per dare alla programmazione la struttura di un metodo rigoroso è necessario individuare i problemi e le tecniche tipiche della programmazione, cioè valide al di là delle particolari applicazioni. Per questo motivo, gli esercizi e gli esempi sono stati scelti per illustrare problemi e metodi generali, e non per un loro interesse specifico. Gli stessi criteri di generalità hanno guidato la scelta del linguaggio di programmazione, inteso semplicemente come un mezzo e non come il fine. Infatti, il fine di un corso di programmazione non è la conoscenza dettagliata di un particolare linguaggio; il linguaggio o il formalismo usato deve invece rispecchiare in modo chiaro, comprensibile e naturale i concetti tipici degli algoritmi, tenendo conto delle proprietà e dei limiti dei calcolatori digitali.

Ho cercato inoltre di trattare le principali idee e le principali tecniche di verifica dei programmi. Un programma non descrive un unico processo di calcolo: esso governa un'intera classe di processi; se si vuol dimostrare rigorosamente che un programma è corretto, bisogna allora verificare se esso possiede le proprietà richieste in tutti i casi possibili. La verifica "empirica" dei programmi, usata

nella pratica, consiste, invece, nel "provare" dei singoli processi, verificando che la loro esecuzione soddisfi alle proprietà richieste, ma nulla dice sulla totalità dei processi descritti dal programma in esame. Per fare delle affermazioni valide sull'intero programma è necessaria una verifica analitica.

L'esposizione dei principali metodi di verifica analitica dei programmi richiede un livello di astrazione superiore a quello di un normale corso di programmazione. Per questo motivo, da più parti mi hanno rivolto delle critiche sull'opportunità di trattare questo tema in un corso introduttivo. Tuttavia, ho la ferma convinzione che gli elementi qui esposti, relativi al concetto di asserzione e di invariante, sono di importanza fondamentale e che vanno trattati nella parte iniziale del corso: essi sono infatti concetti essenziali per comprendere a fondo gli algoritmi e per superare i limiti della sola intuizione.

Questo testo è orientato verso chi considera i metodi di costruzione degli algoritmi e dei programmi come parte integrante della propria formazione matematica; sono perciò trascurate le esigenze di chi vuole imparare immediatamente a codificare qualche problema in programmi per calcolatore.

La notazione formale usata in questo libro si basa sul linguaggio di programmazione ALGOL 60, nato nel 1960. Il motivo per cui non si è impiegato direttamente l'ALGOL 60 è che la programmazione, rispetto al '60, investe oggi un campo di applicazioni assai più ampio, e che un corso introduttivo alla programmazione non deve essere orientato a un'area particolare di applicazioni. L'ALGOL 60 fu infatti sviluppato soprattutto per le applicazioni numeriche, e la formulazione di programmi relativi a problemi di altro tipo richiede un uso non appropriato e poco elegante del linguaggio. Ma l'uso inappropriato degli strumenti di programmazione è certo un esempio di cattiva programmazione, che non deve essere insegnato.

La mia esperienza di insegnante mi ha portato a ricercare una notazione formale che permetta di rappresentare, in modo chiaro e sistematico, sia le strutture dei processi che le strutture dei dati: e questo perché ognuno è in grado di usare con maggior facilità il linguaggio imparato per primo. Non è questione di "economia mentale": la ragione è, invece, che il "primo" linguaggio diventa lo strumento più naturale per formulare concretamente dei concetti. In altre parole, non si impara soltanto un nuovo linguaggio, ma anche un nuovo modo di pensare: la scelta del linguaggio va fatta perciò con particolare cura.

Per la comprensione del testo sono richieste alcune conoscenze di matematica elementare, impartite nella scuola superiore. In particolare, è bene conoscere le basi del calcolo logico proposizionale e il concetto di induzione matematica; sono utili anche alcuni concetti

del calcolo delle probabilità, mentre il calcolo infinitesimale non è necessario, se non per alcuni problemi che possono essere saltati.

Gli esercizi hanno una grande importanza: per imparare a programmare, bisogna affrontare e risolvere i problemi con impegno individuale. Infatti, la programmazione è un'attività di costruzione e di sintesi, nella quale si impara molto dall'esperienza e dai propri errori. Preferisco problemi semplici ed esposti con chiarezza; lo scopo e il risultato devono essere spiegati senza ricorrere a un complicato formalismo. Gli esercizi servono soprattutto per abituare lo studente a usare i concetti e le tecniche illustrate, non certo per metterlo di fronte a complicati enigmi, la cui soluzione richieda molto tempo e particolare esperienza. I problemi riportati alla fine di ogni capitolo possono servire da schema: essi possono essere modificati ed estesi nei modi più diversi.

I risultati di un corso di programmazione dipendono in larga misura dal centro di calcolo: se chi frequenta non vi può accedere liberamente e se il centro non garantisce un servizio adeguato, il corso finisce per deludere e scoraggiare gli studenti. Per prima cosa, il centro deve accettare ed eseguire immediatamente i piccoli lavori: un programma che occupa il calcolatore al più per pochi secondi, e che dà come risultato uno stampato di poche dozzine di righe, deve essere elaborato e restituito al massimo in un quarto d'ora. Inoltre, è molto importante che il sistema di compilazione impiegato dia delle risposte chiare e comprensibili in tutte le circostanze; infatti, in particolar modo per i principianti, raramente lo stampato di uscita contiene i risultati desiderati: per lo più, contiene dei "messaggi" che indicano gli errori individuati. Il sistema impiegato deve esprimere tali messaggi nel linguaggio umano, oppure nel linguaggio di programmazione usato. In nessun caso devono comparire messaggi del sistema operativo oscuri e non motivati, o addirittura il contenuto di certe celle di memoria stampato in ottale o in esadecimale. Le frasi non appartenenti al linguaggio di programmazione ma, per esempio, proprie del sistema operativo, devono essere ridotte al minimo.

Questo libro è nato dagli appunti di un corso; è perciò impossibile ringraziare per tutti i contributi. Tuttavia, devo un particolare ringraziamento ai miei colleghi E. W. Dijkstra (Eindhoven), C.A.R. Hoare (Belfast) e P. Naur (Copenaghen), che, con il loro contributo, hanno influito profondamente non solo su questo testo, ma sull'intero soggetto della programmazione. Con particolare gratitudine, ricordo qui le mie discussioni con H. Rutishauser: come "autore" dell'idea di linguaggio di programmazione, e soprattutto come un autore dell'ALGOL 60, e colui che ha maggiormente influenzato questo libro. Infine ringrazio i miei collaboratori U. Amman, E. Marmier e R. Schild per il loro valido aiuto nella stesura del compilatore PASCAL,

che dimostra come la notazione qui usata sia adatta non solo a formulare degli algoritmi astratti, ma anche a scrivere dei programmi efficienti per calcolatori reali.

La seconda edizione di questo libro fornisce l'occasione attesa di tener conto delle esperienze fatte negli ultimi due anni, che hanno mostrato l'esigenza di piccole modifiche del linguaggio di programmazione PASCAL esposto in 12, e che hanno condotto a una sua revisione. Le differenze riguardano i capitoli relativi alle operazioni sui file (Capitolo 10) e alle procedure con parametri (12.3). Nel paragrafo 12.3, l'uso della costante parametrica viene sostituito con l'uso della chiamata per valore dell'ALGOL. Nel capitolo 12 si fa inoltre un breve paragone fra i tre principi di sostituzione più noti. La seconda modifica riguarda le operazioni sui file e, in particolare, sui textfile: si è abbandonato il presupposto che l'alfabeto contenga uno speciale simbolo di separazione delle righe; ciò non rappresenta una semplificazione, ma l'adeguamento alla prassi corrente.

Zurigo, primavera 1974.

Niklaus Wirth

L'impiego dei calcolatori in campo amministrativo, industriale e scientifico si è ampliato notevolmente in questi ultimi anni. In questi rami i calcolatori sono diventati strumenti indispensabili, che permettono di svolgere lavori altrimenti impossibili. Un calcolatore è un "automa", il cui comportamento segue delle regole ben precise: esso è in grado di "comprendere" un repertorio assai ristretto di istruzioni elementari e di eseguirle con una enorme rapidità e con la massima precisione. La capacità di eseguire rapidamente lunghe successioni di istruzioni permette di combinare in modi quasi infiniti le poche azioni elementari che il calcolatore sa eseguire; ciò spiega come un calcolatore sia adatto a risolvere i problemi più diversi. Il lavoro che si fa per costruire delle sequenze di istruzioni adatte a rappresentare i processi di calcolo desiderati è detto *programmazione*.

Si può scrivere un programma senza usare il calcolatore, e i concetti principali della programmazione si possono spiegare e comprendere senza far riferimento a esso. La programmazione è una disciplina con molte applicazioni, che richiede uno studio e un'analisi sistematica condotte con rigore matematico, e che comporta molti problemi non semplici. Tuttavia, solo da poco tempo è diventata una disciplina in modo sistematico. Ciò è dovuto al fatto che la programmazione pone dei problemi difficili, che è necessario affrontare con metodo e con solide basi teoriche, solo quando i programmi raggiungono una certa lunghezza e una certa complessità (quando sono composti da migliaia o, addirittura, da milioni di istruzioni). Prima dell'avvento dei calcolatori non esistevano certo degli "schiavi" disposti a eseguire così lunghe sequenze di istruzioni instancabilmente, con esattezza e con obbedienza assoluta. Quindi, solo l'impiego dei moderni calcolatori ha reso la programmazione una disciplina di rilievo e di utilità pratica.

Concetti fondamentali

Introdurremo in questo capitolo alcuni dei più importanti concetti della programmazione. Sono concetti-base, cioè non possono essere definiti formalmente per mezzo di concetti più elementari; vengono invece descritti informalmente e illustrati con alcuni esempi semplici.

Il concetto di *azione* è il più importante. Un'azione è un evento che si compie in un intervallo di tempo finito e che produce un risultato — un *effetto* — previsto e ben determinato. Ogni azione modifica lo *stato* di un qualche *oggetto*, e il suo effetto può essere riconosciuto dal *cambiamento di stato* dell'oggetto in questione. Inoltre, è necessario disporre di un *linguaggio* o di una notazione formale che descriva le azioni: le descrizioni sono dette *istruzioni*.

Se un'azione può essere scomposta in azioni più semplici, è chiamata *processo*. Se tali azioni sono eseguite una di seguito all'altra, il processo è detto *sequenziale*. Analogamente, accade che un'istruzione descriva le singole azioni di un processo per mezzo di istruzioni più semplici: in tal caso, è chiamata *programma*. Un programma è dunque composto da un certo numero di istruzioni; in generale, non vi è corrispondenza fra la successione temporale delle azioni a esse corrispondenti.

Chiameremo *processor* quell'entità che esegue le azioni secondo le istruzioni e i processi secondo i programmi. *Processor* è un nome "neutro", che non indica se si tratti di un esecutore umano oppure automatico. D'altronde, se il linguaggio di programmazione è definito in modo sufficientemente preciso, il significato dei programmi non dipende dal particolare processor. Quindi il programmatore deve conoscere le caratteristiche del processor solo quel tanto che gli serve per capire il linguaggio di programmazione che descrive

le norme di comportamento del processor medesimo. Egli deve conoscere il tipo di istruzioni che il processor è in grado di eseguire e deve formulare i programmi in un linguaggio a esso adatto.

Ogni operazione richiede una certa *quantità di lavoro*, la cui entità dipende dal processor; è possibile misurare questa entità in base all'intervallo di tempo impiegato dal processor per eseguire l'operazione. L'intervallo di tempo può essere tradotto più o meno direttamente in un *costo*. Un programmatore esperto considera le capacità di lavoro di un processor sotto questo aspetto e sceglie, fra i diversi processi, quelli che portano al risultato desiderato con il costo minimo.

Questo libro si occupa soprattutto della costruzione di programmi che possono essere eseguiti da un *calcolatore*, e quindi da un processor automatico. Nel prossimo capitolo verranno descritte le proprietà comuni a tutti i calcolatori. Prima può essere utile illustrare i concetti ora esposti con due semplici esempi.

Esempio 2.1. Sia data l'istruzione:
moltiplica due numeri naturali x e y ;
indica il loro prodotto con z . (2.1)

Nella misura in cui il processor comprende questa istruzione (cioè sa cosa sono i numeri naturali e cosa significa "moltiplicatore"), ogni ulteriore dettaglio dell'istruzione è superflua. Qui supporremo però che il nostro processor:

- 1) non comprenda le frasi espresse nel linguaggio naturale, ma solo certe espressioni formali;
- 2) non sappia moltiplicare, ma solo sommare.

Innanzitutto, osserviamo che gli oggetti del calcolo sono dei numeri naturali. Per ogni coppia di numeri, il programma in questione deve descrivere il processo eseguito per moltiplicarli; perciò, al posto dei numeri, si usano nomi che denotano degli oggetti variabili e che son detti, appunto, *variabili*. All'inizio di ogni processo, bisogna "assegnare" alle variabili dei valori particolari.

L'*assegnamento* di un valore a una variabile è l'azione più importante nei processi eseguiti da un calcolatore. Possiamo paragonare una variabile a una lavagna: è sempre possibile leggervi quanto scritto, ed è del pari possibile cancellare e riscrivervi dell'altro. L'assegnamento di un valore w a una variabile v viene solitamente indicato con il simbolo " $:=$ ", detto *operatore di assegnamento*

$v := w$

(2.2)

Così, l'istruzione dell'*Esempio 2.1* può essere descritta formalmente come segue:

$$z := x * y \quad (2.3)$$

Se questa istruzione viene scomposta in più addizioni, da eseguire l'una di seguito all'altra, l'azione di moltiplicazione diventa un processo sequenziale e l'istruzione (2.3) diventa un *programma*. Una prima formulazione di tale programma è la seguente:

passo 1: $z := 0$
 $u := x$

passo 2: $z := z + y$
 $u := u - 1$
finché $u = 0$

Una volta assegnati i valori di x e di y , il programma dà luogo a un *processo*, che può essere rappresentato annotando in una tabella la successione dei valori assegnati alle variabili durante l'esecuzione. Con $x = 5$ e $y = 13$, la tabella è la 2.I.

TABELLA 2.I.

Passo	Valore della variabile	
	z	u
1	0	5
2	13	4
2	26	3
2	39	2
2	52	1
2	65	0

Il processo termina, conformemente all'istruzione "passo 2", quando $u = 0$; allora z possiede il valore finale $65 = 5 * 13$.

Chiameremo *traccia* una tabella del tipo precedente. Si osservi che una tabella fatta così descrive un unico processo, durante il quale ogni variabile possiede, a ogni passo, un unico valore. Si osservi anche che l'azione di assegnamento scrive "sopra" al precedente valore di una variabile cioè che il vecchio valore viene cancellato dal nuovo.

Gli oggetti sui quali opera il processo descritto sono dei numeri interi. Ora, per eseguire delle operazioni su dei numeri, è necessario

rappresentarli in una forma specifica; ciò significa che, per eseguire il processo, è necessario scegliere una particolare *rappresentazione* dei numeri. Tuttavia, il programma resta valido indipendentemente dalla rappresentazione scelta. Perciò, è necessario distinguere fra le cose in sé — per esempio i numeri come oggetti astratti — e la loro rappresentazione. Così, nei calcolatori i numeri sono rappresentati dagli stati di elementi di memoria magnetici, mentre i programmi che descrivono i processi sui numeri sono formulati in un linguaggio che non contiene alcun riferimento a tali stati; un altro esempio, che illustra la distinzione fra oggetti e rappresentazione, è fornito dalla tabella 2.II, che rappresenta esattamente, ma con numeri romani, lo stesso processo già descritto dalla tabella 2.I.

TABELLA 2.II.

Passo	Valore della variabile	
	z	u
1	0	V
2	XIII	IV
2	XXVI	III
2	XXXIX	II
2	LII	I
2	LXV	0

Esempio 2.2. Consideriamo l'istruzione: "Dividere un numero naturale x per un numero naturale y ; indicare il quoziente intero con q e il resto con r ".

Valgono le relazioni:

$$x = q * y + r \quad \text{e} \quad 0 \leq r < y \quad (2.4)$$

Rappresenteremo la precedente istruzione per mezzo della "istruzione formale":

$$(q, r) := x \text{ div } y \quad (2.5)$$

A sua volta l'istruzione (2.5) dà luogo a un programma, se supponiamo che il processo non conosca l'operazione *div*, ma sia in grado di eseguire solo somme e sottrazioni. In tal caso la divisione viene eseguita sottraendo y da x ripetutamente, finché non si ottiene il numero q delle sottrazioni possibili.

Passo 1: $q := 0$
 $r := x$ (2.6)

Passo 2: fintanto che $r \geq y$, ripetere

$q := q + 1$
 $r := r - y$

x e y rappresentano ancora delle variabili cui è necessario assegnare determinati valori all'inizio del processo. Il processo descritto dal programma (2.6) per i valori $x = 100$ e $y = 15$ è rappresentato dalla tabella di traccia (2.III).

Tabella 2.III.

Passo	Valore della variabile	
	q	r
1	0	100
2	1	85
2	2	70
2	3	55
2	4	40
2	5	25
2	6	10

Il processo termina quando $r < y$. I risultati sono $q = 6$ e $r = 10$.

Entrambi gli esempi sono descrizioni di processi sequenziali, nei quali le azioni (assegnamenti) vengono eseguite in stretta successione temporale. Nel seguito prenderemo in considerazione soltanto processi sequenziali e il termine processo andrà inteso sempre come abbreviazione di *processo sequenziale*. Questa scelta restrittiva è stata fatta di proposito, non solo perché, normalmente, i calcolatori eseguono dei processi sequenziali, ma soprattutto perché la stesura e la verifica di programmi che regolano processi non sequenziali rappresenta un compito difficile, che richiede una profonda conoscenza della programmazione dei processi sequenziali.

Gli esempi precedenti mostrano che un programma descrive delle trasformazioni di stato delle proprie variabili. Se uno stesso programma viene eseguito due volte, con valori d'ingresso (x e y) differenti, esso dà luogo a due processi differenti. Però tutti i processi descritti da uno stesso programma seguono le stesse *regole di comportamento*. Una descrizione delle regole di comportamento che non faccia riferimento a un particolare processor, sarà chiamata *algoritmo*.

Il termine *programma* si riferisce a un algoritmo formulato in modo che sia possibile la sua esecuzione da parte di un certo processor, o di un certo tipo di processor. La principale differenza fra un *algoritmo generale* e un *programma per calcolatore*, consiste nel fatto che in un programma ogni dettaglio deve essere descritto con precisione in un formalismo che obbedisce a regole molto rigide. Le ragioni di ciò sono le limitate *istruzioni di macchina* che la macchina sa riconoscere ed eseguire, e la *obbedienza assoluta* con cui le esegue, dovuta alla sua totale mancanza di *senso critico*. La mancanza di elasticità è criticata da tutti (o quasi) coloro che iniziano a programmare, per la pignoleria necessaria con i calcolatori, dove persino un banale errore di scrittura può causare un comportamento del tutto imprevisto. L'evidente mancanza di ogni *senso comune*, al quale si possa richiamare chi scrive un programma, è stata criticata anche da esperti professionisti, e vi sono stati molti tentativi di porre un rimedio a questo apparente difetto. Ma il programmatore esperto impara ad apprezzare questa *attitudine servile* del calcolatore come una qualità positiva, che permette di ottenere anche comportamenti *inusitati* — che non si potrebbero pretendere da un (umano) processor che arrotondasse ogni istruzione secondo l'interpretazione per lui più plausibile —.

3

Cenni sulle parti principali di un calcolatore

Per scrivere un programma da eseguire con un calcolatore, il programmatore deve conoscere lo strumento che utilizza. Più è precisa la sua conoscenza del processor, tanto meglio egli è in grado di trasformare un algoritmo generale in un programma che sfrutta le particolari capacità di quel processor; ma, d'altro canto, migliori sono le prestazioni del programma, maggiore è il lavoro necessario per ottenerlo. Di solito è utile scegliere una giusta via di mezzo, ricorrendo solo agli accorgimenti che sono facili da eseguire e che comportano un sensibile risparmio di tempo di macchina. Per questo scopo è sufficiente conoscere le caratteristiche più importanti comuni a tutti i calcolatori; queste caratteristiche sono descritte qui di seguito, mentre sono ignorate le peculiarità di particolari modelli di calcolatore.

In tutti i calcolatori digitali oggi in uso, si possono individuare due parti principali.

- 1) La *memoria*: essa contiene in forma codificata gli oggetti manipolati durante il processo di calcolo. Questi oggetti codificati vengono chiamati *dati*. Le capacità di lavoro della memoria sono misurate in base alle sue dimensioni e alla velocità con cui i dati possono esservi depositati, o prelevati da essa. La memoria ha sempre dimensione finita.
- 2) Il *processor* (unità logico-aritmetica): in tale unità vengono eseguite addizioni, moltiplicazioni, confronti. I dati vengono prelevati (letti) dalla memoria e depositati (scritti) nella memoria, facendo in modo che il processor contenga solo i dati di volta in volta necessari.

Il processor può contenere un numero assai limitato di dati; le sue unità (celle) di memoria vengono chiamate *registri*. Tutti

i dati non immediatamente necessari per la elaborazione sono custoditi nella memoria, che così ha la funzione di *magazzino*. Per esempio, il calcolo di una espressione aritmetica con più operandi viene eseguito immagazzinando temporaneamente nella memoria dei risultati intermedi, come mostrato qui di seguito.

Il calcolo della espressione

$$a * b + c * d \quad (3.1)$$

viene diviso nelle azioni specificate dalle istruzioni:

```
R1 := a
R2 := b
R1 := R1 * R2
z := R1
R1 := c
R2 := d
R1 := R1 * R2
R2 := z
R1 := R1 + R2
```

(3.2)

dove R1 e R2 indicano due registri del processor e z indica il risultato intermedio memorizzato. Il risultato del processo è posto nel registro R1. In tal modo il calcolo della espressione è stato trasformato in un programma che contiene solo istruzioni di tre tipi:

- 1) trasferire i dati da una cella di memoria in un registro;
- 2) eseguire operazioni (aritmetiche) sui dati contenuti nei registri;
- 3) trasferire nella memoria i dati contenuti nei registri.

Questo metodo, consistente nello scomporre le istruzioni in passi elementari e nell'immagazzinare i risultati finali e i risultati intermedi in memoria, rappresenta la quintessenza del calcolo digitale; esso spiega come le stesse elaborazioni possono essere eseguite sia dai calcolatori più complessi, che da quelli più semplici (questi ultimi impiegano semplicemente un tempo maggiore). Infatti, la decomposizione in azioni elementari permette di utilizzare sistemi automatici relativamente semplici per risolvere problemi assai complessi, purché questi sistemi siano in grado di eseguire sequenze di istruzioni molto lunghe (miliardi di istruzioni) in modo esatto (e possibilmente con rapidità). La realizzazione pratica di processor con questa caratteristica è uno dei veri trionfi della moderna tecnologia.

Il breve esempio precedente mostra anche che è necessario uno stretto collegamento tra il processor e la memoria, poiché la quantità

delle informazioni scambiate tra le due componenti è molto elevata. Inoltre, il processore deve aver accesso ai dati contenuti nella memoria, individuandoli con dei nomi (per esempio, a , b , z , ...). Di conseguenza, ci deve essere un ordinamento preciso nella memoria. I dati sono contenuti in una successione ordinata di *celle di memoria*; all'interno della successione, ogni cella è individuata da un unico indirizzo. Per prelevare un dato dalla memoria, il processore deve fornire l'*indirizzo* della cella che lo contiene.

Le celle della memoria di un calcolatore possono essere paragonate a dei normali contenitori (quali, per esempio, le cassette di sicurezza, le scatole delle scarpe, ecc): vi si possono depositare e conservare degli oggetti. Ma, mentre un contenitore contiene fisicamente un oggetto, una cella di memoria contiene invece una *rappresentazione* dell'oggetto, fornita dallo *stato* della cella. Perciò è necessario che la cella possa assumere un certo numero di *stati*. La realizzazione di componenti in grado di assumere e mantenere un grande numero di stati, distinguibili chiaramente l'uno dall'altro, è difficile sul piano tecnico. È molto più facile realizzare componenti che assumono solamente due stati. Essi sono chiamati elementi di memoria *binari*. Ora, se si considerano n elementi binari, i loro stati possono essere combinati in 2^n modi distinti; quindi essi, se trattati come una unica entità, realizzano una cella di memoria che può assumere 2^n stati distinti.

Un utile esempio di oggetti rappresentati da n elementi binari, è dato dalla scrittura dei numeri naturali nel sistema binario: un numero x è rappresentato da una successione di n cifre binarie (0 e 1, corrispondenti ai due stati di un elemento binario) nel modo seguente:

$$x : b_{n-1} \dots b_1 b_0 \quad (3.3)$$

dove x è determinato dalla regola di codificazione:

$$x = b_0 + 2b_1 + 4b_2 + \dots + 2^{n-1}b_{n-1} \quad (3.4)$$

Questa regola non è certo l'unica possibile; tuttavia essa è, per molti aspetti, la più appropriata. Dopo tutto, si tratta della regola usata nella rappresentazione decimale dei numeri:

$$x : d_{m-1} \dots d_1 d_0 \quad (3.5)$$

$$x = d_0 + 10d_1 + \dots + 10^{m-1}d_{m-1} \quad (3.6)$$

Esempi di numeri codificati in binario e in decimale sono riportati nella tabella 3.1.

TABELLA 3.1.

Binario	Decimale
1101	13
10101	21
111111	63
1101011	107

La considerazione più importante, che si può trarre da questo esempio, è che celle di memoria finite possono contenere dei numeri appartenenti solo a insiemi finiti di valori. In un calcolatore le celle di memoria indirizzabili (dette *parole*) sono formate tutte dallo stesso numero di elementi binari; questo numero è detto *lunghezza di parola* del calcolatore. Il funzionamento della unità aritmetica di un calcolatore è adattato alla sua lunghezza di parola. Valori usuali di questa lunghezza sono 8, 16, 24, 32, 48, 64.

Per poter eseguire un programma, il calcolatore deve poter accedere a quel programma "con facilità". Dove si deve trovare, di conseguenza, il programma? Fu un'idea geniale di John von Neumann (anche se oggi quest'idea sembra ovvia), che anche i programmi dovessero essere collocati nella memoria. Dunque, la medesima memoria contiene sia i dati da elaborare che il programma da eseguire.

Una tale soluzione richiede chiaramente che anche le istruzioni siano codificate. Nel precedente esempio (calcolo di un'espressione), ogni azione può essere rappresentata per mezzo di un *codice d'operazione* (che specifica l'azione di lettura, di scrittura, di addizione, di moltiplicazione, ecc.). Se, oltre alle azioni, vengono rappresentati con dei numeri anche gli indirizzi delle celle di memoria (cella 0, 1, 2, ecc.), il problema è risolto: ogni programma può esser rappresentato da una sequenza di numeri (o di gruppi di numeri) e in tal modo può essere collocato nella memoria del calcolatore. Una conseguenza di cui bisogna tener conto, è che il programma occupa un certo numero di celle di memoria, proporzionale alla sua lunghezza: queste celle non sono più disponibili per la memorizzazione dei dati. Il programmatore deve quindi stare attento a non costruire programmi che occupino uno spazio maggiore di quello disponibile.

Dall'idea di collocare nella stessa memoria i dati ed i programmi, derivano le caratteristiche più importanti dei moderni calcolatori digitali:

- 1) non appena l'esecuzione di un programma è terminata, può essere accettato un nuovo programma; (flessibilità, possibilità di molte applicazioni);

- 2) il calcolatore può generare (secondo un programma assegnato) delle sequenze di numeri che in un secondo tempo verranno interpretate e trattate come programmi, i dati generati nel primo passo diventano programmi nel secondo;
- 3) si può fare in modo che un calcolatore consideri le sequenze dei numeri che rappresentano i programmi come dei dati, da trasformare (secondo qualche programma di traduzione), in sequenze di numeri che rappresentino gli stessi programmi codificati per un calcolatore differente.

4

Ausilii alla programmazione e sistemi di programmazione

Fin verso il 1960, la programmazione consisteva ancora nella traduzione minuziosa delle istruzioni in numeri binari, ottali o esadecimali, e la successione dei numeri corrispondenti alle istruzioni forniva il programma, pronto per essere eseguito dal calcolatore. Questo lavoro di traduzione era chiamato *codificazione*. Ma un procedimento così lungo e laborioso diventa sempre più inadeguato rispetto alla crescente velocità e grandezza dei calcolatori.

1) Nel lavoro di codificazione, il programmatore era costretto ad adattare il programma alle caratteristiche del calcolatore utilizzato. Doveva perciò conoscere nel modo più preciso tutti i dettagli della macchina, delle sue istruzioni e della organizzazione interna del processor. Era impossibile adattare i programmi a macchine diverse e il fatto di conoscere i metodi di codificazione adatti a un calcolatore non era di nessun aiuto nel lavoro di codificazione su un altro calcolatore. Ogni centro metteva a punto i propri programmi ed era poi costretto, acquistando un nuovo calcolatore, ad abbandonarli e a iniziare nuovamente il lavoro di codificazione. Divenne chiaro che la scelta di adeguare gli algoritmi alle caratteristiche della macchina non era un uso sensato del lavoro umano.

2) Lavorando sempre con lo stesso tipo di calcolatore, il programmatore era portato a usare le conoscenze acquisite con l'esperienza, per inventare tutti i trucchi immaginabili per ricavare il massimo dalle capacità di qual calcolatore. Quando la programmazione "artistica" — la "truccologia" — era di gran moda, il programmatore impiegava gran parte del suo tempo per mettere a punto dei programmi "ottimali", che però erano molto difficili da verificare. Era pressoché impossibile risalire alla struttura logica di un programma scritto da un collega di lavoro (e, spesso, era difficile com-

prendere un programma, persino per il suo stesso autore!). La programmazione "artistica" oggi non è più considerata un buon metodo: un programmatore intelligente evita l'uso dei trucchi.

3) Il così detto *codice di macchina* conteneva ben poche ridondanze che facilitassero l'individuazione degli errori di codificazione. Anche gli errori di battitura, che potevano avere conseguenze disastrose nella esecuzione del programma, erano assai difficili da individuare.

4) Una sequenza lineare di istruzioni, informe e priva di struttura, non è una forma adeguata per rappresentare i programmi, perché non corrisponde al modo nel quale l'uomo si esprime abitualmente. Vedremo come la presenza di una struttura sia lo strumento principale per costruire in modo sistematico (*sintetizzare*) i programmi e per renderli di più facile e immediata comprensione.

I difetti ora elencati portarono allo sviluppo dei cosiddetti *linguaggi di programmazione di alto livello*. Un linguaggio di programmazione di alto livello può essere considerato come il formalismo nel quale si esprimono le istruzioni di una *macchina ideale*, costruita a misura d'uomo, al di là delle possibilità tecniche. Cioè, si può pensare che vi siano due macchine: una macchina *A*, realizzabile sul piano tecnico ed economico, ma complicata da usare, e una macchina *B*, progettata a misura d'uomo, ma solo sulla carta. Il legame tra *A* e *B* è fornito dai così detti sistemi *software* (la macchina *A* è chiamata *hardware*). Un sistema *software* è un programma *C*, che può essere eseguito sulla macchina *B* in programmi adatti alla macchina *A*. Il programma *C* viene chiamato traduttore o *compilatore* e fornisce alla macchina *A* la capacità di accettare e di eseguire i programmi scritti per la macchina *B*.

L'utilizzazione del compilatore *C* permette al programmatore un certo distacco dalle caratteristiche del calcolatore *A*; ma egli deve sempre assicurarsi che, alla fine, il programma possa essere eseguito dal calcolatore *A*, entro i limiti di memoria consentiti.

Di norma, un sistema hardware/software elabora un programma *P* in due fasi successive. Nella prima fase il programma *P* viene tradotto dal compilatore *C* in una forma che il calcolatore *A* è in grado di interpretare direttamente. Questa fase viene chiamata *fase di traduzione* o di *compilazione*. Nella seconda fase il programma tradotto viene eseguito; essa si chiama perciò *fase di esecuzione*.

Fase di traduzione:

programma = compilatore *c*

dati di ingresso = programma *P* nel linguaggio *B*

dati di uscita = programma *P'* nel linguaggio *A*

Fase di esecuzione:

programma = *P'*

dati di ingresso = *X*

dati di uscita = risultati del calcolo *Y*.

5

Semplici esempi
di programmi

Dal capitolo precedente risulta chiaro che un programma deve essere composto da una sequenza di istruzioni che il calcolatore *comprende*. Anche se non sappiamo ancora con precisione che cosa *comprende* un calcolatore, e quali sono i tipi e le forme delle istruzioni contenute in un linguaggio di programmazione, sappiamo però che le istruzioni debbono specificare *esattamente* le azioni desiderate. Questa ineliminabile necessità di esattezza costituisce forse la differenza principale fra la comunicazione con le macchine e la comunicazione tra gli uomini. Per lavorare con i calcolatori, bisogna usare espressioni esatte. Mancanza di chiarezza, imprecisione o ambiguità devono essere escluse.

Gli *schemi di flusso* sono un modo facilmente comprensibile e molto usato di rappresentare i programmi. Per esempio il programma (2.1) è rappresentato dallo schema di flusso riportato nella figura 5.1.

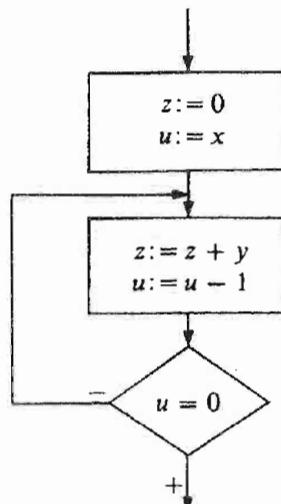


Figura 5.1.

Questo tipo di rappresentazione grafica fornisce una immagine chiara di come un'azione succede alla precedente. Infatti, vi sono rappresentati due tipi di istruzioni:

- 1) gli assegnamenti, racchiusi nei rettangoli;
- 2) le decisioni, racchiuse nei rombi.

Ad una decisione seguono più istruzioni ed essa indica una scelta. Se la condizione da essa specificata è soddisfatta, viene scelta l'*uscita* indicata con $+$; in caso contrario viene scelta l'*uscita* indicata con $-$. Una ripetizione è indicata da un *loop*, cioè da una sequenza ciclica di istruzioni contenente almeno una decisione, che determina la fine della ripetizione.

Allo stesso modo, il programma (2.6) è rappresentato dallo schema di flusso della figura 5.2.

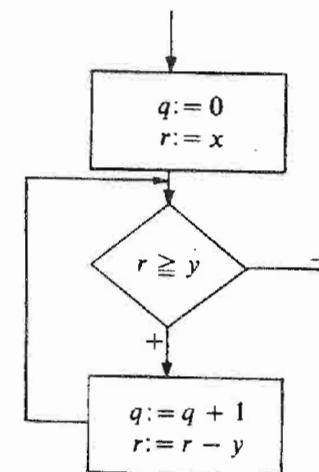


Figura 5.2.

Un programma dà le regole di comportamento di un numero in generale impreciso, per lo più infinito, di processi. I singoli processi presentano lo stesso modello di comportamento, ma differiscono per i valori assunti dalle variabili in ogni istante e, in particolare, per i valori iniziali. Quindi occorre verificare che tutti i processi, che obbediscono a un dato programma, conducano ai risultati desiderati; cioè, si pone il problema di verificare la *correttezza dei programmi*.

La correttezza dei programmi (2.1) e (2.6) è stata dimostrata nelle tabelle 2.I e 2.III, per una coppia di valori di x e di y fissata. Questo modo di stabilire la correttezza è detto *test del programma*; esso consiste nella scelta di opportuni valori da attribuire agli argo-

menti (x e y), nell'esecuzione del processo con i valori scelti e nel confronto del risultato del calcolo con il risultato corretto, noto in precedenza. Il mezzo ideale per l'esecuzione dei *test* è, naturalmente, il calcolatore stesso. Ciò nonostante, questo metodo tradizionale è lungo e costoso, in particolare quando il numero dei test da eseguire è elevato. Ma esso è soprattutto insufficiente. Infatti è un metodo completamente empirico, per cui, a rigore, sarebbe necessario eseguire un test per ogni possibile valore d'ingresso. Ma ciò è impossibile anche nei casi più semplici, perché è necessario eseguire il programma troppe volte: consideriamo, per esempio, un programma che esegue la moltiplicazione di due numeri.

Supponendo che un dato calcolatore impieghi 1μ sec per eseguire una moltiplicazione, e che in esso si possano rappresentare tutti i numeri interi fino a un valore assoluto di 2^{60} , il tempo necessario per una verifica esaustiva della moltiplicazione è di:

$$2^{2 \cdot 60} * 10^{-6} \text{ sec} \doteq 3.2 * 10^{22} \text{ anni}$$

Da questo esempio, che dimostra l'impossibilità di un *test empirico* esaustivo, si deduce la seguente regola generale:

"l'esame di un programma mediante un test empirico può, al più, rilevare la presenza di errori, ma non può garantire l'assenza di errori, cioè la correttezza del programma".

È quindi necessario astrarre dalle proprietà dei singoli processi, e ricavare, direttamente dalle regole di comportamento, delle proprietà valide in generale. Tale metodo di prova analitica viene chiamato *verifica dei programmi*. Mentre con il metodo del test empirico l'oggetto della prova è il singolo processo, con la verifica esso diviene l'intero programma.

Nella sostanza la verifica si basa sugli stessi principi del test empirico; ma, invece di elencare in una tabella la successione dei valori assunti dalle variabili durante l'esecuzione, si annotano, dopo ogni istruzione, delle relazioni fra i valori delle variabili, *valide in generale*; qui, *valida in generale* significa: *validaogniqualvolta il processo raggiunge il punto di annotazione, indipendentemente dalle azioni precedentemente eseguite*.

Esporremo ora quattro *regole-base* di verifica:

- 1) prima e dopo di ogni istruzione, si annotano una o più relazioni fra le variabili, che valgono in generale prima e dopo l'esecuzione dell'istruzione; tali relazioni vengono anche chiamate *condizioni di verifica* (asserzioni); la condizione che precede una istruzione I si chiama *premessa* di I (antecedente); quella successiva a I si chiama *conseguenza* di I (conseguente) (v. fig. 5.3);

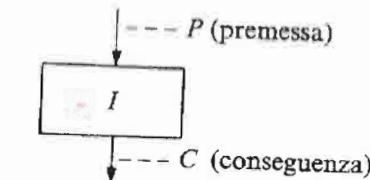


Figura 5.3.

- 2) quando più cammini del programma si congiungono nel punto che precede un'istruzione B , la premessa P_B deve essere logicamente implicata dalle conseguenze $C_{I_1}, C_{I_2}, \dots, C_{I_n}$ di tutte le istruzioni I_1, I_2, \dots, I_n che precedono B (v. fig. 5.4);

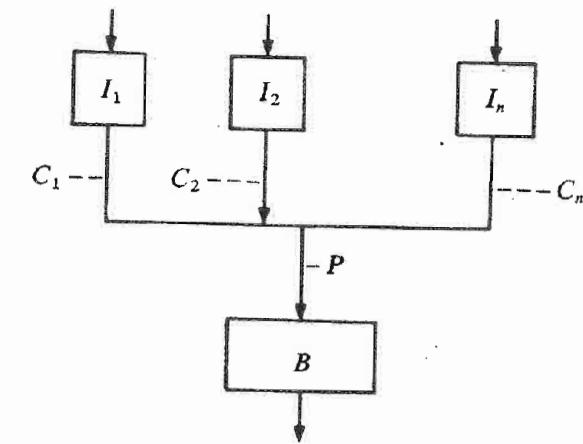
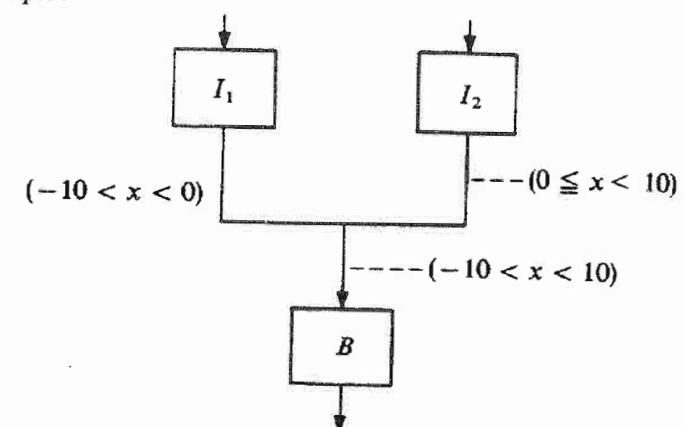


Figura 5.4.

Esempio:



3) se vale la condizione di verifica P prima di una decisione contenente la condizione D , allora vale lo schema della figura 5.5;

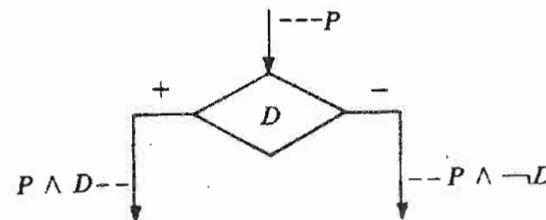


Figura 5.5.

4) se vale la condizione di verifica $P(w)$ prima dell'assegnamento della espressione w alla variabile v , allora dopo di esso vale $P(v)$ (v. fig. 5.6).

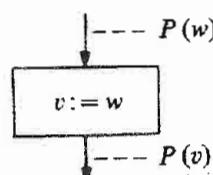


Figura 5.6.

Esempi (v. fig. 5.7).

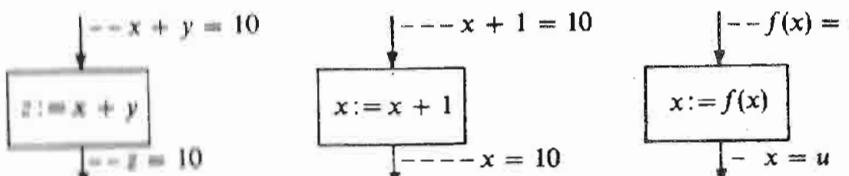


Figura 5.7.

Come esempio di applicazione delle precedenti regole base eseguiremo la verifica dei programmi riportate nelle figure 5.1 e 5.2; le condizioni di verifica riportate nelle figure 5.8 e 5.9 servono come passo intermedio della verifica; esse possono venire dedotte facilmente, seguendo le regole base indicate nelle figure 5.3 e 5.6.

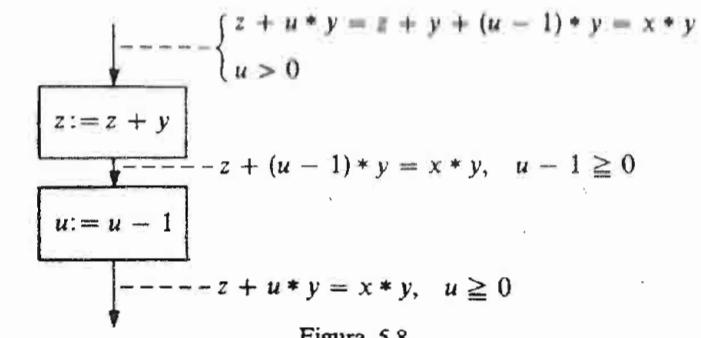


Figura 5.8.

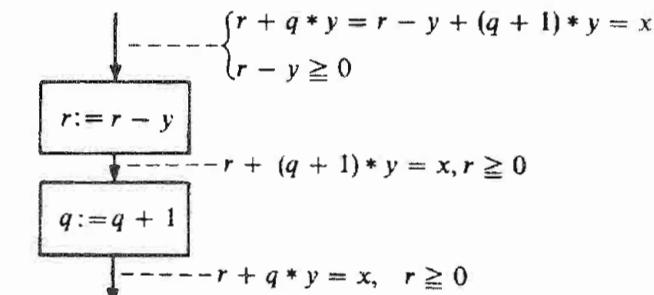


Figura 5.9.

I programmi con le condizioni di verifica complete sono riportati nelle figure 5.10 e 5.11.

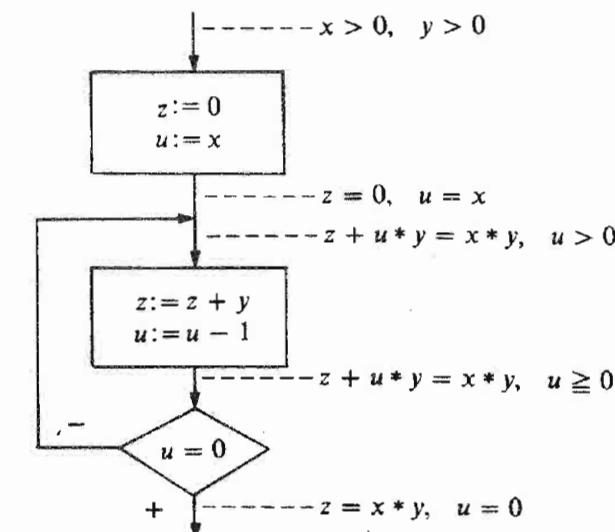


Figura 5.10.

Osserviamo, per inciso, che la separazione degli assegnamenti in due rettangoli distinti, fatta nella figura 5.8 e 5.9, prescrive rigidamente l'ordine in cui debbono venir eseguiti gli assegnamenti; ma ciò non è indispensabile. Infatti valgono le medesime premesse e le medesime conseguenze anche se si cambia l'ordine delle istruzioni. Il programma risulta così, in un certo senso, sovradocumentato; se una documentazione ridondante non disturba in programmi così brevi, essa diventa illeggibile, e quindi inutile, per programmi più lunghi.

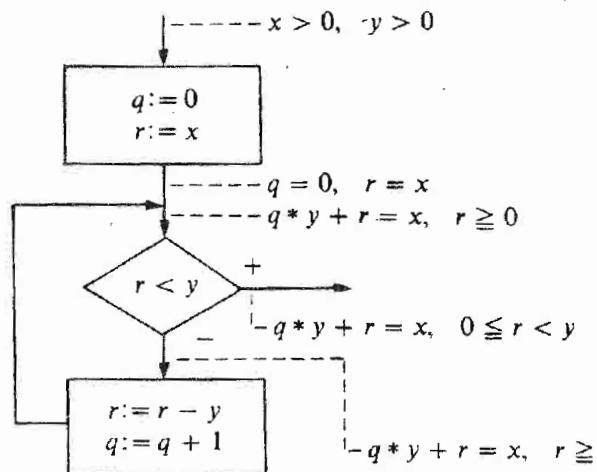


Figura 5.11.

La determinazione delle condizioni di verifica di una sequenza lineare di istruzioni in base alle regole delle figure da 5.3 a 5.7 è semplice, quando siano assegnate la prima premessa o l'ultima conseguenza. Si incontrano invece delle difficoltà non appena entrano in gioco delle ripetizioni, cioè, non appena in un programma si presenta un cammino che si chiude su se stesso. Il ciclo viene allora *tagliato* e si fa l'*ipotesi* che nel *punto di taglio* valga una certa condizione di verifica; se la conseguenza dell'ultima istruzione della sequenza lineare così ottenuta implica logicamente la premessa della prima istruzione, l'*ipotesi* fatta è valida ed il ciclo può venire richiuso (v. la regola espressa nella fig. 5.4). Di norma, è utile scegliere il *punto di taglio* in corrispondenza della decisione che determina la terminazione del ciclo; in tal caso, la condizione di terminazione e la condizione di verifica, congiunte, forniscono la conseguenza della istruzione di ripetizione.

La premessa relativa al punto di taglio viene chiamata *invariante* ciclico, poiché essa rappresenta una relazione valida per qualsiasi

numero di ripetizioni dell'istruzione ciclica, cioè invariante. Nei programmi delle figure 5.10 e 5.11 gli invarianti sono:

$$\begin{aligned} & (z + u * y = x * y) \wedge (u \geq 0) \\ \text{e} \quad & (q * y + r = x) \wedge (r \geq 0) \end{aligned} \quad (5.1)$$

Poiché le ripetizioni e i cicli sono le strutture fondamentali dei processi e dei programmi, è utile dare alle precedenti considerazioni una formulazione generale, introducendo opportune regole di verifica. Tali regole danno indicazioni sul tipo delle condizioni di verifica che si possono fornire per la ripetizione di un gruppo di istruzioni, del quale si conoscono la premessa e la conseguenza.

Regola di verifica 1. Sotto l'*ipotesi* che la premessa P sia invariante rispetto all'istruzione I , cioè che essa valga anche dopo l'istruzione, posto che valesse prima — rappresenteremo ciò con la figura 5.12.

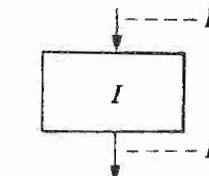


Figura 5.12.

per l'istruzione ripetitiva I' vale la *conseguenza* illustrata nella figura 5.13.

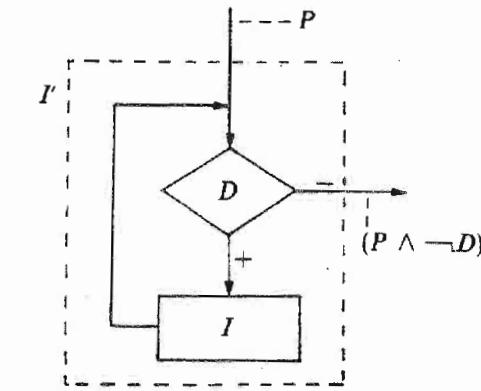


Figura 5.13.

Osservazione: l'ipotesi precedente può anche essere modificata nel modo illustrato nella figura 5.14, poiché, nel ciclo, I viene eseguita soltanto quando D è soddisfatta.

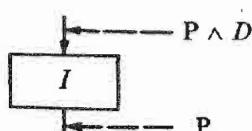


Figura 5.14.

Regola di verifica 2. Sotto le condizioni illustrate nella figura 5.15,

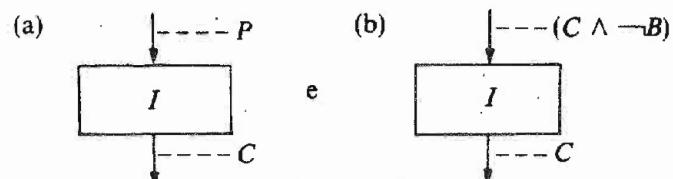


Figura 5.15.

per la istruzione ripetitiva I'' vale la conseguenza illustrata nella figura 5.16.

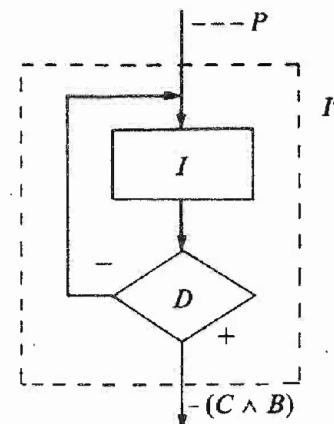


Figura 5.16.

Si osservi che, per l'istruzione ripetuta I , devono valere *due* condizioni, affinché la regola relativa a questa seconda forma-base di ripetizione sia applicabile. Questa forma, quindi, è quella che richiede la maggiore attenzione; infatti, numerosi errori di programmazione derivano dall'aver dimenticato la prima delle due condizioni. Nei casi dubbi, è raccomandabile usare la prima struttura ripetitiva, in cui la decisione D precede l'istruzione I .

Le precedenti considerazioni servono come indicazioni di fondo per verificare la correttezza e per comprendere la logica di un programma, ma servono soprattutto per far vedere come, in molte circostanze, può essere difficile determinare un invarianto ciclico adatto a un programma già esistente. La regola, che ogni programmatore dovrebbe imparare, è che l'*indicazione esplicita degli invarianti ciclici rilevanti di ogni ripetizione è la base di ogni buona documentazione dei programmi*. Anche quando un programma deve essere usato soltanto dal suo autore, l'indicazione esplicita degli invarianti può essere utile per evitare errori, che altrimenti dovrebbero essere individuati con un *test empirico* anche molto laborioso, e che spesso non vengono neppure scoperti e rimangono nel programma, presunto corretto.

Altrettanto indispensabile, per una buona documentazione dei programmi, è la indicazione esplicita delle limitazioni imposte ai valori delle variabili e, in particolare, ai valori iniziali, poiché da essi dipende il risultato. Una variabile che può assumere solo valori interi verrà indicata, per esempio, con una notazione del tipo

$v:integer$

Per concludere riportiamo qui di seguito i programmi (5.1) e (5.2) scritti in una forma compatta, e documentati in un modo adeguato, conforme alle considerazioni precedenti.

Moltiplicazione di due numeri naturali: gli argomenti sono x e y , il risultato è z e u è una variabile *ausiliaria* (o *interna*, cioè usata all'interno del programma) (v. fig. 5.17).

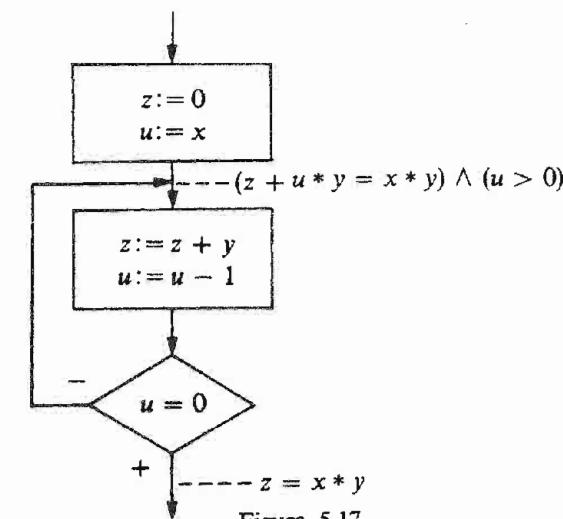


Figura 5.17.

Divisione di due numeri naturali: gli argomenti sono x e y , numeri naturali; risultati: q = quoziente intero, r = resto (v. fig. 5.18).

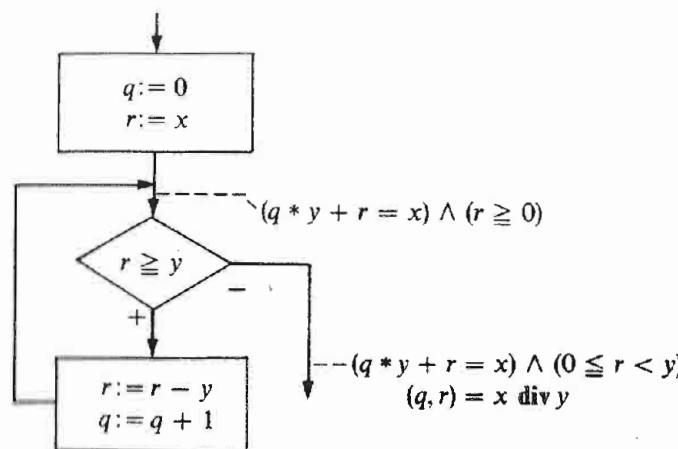


Figura 5.18.

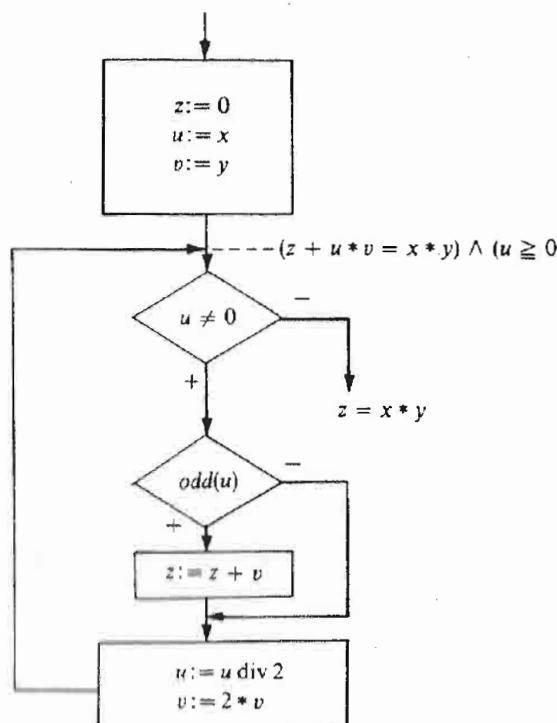


Figura 5.19

5.1. Problemi

5.1. Il programma nella figura 5.19 calcola il prodotto:

$$z := x * y$$

usando soltanto le operazioni di addizione, di raddoppio e di dimezzamento. Gli argomenti sono x e y , numeri naturali; u e v sono variabili con valori interi (ausiliarie).

Il predicato $odd(u)$ è soddisfatto quando u è dispari.

Determinare, per ogni istruzione e per ogni blocco, le premesse e le conseguenze che si possono ricavare dall'invariante ciclico dato nella figura, applicando le regole illustrate nelle figure 5.3 e 5.6.

5.2. Il programma riportato nella figura 5.20 calcola il massimo comun divisore (MCD) di due numeri naturali x e y ; a e b sono variabili ausiliarie intere, il cui valore finale rappresenta il risultato.

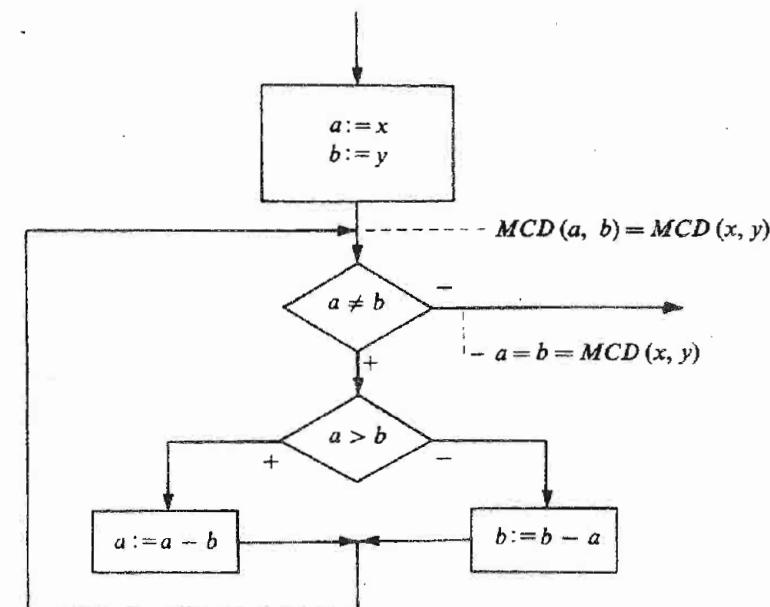


Figura 5.20

Determinare, come nell'esercizio 1, le necessarie condizioni di verifica, a partire dalle seguenti relazioni:

- 1) $u > v$: $MCD(u, v) = MCD(u - v, v)$
- 2) $MCD(u, v) = MCD(v, u)$
- 3) $MCD(u, u) = u$

5.3. Seguendo lo schema del programma della figura 5.19, costruire un programma che, per ogni coppia di numeri naturali x e y , calcoli il valore $z = x^y$. Inserire le condizioni necessarie per la verifica del programma.

Il *ciclo* è la struttura caratteristica dei programmi per calcolatore, poiché esso specifica la *ripetizione* di un'azione, e i calcolatori sono particolarmente adatti a eseguire azioni ripetitive. Preziosa è la loro caratteristica di non stancarsi mai e di non perdere in sicurezza e in precisione, anche dopo aver ripetuto migliaia di volte la stessa azione. D'altro canto, è proprio questa instancabilità del calcolatore a rendere necessaria una maggiore attenzione da parte del programmatore. Infatti, egli deve assicurarsi che tutti i processi, che possono avvenire secondo un dato programma, abbiano termine dopo un numero finito di ripetizioni; cioè, egli deve poter garantire la *terminazione* del programma. Sfortunatamente, i processi che non terminano (detti *loop* del calcolatore) sono inconvenienti costosi e abbastanza frequenti nella programmazione. Tuttavia, essi possono essere evitati usando maggiori precauzioni nella stesura e nella verifica dei programmi. Per esporre le precauzioni necessarie per garantire terminazione, ci riferiremo al semplice schema della figura 6.1.

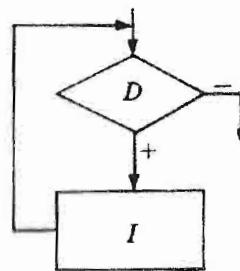


Figura 6.1.

Il minimo che si possa chiedere, è che nella istruzione *I* venga modificata (almeno) una variabile, in modo che, dopo un numero finito di ripetizioni, la condizione *D* non possa essere più soddisfatta. In generale, la terminazione di una ripetizione può essere dimostrata secondo la regola seguente: si individua una grandezza intera *N* in funzione delle variabili del programma e si dimostra che:

- 1) se *D* è soddisfatta, vale $N \geq 0$, e
- 2) l'esecuzione della istruzione *I* determina una diminuzione del valore di *N*.

Illustriamo l'utilizzazione di tale regola riprendendo in esame il programma illustrato nella figura 5.2. In esso l'istruzione *I* è:

$$\begin{aligned} r &:= r - y \\ q &:= q + 1 \end{aligned}$$

e la condizione *D* è:

$$r \geq y$$

r, *q*, *y* sono numeri naturali; scegliamo $N = r - y$. Segue che:

- 1) $r \geq y$ implica sempre $N \geq 0$, e
- 2) nella esecuzione di *I* il valore di *r* (e con esso il valore di $r - y$) diminuisce, poiché $y > 0$.

Quindi la terminazione del programma è dimostrata. (Si osservi che è indispensabile la condizione $y > 0$).

Come secondo esempio di applicazione della precedente regola, consideriamo il programma illustrato nella figura 5.20. L'istruzione *I* in esso è:

$$\begin{aligned} a &:= a - b && \text{se } a > b \\ b &:= b - a && \text{se } b > a \end{aligned}$$

e la condizione *D* è $a = b$; *a* e *b* sono numeri naturali con valori iniziali $a > 0$, $b > 0$ e con $a \neq b$.

Una scelta adeguata di *N* è:

$$N = \max(a, b)$$

L'effetto di *I* su *N* può essere studiato separatamente, a seconda che sia $a > b$ oppure $b > a$. Se $a > b$, *a* viene decrementato della quantità *b* e *b* rimane invariato. Poiché inizialmente $a > 0$, $b > 0$

e $a \neq b$, le prime due condizioni ($a > 0$, $b > 0$) vengono conservate e N diminuisce. Se invece $b > a$, a rimane invariato, $b = \max(a, b) = N$ diminuisce perché decrementato di a e vengono mantenute le condizioni $a > 0$, $b > 0$. Quindi, in entrambi i casi, $N = \max(a, b)$ diminuisce e $\min(a, b)$ rimane positivo; ne segue che, dopo un numero finito di cicli, deve diventare $\max(a, b) = \min(a, b)$ e quindi $a \neq b$ non può più essere soddisfatto.

In tal modo abbiamo dimostrato la terminazione del programma.

6.1. Problemi

6.1. Determinare le condizioni iniziali per i valori di x e di y , sufficienti per garantire la terminazione dei programmi dei problemi 5.1, 5.2 e 5.3.

6.2. Per quali intervalli dei valori x e di y il programma illustrato nella figura 6.2 termina?

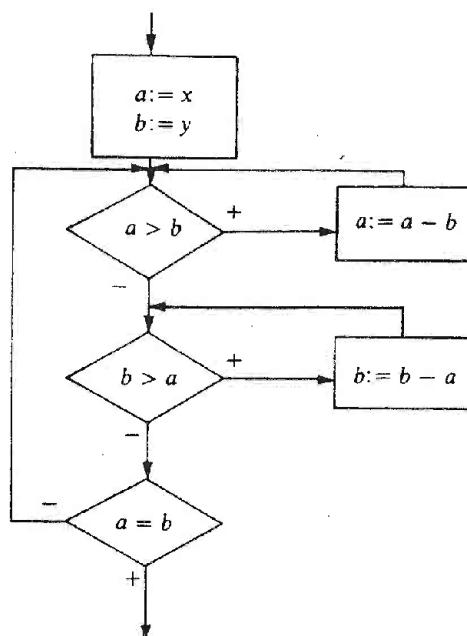


Figura 6.2.

6.3. Sotto quali condizioni iniziali il programma di cui sopra calcola il massimo comun divisore di x e di y ? Scrivere la condizione necessaria e sufficiente.

7

Notazione lineare e linguaggi di programmazione

7.1. Generalità

Un programma, rappresentato da uno schema di flusso, non può essere elaborato direttamente dal calcolatore; infatti, questa forma di rappresentazione non è accettata dai comuni dispositivi di lettura dei dati. Gli schemi di flusso devono essere *tradotti* in una forma che possa essere letta ed elaborata meccanicamente. Poiché, nella traduzione, è molto facile fare degli errori, è utile disporre di una notazione adatta a formulare i programmi e, contemporaneamente, a essere elaborata meccanicamente. Questa notazione dovrà essere definita in modo chiaro e dovrà essere facile da leggere e da capire; inoltre, dovrà poter essere elaborata direttamente dal calcolatore.

I dispositivi di *ingresso* dei dati più usati sono il lettore di schede e la telescrivente. In entrambi i casi, i dati d'ingresso sono rappresentati da *sequenze lineari di caratteri*, cioè, da *testi lineari*. Le diverse notazioni, usate per scrivere il testo lineare di un programma, sono comunemente chiamate *linguaggi di programmazione*. Le frasi di un linguaggio di programmazione sono costruite secondo regole molto rigide, che indicano con precisione quali sono le *istruzioni formali* permesse; l'insieme delle regole viene chiamato *sintassi*. Un calcolatore legge e interpreta i programmi, analizzando le istruzioni in un modo piuttosto rigido; ne segue che le regole sintattiche devono essere precise anche nei più piccoli dettagli. Quindi, per imparare un linguaggio di programmazione, bisogna imparare il significato delle frasi che si possono costruire e le regole per costruirle; queste ultime, richiedono una attenzione maggiore rispetto all'apprendimento di un linguaggio naturale, poiché è molto più importante saper inventare e formulare correttamente i programmi, che saperli leggere e capire.

A meno che non si tratti di un programma molto semplice, il lavoro più grosso consiste nell'invenzione e nella verifica dell'algo-

ritmo sul quale si basa il programma; in confronto, il lavoro di codifica in un dato linguaggio è trascurabile. La costruzione di un algoritmo avviene secondo un procedimento lungo e complesso, che comporta uno sviluppo per *passi* successivi. A ogni passo il programma viene specificato più in dettaglio. Nei primi passaggi, la notazione deve essere il più possibile adatta a descrivere il problema trattato (e all'attitudine del programmatore) e non è necessario impiegare direttamente un linguaggio di programmazione; in questa fase, sono adatti il formalismo della matematica e l'uso degli schemi di flusso. Occorre, comunque, tener presente che il linguaggio di programmazione nel quale deve essere scritta la versione finale del programma influenza l'intero processo. Quindi, è necessario esporre tale linguaggio già all'inizio di un corso di programmazione (fermo restando il fatto che lo scopo principale di un corso di programmazione dev'essere insegnare a *inventare e formulare gli algoritmi*).

La codificazione di un algoritmo in un certo linguaggio, per esempio nel codice di macchina di un calcolatore, segue un procedimento complicato, ma abbastanza meccanico. Quindi è molto utile automatizzare il processo di traduzione nel codice di macchina: in particolare, è utile sviluppare un linguaggio che fornisca una rappresentazione chiara e naturale dei concetti principali della programmazione e che, contemporaneamente, possa essere usato direttamente per la elaborazione con il calcolatore. I tentativi di sviluppare un siffatto linguaggio di programmazione, spesso, sono stati influenzati da un campo di applicazioni particolare, o da un particolare calcolatore, o da entrambi. È stato il campo dell'analisi numerica a diventare il punto di partenza per sviluppare un linguaggio conciso e dotato delle caratteristiche richieste; infatti, è stato il punto di partenza per adottare e allargare la notazione formale tradizionale della matematica, ormai ampiamente sperimentata. Questa idea fu espressa con chiarezza, per la prima volta, da H. Rutishauser, già nel 1952, ma trovò una prima attuazione solo nel 1956-57, quando la ditta IBM pubblicò un linguaggio formale, detto FORTRAN, e lo mise a disposizione dei clienti, assieme a un apposito programma, che traduceva i programmi scritti in FORTRAN in programmi scritti nel codice di macchina dei calcolatori IBM (di qui il nome *FORmula TRANslator*). Lo studio dei linguaggi formali acquistò così un interesse pratico, cioè economico, oltre che teorico. Il linguaggio FORTRAN era però ancora legato ad alcune proprietà, particolari di un certo tipo di calcolatore e, in questo senso, la sua definizione e la sua logica interna lasciavano ancora a desiderare.

Solo nel 1958 le idee originali di H. Rutishauser furono adottate

da un gruppo internazionale di studiosi, e furono attuate con la definizione di un linguaggio di programmazione. Tale linguaggio fu chiamato ALGOL (*ALGOrithmic Language*) e divenne il punto di partenza per la costruzione del linguaggio di programmazione ALGOL 60, ben noto e largamente usato in campo scientifico. L'ALGOL 60 fu costruito nel 1960, da un gruppo di tredici studiosi, e fu formulato da P. Naur; esso si differenzia dal FORTRAN, principalmente perché è definito con precisione da una documentazione relativamente breve; anziché riferirsi a un certo tipo di calcolatore, esso si richiama largamente alla già sperimentata tradizione formale della matematica. Per descriverne la *struttura sintattica*, fu usata per la prima volta una notazione che permette di decidere se un testo obbedisce alle regole sintattiche. Questa notazione è conosciuta come *forma di Backus-Naur* (BNF) ed è stata impiegata anche in seguito, per la definizione di molti altri linguaggi di programmazione (v. bibliografia [6]). Studi analoghi furono iniziati nel campo della elaborazione dei dati commerciali. Con i finanziamenti dello U.S. Department of Defense, nel 1962, fu sviluppato il linguaggio di programmazione COBOL (*Common Business Oriented Language*), adatto a risolvere i problemi di elaborazione dei dati, che si pongono in campo commerciale. Oggi, il COBOL è uno dei linguaggi più diffusi; tuttavia, esso è inferiore al FORTRAN, per quanto riguarda la chiarezza delle definizioni, la sistematicità della struttura e le proprietà delle istruzioni di programmazione.

Un effetto negativo della rapida espansione di questi primi linguaggi, fortemente orientati al loro campo di applicazione, fu la divisione della programmazione in due branche ben distinte: la programmazione *scientifica* e la programmazione *commerciale*. Considerando la programmazione principalmente come la codificazione di algoritmi già noti in un dato linguaggio, si pensava che la programmazione *scientifica* e quella *commerciale* dovessero essere due discipline distinte e che dovessero dar luogo a due corsi di studi separati. In realtà, le operazioni e i concetti fondamentali della programmazione sono sostanzialmente i medesimi e non dipendono dal campo di applicazione.

Negli anni che vanno dal 1964 al 1967 la ditta IBM affrontò il problema di riunire in un unico linguaggio i due campi: così fu sviluppato un linguaggio indipendente dalla macchina adatto a tutte le applicazioni allora note. Tale linguaggio è conosciuto col nome di PL/I. Il linguaggio e la sua descrizione hanno raggiunto una dimensione considerevole; il PL/I è un linguaggio poco adatto alla didattica, per la sua carenza di sistematicità e di chiarezza, dovuta al fatto che esso è nato dalla fusione di linguaggi differenti, senza un principio unificatore.

Tenendo presente che lo scopo principale di un corso di programmazione è l'insegnamento del modo con cui costruire i programmi, mentre del tutto secondaria è la codificazione, introdurremo e useremo una notazione che differisce da tutti i linguaggi fin qui menzionati. La notazione rispecchia i concetti elementari della programmazione in una forma naturale, comprensibile e precisa. Inoltre le regole sintattiche sono semplici, immediate e sistematiche, sì che il loro apprendimento non richiede uno sforzo particolare. Fra i linguaggi precedentemente menzionati, l'ALGOL 60 è quello che, maggiormente e meglio degli altri, si presta alle esigenze sopraccennate. La notazione che useremo è perciò definita in stretta corrispondenza all'ALGOL 60 e può essere considerata come una sua estensione (v. bibliografia [12]).

Man mano che esporremo i principali concetti della programmazione, introdurremo i termini precisi del linguaggio a essi corrispondenti. Per ora, ci limiteremo a una breve esposizione delle regole che definiscono la struttura sintattica del linguaggio.

Ogni linguaggio si basa su un *dizionario*. Le frasi, cioè i programmi, vengono costruiti disponendo i *simboli base* di questo dizionario in una sequenza lineare, secondo le *regole sintattiche* (o regole di costruzione). Il dizionario è composto da lettere, da cifre e da simboli speciali (per esempio, +, -, *). Poiché il numero dei simboli speciali usati in un linguaggio di programmazione è piuttosto elevato, essi sono per lo più denotati da parole inglesi, il cui significato è più immediato. In seguito, per evitare confusioni fra sequenze di caratteri e simboli base speciali, indicheremo questi ultimi scrivendoli in neretto (per esempio **begin-end**); il dizionario è dato nell'appendice A.

Le regole sintattiche sono formulate in modo che sia possibile stabilire facilmente e con rapidità se una sequenza di simboli base è permessa. Le regole sono rappresentate da diagrammi di flusso, detti *diagrammi sintattici* (v. appendice A). Ogni cammino rappresenta una successione di simboli permessa; esso inizia dal diagramma intitolato *programma* e prosegue secondo le regole seguenti:

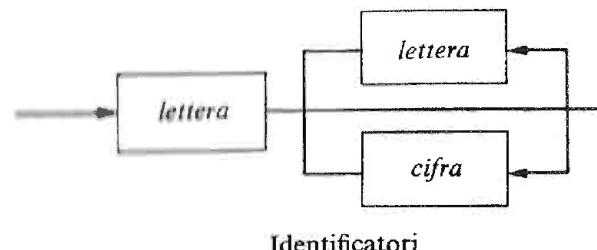


Figura 7.1.

- a) quando si incontra un rettangolo (contenente il nome di un altro diagramma), si prosegue dal diagramma indicato nel rettangolo;
- b) quando si incontra un circolo, contenente un simbolo *S*, il simbolo *S* viene inserito nel testo del programma, il cammino prosegue direttamente nello stesso diagramma. Come esempio, si veda la figura 7.1.

Secondo il diagramma ivi illustrato, le seguenti sequenze di cifre e di lettere sono possibili *identificatori*:

A
ABCDEF
A15
Q2P9
MEIER19

mentre le seguenti non sono identificatori:

SBB-FFS
C. F. MEIER
7X

7.2. Espressioni e istruzioni

La sintesi di un linguaggio definisce ben determinati costrutti. Fra le componenti sintattiche fondamentali, presenti nella struttura di un linguaggio di programmazione, particolare risalto va dato alle espressioni e alle istruzioni.

Una *espressione* è una formula (o regola di calcolo) che specifica sempre un valore (o risultato). Ogni espressione è composta da *operatori* e da *operandi*. Gli operandi possono essere delle costanti (per esempio numeri), delle funzioni o delle variabili. Gli operatori si dividono normalmente in *operatori monadici* (con un solo argomento) e in *operatori diadiici* (con due argomenti). Se una espressione contiene più d'un operatore, è necessario specificare l'ordine secondo cui eseguire le operazioni; quest'ordine è assegnato esplicitamente con delle parentesi, oppure è determinato da convenzioni implicite, contenute nelle regole sintattiche del linguaggio. Nei linguaggi di programmazione più comuni, gli operatori diadiici vengono divisi in (almeno) due classi: gli operatori *additivi* e gli operatori *moltiplicativi*. Questi ultimi appartengono alla *classe di precedenza più forte*, cioè vengono eseguiti per primi; invece, in una sequenza formata da operatori appartenenti a una medesima classe, le opera-

Istruzioni vengono eseguite procedendo da sinistra verso destra. Le regole ora esposte sono illustrate dagli esempi seguenti:

$$\begin{aligned}
 x + y + z &= (x + y) + z & (7.1) \\
 x * y + z &= (x * y) + z \\
 x + y * z &= x + (y * z) \\
 x - y * z - w &= (x - (y * z)) - w \\
 x * y - z * w &= (x * y) - (z * w) \\
 -x + y/z &= (-x) + (y/z) \\
 x * y/z &= (x * y)/z \\
 x/y * z &= (x/y) * z
 \end{aligned}$$

L'istruzione più elementare è l'assegnamento, ed è descritta dall'espressione formale:

$$V := E \quad (7.2)$$

dove V è una variabile ed E è un'espressione. Tale istruzione indica che viene calcolato il valore dell'espressione E e che il risultato viene assegnato a V . Mentre una espressione indica un *valore*, una istruzione produce un *effetto*.

Le sequenze d'istruzioni, le istruzioni condizionali e le istruzioni ripetute sono descritte da costrutti sintattici appropriati, chiamati *istruzioni strutturate*. Qui di seguito sono descritti sei tipi di istruzioni strutturate; si tratta delle strutture fondamentali di uso più frequente. Il loro significato sarà illustrato dagli schemi di flusso a esse equivalenti.

7.2.1. Istruzioni strutturate

1) Composizione di istruzioni (v. fig. 7.2):

$$\text{begin } I_1; I_2; \dots; I_n \text{ end} \quad (7.3)$$

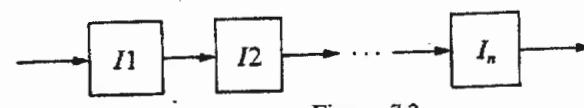


Figura 7.2.

Il punto e virgola è l'operatore di successione: indica che l'istruzione a esso successiva va eseguita dopo che è terminata la precedente. I simboli-base **begin** ed **end** fungono da *parentesi per le istruzioni*. Infatti, poiché le sequenze di istruzioni possono assumere una lunghezza considerevole, è utile usare delle parentesi appaiate per porre in risalto i raggruppamenti interni a una sequenza di istruzioni.

2) Istruzioni condizionali (v. fig. 7.3 a) e b).

$$\text{if } D \text{ then } I_1 \text{ else } I_2 \quad \text{if } D \text{ then } I \quad (7.4)$$

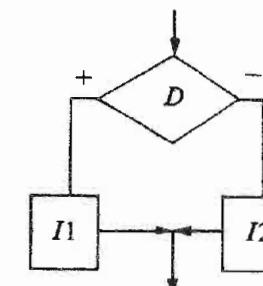


Figura 7.3.(a).

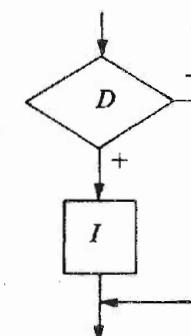


Figura 7.3.(b).

La seconda forma può essere considerata come una scrittura abbreviata della prima, dove manca l'alternativa I_2 .

L'uso delle parentesi **begin** e **end** diventa significativo nel caso delle istruzioni condizionali; per esempio, la sequenza di istruzioni

$$\text{if } D \text{ then } I_1; I_2 \quad (7.5)$$

ha lo stesso significato di

$$\text{begin if } D \text{ then } I_1 \text{ end; } I_2 \quad (7.6)$$

ma non ha lo stesso significato di

$$\text{if } D \text{ then begin } I_1; I_2 \text{ end} \quad (7.7)$$

3) Istruzioni ripetitive (v. fig. 7.4):

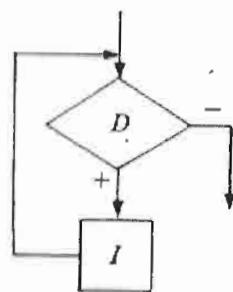
while D **do** I 

Figura 7.4(a).

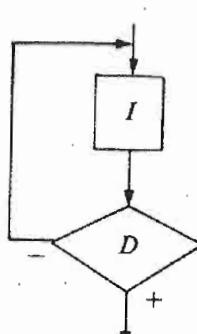
repeat I **until** D 

Figura 7.4(b).

Osservazione: poiché nel secondo caso i simboli-base **repeat** e **until** sono già una coppia di parentesi, è lecito usare direttamente la forma

repeat $I_1; I_2; \dots; I_n$ **until** D senza dover introdurre una coppia **begin-end**.

4) Istruzioni di selezione (v. fig. 7.5):

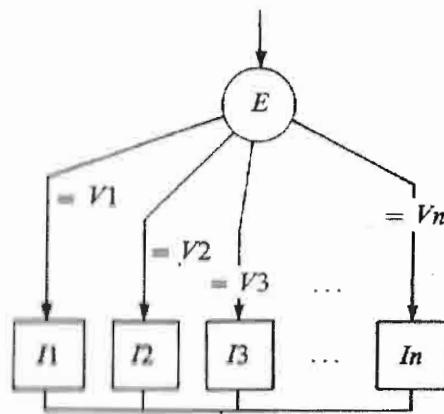
case E **of** $V_1:I_1; V_2:I_2; \dots; V_n:I_n$ **end**

Figura 7.5.

Se l'espressione E possiede il valore V_k , viene scelta ed eseguita l'istruzione I_k ($V_i \neq V_j$ per $i \neq j$).

Nota: quando $I_1 = I_2 = \dots = I_k = I$, useremo la notazione abbreviata: $V_i, V_j, \dots, V_k : I$.

Come già negli schemi di flusso, è facile inserire commenti, condizioni di verifica, ecc., nei testi lineari dei programmi. Basta infatti usare la seguente convenzione: le parti comprese fra parentesi graffe **{and}**, rappresentano dei commenti e non hanno effetto sull'esecuzione del programma; esse vengono ignorate dal processor. Con questa convenzione, è possibile formulare in una notazione lineare anche le regole di verifica relative alle strutture fondamentali.

La forma usuale è:

$$\{P\} I \{C\}$$

ove P è la premessa, I è l'istruzione e C la conseguenza.

7.2.2. Regole di verifica in notazione lineare.

1. Assegnamento $\{P(w)\} v := w \{P(v)\}$ (7.11)

2. Sequenza di istruzioni:
premesse: $\{P\} I_1 \{C\}$
 $\{C\} I_2 \{R\}$
conseguenza: $\{P\} I_1; I_2 \{R\}$ (7.12)

3. Istruzioni condizionali:
premesse $\{P \wedge D\} I_1 \{C\}$
 $\{P \wedge \neg D\} I_2 \{C\}$
conseguenza: $\{P\} \text{ if } D \text{ then } I_1 \text{ else } I_2 \{C\}$ (7.13)

premesse: $\{P \wedge D\} I \{C\}$
 $\{P \wedge \neg D\} \supset C$
conseguenza: $\{P\} \text{ if } D \text{ then } I \{C\}$ (7.14)

4. Istruzioni ripetute:
premesse: $\{P \wedge D\} I \{P\}$
conseguenza: $\{P\} \text{ while } D \text{ do } I \{P \wedge \neg D\}$ (7.15)

premesse: $\{P\} I \{C\}$
 $\{C \wedge \neg D\} I \{C\}$
conseguenza: $\{P\} \text{ repeat } I \text{ until } D \{C\}$ (7.16)

5. Istruzione di selezione:

premesse: $\{P \wedge (E = V_K)\} I_K \{C\}$ per ogni K
conseguenza: $\{P\}$ case E of
 $V_1:I_1;$ (7.17)
 $V_2:I_2;$
.....
 $V_n:I_n$ end $\{C\}$

7.3. Semplici esempi di programmi in notazione lineare

È possibile fin d'ora usare le notazioni introdotte per scrivere in forma lineare i programmi esposti nei capitoli precedenti:

1) Moltiplicazione di due numeri naturali x e y (v. fig. 5.17):

```
begin z := 0; u := x;
repeat {z + u * y = x * y, u > 0}
      z := z + y; u := u - 1
until u = 0
end
```

(7.18)
2) Divisione intera di due numeri naturali x e y (v. fig. 5.18):

```
begin q := 0; r := x;
while r ≥ y do
  begin {q * y + r = x, r ≥ y}
    r := r - y; q := q + 1
  end
end
```

(7.19)
3) Moltiplicazione di due numeri naturali x e y (v. fig. 5.19):

```
begin z := 0; u := x; v := y;
while u ≠ 0 do
  begin {z + u * v = x * y, u > 0}
    if odd(u) then z := z + v;
    u := u div 2; v := 2 * v
  end
  {z = x * y}
end
```

(7.20)
4) Calcolo del massimo comun divisore di due numeri naturali x e y (v. fig. 5.20):

```
begin a := x; b := y;
while a ≠ b do
  if a > b then a := a - b else b := b - a
  {MCD(a, b) = MCD(x, y)}
  {MCD(x, y) = a = b}
end
```

(7.21)

Si osservi che la struttura dei programmi ne facilita la lettura. Ciò è dovuto alle particolari convenzioni di scrittura, per cui si vengono a trovare sulla stessa colonna le istruzioni appartenenti a uno stesso gruppo, successivo a una certa condizione o a una certa clausola di ripetizione. In particolare, per ogni coppia di parentesi begin-end, il begin e l'end corrispondente compaiono in evidenza sulla stessa colonna.

I programmi (7.18) e (7.20) calcolano il medesimo risultato. Però essi differiscono per il tempo di calcolo richiesto. Questo può essere determinato contando il numero delle operazioni da eseguire. Il programma (7.18) richiede due addizioni e due assegnamenti per ogni ripetizione. Indicando con z il tempo richiesto per un assegnamento, e con a quello richiesto per una addizione, il tempo complessivo richiesto dal programma (7.18) per eseguire il prodotto $x * y$ è:

$$2z + 2(z + a) * x \quad (7.22)$$

Invece, nel processo descritto dal programma (7.20), oltre a una moltiplicazione per 2, a una divisione per 2 e un confronto per stabilire se n è dispari, per ogni ripetizione vengono eseguiti, a seconda dei casi, o due assegnamenti (n pari), o tre assegnamenti e un'addizione (n dispari). La presenza di due casi distinti non permette di valutare in generale il tempo di calcolo esatto; è possibile valutare il tempo minimo e il tempo massimo. È più importante osservare che, a ogni ripetizione, n viene diviso per due (senza contare il resto) e che perciò vengono eseguite al più $\log_2 x$ ripetizioni. Quindi il tempo richiesto dal programma (7.20) per eseguire una moltiplicazione è, al più:

$$3z + \log_2 x (3z + a + h) \quad (7.23)$$

dove h indica il tempo complessivo richiesto dalla moltiplicazione per 2, dalla divisione per 2 e dal confronto n dispari.

Poiché la rappresentazione interna dei numeri è la rappresentazione binaria, i calcolatori eseguono con la massima rapidità le moltiplicazioni per 2, le divisioni per 2 ed i confronti di parità; perciò il programma (7.20) è migliore del programma (7.18) anche per piccoli valori di x , e quindi è sicuramente preferibile.

Il seguente programma (7.24) costituisce un miglioramento del programma *divisione* (7.19), analogo al precedente. Analizzando gli invarianti del ciclo, è possibile verificare facilmente la correttezza dell'algoritmo e coglierne il funzionamento q , r e w sono variabili con valori interi.

Programma per dividere il numero naturale x per y :

```
begin r := x; q := 0; w := y; (7.24)
  while w ≤ r do w := 2 * w;
  {w = 2n * y > x}
  while w ≠ y do
    begin {q * w + r = x, r ≥ 0}
      q := 2 * q; w := w div 2;
      if w ≤ r then
        begin r := r - w; q := q + 1
        end
      end
    end
  {q * y + r = x, 0 ≤ r < w; q = x div y}
end.
```

La riduzione del tempo di calcolo è ancora dovuta al fatto che la ripetizione viene eseguita solo $\log_2(x/y)$ volte, anziché x/y volte.

Sostituendo le sottrazioni ripetute con una divisione, anche il programma illustrato nella figura 6.2, che calcola il massimo comun divisore di due numeri, può essere modificato in una versione più efficiente. Per semplificare la notazione, introduciamo l'operatore **mod** (modulo) che fornisce il resto della divisione intera di due numeri naturali $x \geq 0$ e $y > 0$; cioè:

$$q = x \text{ div } y \quad (7.25)$$

indica il quoziente, e

$$r = x \text{ mod } y \quad (7.26)$$

indica il resto. Perciò vale sempre l'eguaglianza:

$$x = (x \text{ div } y) * y + (x \text{ mod } y) \quad (7.27)$$

La sottrazione ripetuta

$$\text{while } a \geq b \text{ do } a := a - b \quad (7.28)$$

può essere quindi sostituita dall'operazione

$$a := a \bmod b \quad (7.29)$$

in seguito alla quale $a < b$.

Il programma della figura 6.2, scritto in notazione lineare, è:

```
begin a := x; b := y;
repeat {a > 0, b > 0}
  while a > b do a := a - b;
  while b > a do b := b - a;
  until a = b
  {a = b = MCD(x, y)}
end. (7.30)
```

e la corrisponde versione senza la ripetizione della sottrazione è:

```
begin a := x; b := y;
repeat {a > 0, b > 0}
  if a ≥ b then a := a mod b;
  {0 ≤ a < b}
  if a > 0 then b := b mod a else Exchange(a, b)
  until b = 0
  {a = MCD(x, y)}
end. (7.31)
```

Questa versione del calcolo del massimo comun divisore fu inventata da *Euclide* ed è uno dei primi esempi noti di algoritmo. Esso viene per lo più riportato nella forma ancor più semplice:

```
begin a := x; b := y;
repeat a := a mod b; Exchange(a, b)
until b = 0
{a = MCD(x, y)}
end. (7.32)
```

La relazione

$$x > y > 0: MCD(x, y) = MCD(x \bmod y, x) \quad (7.33)$$

è adeguata per eseguire la verifica.

7.4. Problemi

7.1. Stabilire per mezzo dei diagrammi sintattici (appendice A) quali delle sequenze dei simboli sottoelencate possono rappresentare dei numeri, delle costanti, delle variabili, dei fattori, delle espressioni o delle istruzioni. Fare attenzione alla classificazione degli operatori in tre classi:

operatori di relazione: $= \neq < \leq \geq >$

operatori additivi: $+ - \vee$

operatori moltiplicativi: $*$ / div mod \wedge

dove gli operatori di relazione appartengono alla classe di precedenza più debole.

Numeri	0.31	$+ 237.2$	3.5	$- 0.005$
	4.555	$3 + 5$	$3E5$	two
	33,75	.389	1E00	15
	$10E - 4$	$1.5 + 2$	00037	3,250
Costanti	100	true	+ 15.5	red
	'A'	nine	9/5	'*'
Variabili	x	$A[i]$	$x + y$	$B[i, j]$
Fattori	$B[i, j]$	$\sin(x)$	p	$p \vee q$
	(x)	$x * y$	$x - y$	$\exp(y * \ln(x))$
Termini	(x)	$x * y$	$x - y$	$(x - y)$
Espressioni	x	2	$a = b$	$+ x * y$
	(x)	$(x \leq y) \wedge (y < z)$	$p < q \wedge r < s$	true
Istruzioni				

```

a := b      a := 2      2 := a sin(x * y)
begin a := 1 end
if a = 2 then a := a + 1 else P(x, y)
while a > 0 do a := a - 1 end
if x < y then; z := true; else z := false
repeat z := z + 1.5, y := u - 1 until y = 0

```

7.2. Calcolare le seguenti espressioni:

$$\begin{aligned}
2 * 3 - 4 * 5 &= \\
15 \text{ div } 4 * 4 &= \\
80/5/3 &= \\
2 / 3 * 2 &=
\end{aligned}$$

Scrivere inoltre le seguenti espressioni nel linguaggio definito nell'appendice A:

$$a^2 - c + \frac{a}{b * c + \frac{c}{d + \frac{e}{f}}}$$

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\frac{1}{2} * \ln \left| \frac{w - a}{w + a} \right|$$

$$\frac{\frac{1}{a} + \frac{1}{b}}{c + d}$$

7.3. Eseguire i programmi sotto indicati per i valori di x e di y riportati; scrivere le corrispondenti tabelle di traccia:

1) programmi (7.18) e (7.20):

$$(x, y) = (3, 5), (2, 11), (10, 8), (19, 2)$$

2) programmi (7.19) e (7.24):

$$(x, y) = (83.15), (117.9), (23.27), (1191.37)$$

3) programmi (7.21), (7.30) e (7.32):

$$(x, y) = (84.36), (36.84), (770.441), (15.15)$$

7.4. Determinare per il programma (7.24) il numero massimo e il numero minimo di operazioni da eseguire in funzione di x e di y e stabilire le condizioni di verifica necessarie e sufficienti dopo ogni istruzione.

7.5. Tradurre in notazione lineare lo schema di flusso della figura 7.6, che calcola $MCD(x, y)$ e, contemporaneamente, due fattori c e d tali che:

$$c * x + d * y = MCD(x, y)$$

Dare le condizioni di verifica necessarie e sufficienti.

7.6. Scrivere un programma corredata delle necessarie condizioni di verifica, il quale calcoli il massimo comun divisore di due numeri $MCD(x, y)$ in base alle seguenti relazioni:

- 1) (a) $MCD(2 * m, 2 * n) = 2 * MCD(m, n)$
- 2) (b) $odd(n): MCD(2 * m, n) = MCD(m, n)$
- 3) (c) $m > n: MCD(m - n, n) = MCD(m, n)$
- 4) (d) $MCD(n, m) = MCD(m, n)$
- 5) (e) $odd(m) \wedge odd(n): \neg odd(m - n)$

Nel programma sono possibili solo le operazioni di sottrazione, di confronto, di moltiplicazione e di divisione per 2.

7.7. Si supponga che possa valere, di volta in volta, solo una delle condizioni P, Q, R . Si indichi con W_P la probabilità che P sia soddisfatta (W_Q e W_R avranno analogo significato). Stabilire in funzione di W_P , di W_Q e di W_R il valore di aspettazione del numero di esecuzioni di P , di Q e di R nelle seguenti tre istruzioni:

- 1) (a) if P then A else if Q then B else C
- 2) (b) if Q then B else if R then C else A
- 3) (c) if R then C else if P then A else B

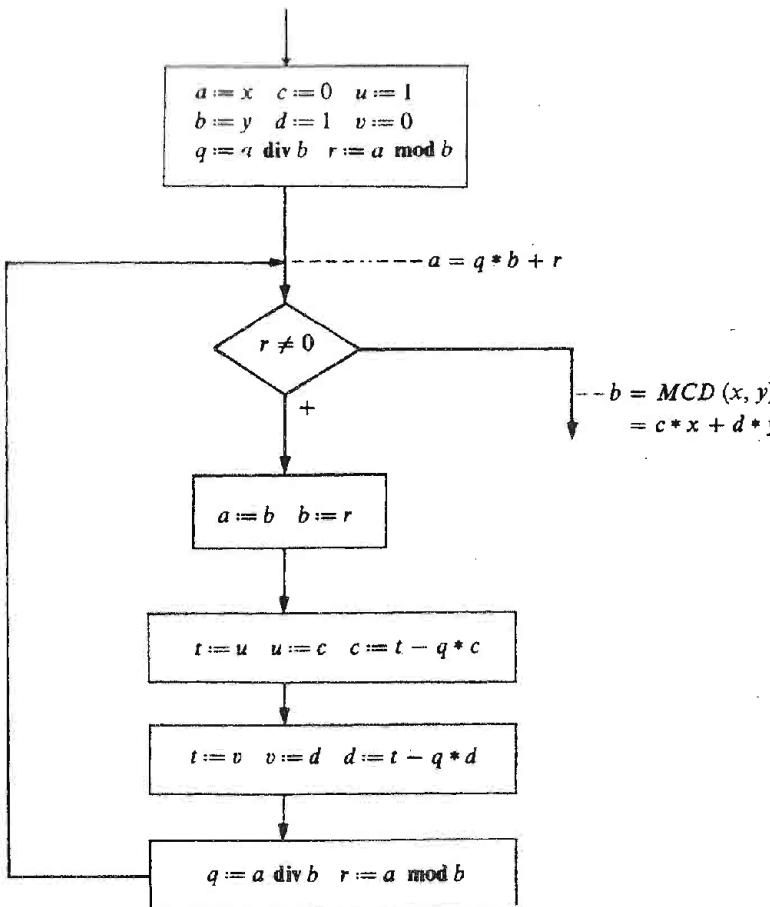


Figura 7.6.

Quale delle tre istruzioni (equivalenti) conviene scegliere se $W_P > W_Q \geq W_R$?

Esempio: $P = (x > y)$, $Q = (x = y)$, $R = (x < y)$;
 $W_P = 0.5$, $W_Q = 0.3$, $W_R = 0.2$

Generalizzare la regola ricavata al caso di n condizioni mutuamente esclusive.

Come già accennato nel capitolo 5, l'elenco di tutte le variabili, posto all'inizio di un programma, ne facilita la lettura e la comprensione; in particolare, ogni buona documentazione è basata sull'indicazione esplicita dei valori *permessi* per ogni variabile. Tale affermazione è giustificata da validi argomenti; i principali sono:

- 1) la conoscenza dei valori permessi per le variabili è indispensabile per capire il funzionamento di un algoritmo; infatti, in assenza di una esplicita indicazione è estremamente difficile stabilire il tipo di oggetti rappresentati da una variabile e individuare eventuali errori;
- 2) l'adeguatezza e la correttezza di un programma dipendono dai valori iniziali degli argomenti; nella maggior parte dei casi sono garantite solo per valori compresi entro determinati intervalli, che ogni buona documentazione deve indicare esplicitamente;
- 3) lo spazio di memoria necessario per rappresentare una variabile dipende dai valori che questa può assumere; è quindi necessario indicare esplicitamente il campo di tali valori, affinché il compilatore possa riservare lo spazio di memoria necessario;
- 4) gli operatori che compaiono nelle espressioni sono definiti e operano correttamente solo se i valori dei loro argomenti appartengono a ben precisi intervalli; la specificazione di tali intervalli permette al compilatore di stabilire se una combinazione di operatori e di operandi è lecita e costituisce quindi una *ridondanza* utile per eseguire un controllo del programma già in fase di compilazione;
- 5) gli operatori sono *realizzati* da opportuni programmi (scritti nel codice della macchina) che, in generale, dipendono dal campo dei valori permessi per gli argomenti: per esempio, nella maggior parte dei calcolatori, la rappresentazione interna dei numeri interi è diversa da quella dei numeri reali e completamente diversa è,

nei due casi, anche la successione delle azioni *di macchina* necessarie per eseguire le operazioni aritmetiche.

Per il particolare e importante ruolo che l'insieme dei valori permessi gioca nella caratterizzazione di una variabile, tale insieme viene chiamato il *tipo* della variabile. Gli elementi appartenenti all'insieme sono detti *costanti* del tipo considerato.

D'ora in poi ci atterremo rigidamente alla seguente convenzione: *tutte le variabili devono essere indicate* in una lista, contenente i nomi scelti per indicare le variabili e i tipi a esse corrispondenti.

Una *indicazione di variabile* è della forma:

$$\text{var } v : T \quad (8.1)$$

dove v è un nome variabile e T è il *tipo* della variabile indicata da v . Nel caso vi siano più variabili dello stesso tipo da indicare, si usa la forma abbreviata:

$$\text{var } v_1, v_2, \dots, v_m : T \quad (8.2)$$

dove v_1, v_2, \dots, v_n sono nomi di variabili e T è il nome o la descrizione di un tipo.

La convenzione di indicare all'inizio di ogni programma tutti i nomi usati per le variabili, presenta l'immediato vantaggio che un compilatore è in grado di stabilire se un qualsiasi nome incontrato nel programma è stato preliminarmente indicato. Se ciò non si verifica per qualche nome (per esempio per un errore nello scrivere un nome), il compilatore è in grado di segnalare il *lapses* con un messaggio di errore. Questa ridondanza serve, così, per rendere più sicura la programmazione.

Come si introducono i tipi dei dati in un programma, e quali campi si possono rappresentare adeguatamente in un calcolatore? Innanzitutto va ricordato che, di solito, i tipi dei dati vengono classificati in diverse classi. La caratteristica essenziale di un tipo è la struttura dei suoi valori. Se un valore non è strutturato, cioè se non può essere decomposto in singole componenti, esso — e il tipo a cui appartiene — viene chiamato *scalare*. In questo capitolo vengono introdotti solo tipi scalari, mentre i *tipi strutturati* verranno trattati nei capitolo 10 e 11. La forma generale della *indicazione di un tipo* è:

$$\text{type } t = T \quad (8.3)$$

dove t è il nome introdotto per indicare il tipo e T ne è la descrizione.

Un *tipo scalare* è descritto dall'elenco dei suoi valori, per mezzo della seguente scrittura:

$$\text{type } t = (w_1, w_2, \dots, w_n) \quad (8.4)$$

Con una indicazione siffatta vengono introdotti sia il nome t , che indica il tipo, sia i nomi w_1, w_2, \dots, w_n che indicano le costanti. Esempi di definizioni di tipi scalari sono:

$$\begin{aligned} \text{type } colore &= (\text{rosso}, \text{giallo}, \text{verde}, \text{blu}) \\ \text{type } cose &= (\text{rosa}, \text{ghianda}, \text{cuore}, \text{scudo}) \\ \text{type } forma &= (\text{triangolare}, \text{rettangolare}, \text{circolare}) \\ \text{type } stato &= (\text{solido}, \text{liquido}, \text{aeriforme}) \end{aligned}$$

Osservazione: se per certe variabili è superfluo assegnare un nome specifico al corrispondente tipo, è possibile combinare le dichiarazioni (8.4) e (8.2) nell'unica indicazione:

$$\text{var } v_1, v_2, \dots, v_m : (w_1, w_2, \dots, w_n) \quad (8.5)$$

D'ora in poi prenderemo in considerazione soltanto insiemi di valori scalari definiti fra loro e ordinati. Per ogni tipo t definito dalla (8.4), devono valere i seguenti assiomi:

- 1) $w_i \neq w_j$ per $i \neq j$ (unicità);
- 2) $w_i < w_j$ per $i < j$ (ordinamento);
- 3) solo i valori w_i ($i = 1, \dots, n$) appartengono al tipo t .

È inoltre utile introdurre le funzioni *successore* e *predecessore*:

$$\begin{aligned} \text{succ } (w_i) &= w_{i+1} \quad \text{per } 1 \leq i < n \\ \text{pred } (w_i) &= w_{i-1} \quad \text{per } 1 < i \leq n \end{aligned} \quad (8.7)$$

Ogniqualvolta dovremo definire più tipi scalari, ci atterremo alla seguente regola: ogni nome di costante va usato una volta sola, in modo che da ogni nome si possa risalire senza ambiguità al valore e al tipo corrispondenti. Devono quindi essere evitate definizioni ambigue, come, per esempio, la seguente:

$$\begin{aligned} \text{type } colore \text{ caldo} &= (\text{rosso}, \text{giallo}, \text{verde}) \\ \text{colore freddo} &= (\text{verde}, \text{blu}, \text{violetto}) \end{aligned} \quad (8.8)$$

in quanto il colore *verde* è definito come appartenente a due tipi diversi.

Definiremo ora quattro tipi scalari, usati assai di frequente e considerati come tipi fondamentali in ogni sistema di calcolo. Essi sono costituiti dai campi: 1) dei valori di verità; 2) dei numeri interi; 3) dei numeri reali; 4) dei caratteri di stampa. Tali campi sono chiamati *tipi standard*; data la loro importanza, in ogni linguaggio di programmazione le loro costanti (eccetto i valori di verità) sono indicate per mezzo di costruzioni sintattiche ben precise, che ora illustreremo.

8.1. Il tipo Boolean

Il tipo *Boolean* indica il campo dei valori di verità, formato dai due elementi *true* e *false*. Il nome *booleano* deriva dal fondatore del calcolo logico, George Boole (1815-1864). Si usa la scrittura:

$$\text{type Boolean} = (\text{false}, \text{true}) \quad (8.9)$$

Sui valori del tipo *Boolean* sono definiti i seguenti operatori:

O logico (disgiunzione inclusiva) (*OR*)

E logico (congiunzione) (*AND*)

NON logico (negazione) (*NOT*)

Dati gli argomenti booleani p e q , i valori delle espressioni $p \vee q$, $p \wedge q$ e $\neg p$ sono definiti nella tabella 8.I.

TABELLA 8.I.

p	q	$p \vee q$	$p \wedge q$	$\neg p$
falso	falso	falso	falso	vero
vero	falso	vero	falso	falso
falso	vero	vero	falso	vero
vero	vero	vero	vero	falso

Si osservi che dalla definizione si possono derivare le seguenti relazioni, spesso utili:

$$1. p \vee q = q \vee p \quad \text{leggi commutativa} \quad (8.10)$$

$$p \wedge q = q \wedge p$$

$$2. (p \vee q) \vee r = p \vee (q \vee r) \quad \text{leggi associativa} \quad (8.11)$$

$$(p \wedge q) \wedge r = p \wedge (q \wedge r)$$

$$3. (p \wedge q) \vee r = (p \vee r) \wedge (q \vee r) \quad \text{leggi distributiva} \quad (8.12)$$

$$(p \vee q) \wedge r = (p \wedge r) \vee (q \wedge r)$$

$$4. \neg(p \vee q) = \neg p \wedge \neg q \quad \text{leggi di de Morgan} \quad (8.13)$$

$$\neg(p \wedge q) = \neg p \vee \neg q$$

L'operatore \vee appartiene alla classe di precedenza (v. cap. 7) più debole, l'operatore \wedge a quella intermedia e l'operatore \neg a quella più forte. Per esempio, $\neg p \vee q \wedge r$ significa $(\neg p) \vee (q \wedge r)$.

Tutti gli operatori indicanti relazioni forniscono un risultato di tipo *Boolean*. Per esempio, l'espressione $x = y$ fornisce il valore booleano *true* se $x = y$, mentre fornisce il valore *false* in tutti gli altri casi. Gli operatori di relazione più largamente usati sono $=$, \neq , \leq , \geq , $<$ e $>$; gli ultimi quattro possono essere applicati solo a valori scalari (per i quali è definito un ordinamento, in base agli assiomi). Valgono inoltre le relazioni:

$$\begin{aligned} x \neq y &\leftrightarrow \neg(x = y) \\ x \leq y &\leftrightarrow (x < y) \vee (x = y) \\ x \geq y &\leftrightarrow \neg(x < y) \\ x > y &\leftrightarrow \neg(x < y) \wedge \neg(x = y) \end{aligned} \quad (8.14)$$

I sei operatori di relazione riportati, quindi, possono venir descritti per mezzo di due di essi e degli operatori logici.

8.2. Il tipo Integer

Il tipo *integer* indica il campo dei numeri interi. Ivi, sono definiti i seguenti operatori:

- + addizione
- sottrazione
- * moltiplicazione
- div divisione intera
- mod resto della divisione intera

In ogni sistema di calcolo automatico, il tipo *integer* specifica un intervallo di numeri interi: più precisamente, specifica l'insieme di tutti gli interi di valore assoluto minore di un certo valore-limite *max*. Ne segue che la maggior parte degli assiomi dell'aritmetica non valgono più nel caso generale. Infatti, indicando per esempio con \oplus l'addizione eseguita da un calcolatore in cui il massimo numero sia *max*, vale

$$x \oplus y = x + y$$

solo se $|x + y| < max$. Quindi, la legge associativa dell'addizione

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

vale solo se $|x + y| < max$ e $|y + z| < max$. Se, per esempio, $max = 100$, vale

$$60 \oplus (50 \oplus (-40)) = 60 \oplus 10 = 70$$

mentre

$$(60 \oplus 50) \oplus (-40)$$

ha valore indefinito, poiché $60 + 50 = 110 > max$. Mentre a prima vista, tale situazione può sembrare insanabile, nell'uso pratico dei calcolatori essa non ha gravi conseguenze; infatti l'aritmetica di un normale calcolatore opera su intervalli di numeri interi talmente ampi, che assai raramente si possono verificare casi analoghi al precedente, dove un risultato intermedio ha valore indefinito. Comunque, una norma minimale di sicurezza è che il calcolo venga interrotto non appena qualche risultato intermedio sia indefinito. Se non vi sono interruzioni di tal fatta, tutte le *operazioni su grandezze di tipo integer* forniscono sicuramente risultati esatti. Se invece un risultato intermedio supera il valore max si ha un'interruzione; tale situazione viene indicata con il nome di *overflow*.

In molti casi si sa a priori che i valori di certe variabili intere rimangono sempre interni a un dato intervallo I . Quest'informazione può essere resa esplicita con una dichiarazione del seguente *tipo di sottocampo*:

type $I = min \dots max$ (8.15)

dove min e max sono gli estremi dell'intervallo. L'indicazione esplicita degli intervalli può, fra l'altro, aumentare in modo essenziale la trasparenza di un programma. Il tipo *integer* stesso, è un intervallo i cui estremi dipendono dal particolare calcolatore usato.

8.3. Il tipo Char

Il tipo *char* indica un campo finito e ordinato di *simboli* (caratteri). Ogni calcolatore dispone di un ben preciso insieme di simboli, per mezzo dei quali comunica con l'esterno attraverso i dispositivi di ingresso-uscita. I simboli sono quelli usati dai dispositivi di let-

tura e di stampa. Una certa *standardizzazione dei simboli* è necessaria, se si desidera che differenti tipi di calcolatore possano comunicare direttamente fra loro (trasmissione dei dati, elaborazione remota, ecc.). Generalmente, si conviene che i simboli usati dai calcolatori comprendano le *lettere dell'alfabeto*, le *cifre decimali* e un certo numero di *simboli speciali*. Il più diffuso è l'insieme standard di simboli deciso dalla International Standards Organization (ISO); in particolare si è affermata la versione americana ASCII (American Standard Code for Information Interchange).

L'ASCII è un insieme di 128 simboli. Poiché $128 = 2^7$, è possibile rappresentare ogni simbolo con una combinazione di 7 cifre binarie. Una corrispondenza fra simboli e combinazioni di cifre viene chiamata *codice*; il codice ASCII è un così detto *codice di 7 cifre* (v. app. B).

Nell'ASCII si fa distinzione fra *caratteri di stampa* e *simboli di controllo*. I caratteri di stampa si suddividono in lettere maiuscole, lettere minuscole, cifre e caratteri speciali. Un'ulteriore distinzione è quella fra l'*insieme completo dei simboli ASCII* e quello *incompleto*. Quest'ultimo (colonne 0-5 della tabella 8.II) non contiene le lettere minuscole ed è di uso corrente su molti calcolatori di tipo commerciale.

TABELLA 8.II. Caratteri ASCII

	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	p	
1	soh	dcl	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	\$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(8	H	X	h	x
9	ht	em)	9	I	Y	i	y
10	lf	sub	*	:	J	Z	j	z
11	vt	esc	+	:	K	[k	{
12	ff	fs	,	<	L	\	l	
13	cr	qs	-	=	M]	m	}
14	so	rs	.	>	N	~	n	~
15	si	us	/	?	O	_	o	del

Il significato dei simboli di controllo è descritto nell'appendice B. Riportiamo qui solo due simboli di controllo: **cr** (*carriage return*) e **lf** (*line feed*). Essi controllano il ritorno del carrello e l'interlinea per la stampante.

Accanto al tipo *char*, introdurremo due funzioni standard, dipendenti dall'ordinamento dei simboli in questione. Esse infatti stabiliscono una corrispondenza biunivoca fra l'insieme dei simboli (il tipo *char*) e un sottoinsieme degli interi (del tipo *integer*). Sono chiamate *funzioni di trasferimento*, poiché permettono di rappresentare le costanti di un tipo con costanti di tipo diverso, e sono definite nel modo seguente:

$ord(c)$ è il numero d'ordine del simbolo c nell'ordinamento dei simboli (nella tabella 8.II, se x e y sono le coordinate del simbolo c , $ord(c) = 16 \cdot x + y$);
 $chr(i)$, analogamente, è il simbolo il cui numero d'ordine è i .

Vale dunque che:

$$chr(ord(c)) = c \quad \text{and} \quad ord(chr(i)) = i \quad (8.16)$$

e la relazione d'ordine tra i simboli è la seguente:

$$c_1 < c_2 \iff ord(c_1) < ord(c_2) \quad (8.17)$$

Per poter indicare che un simbolo rappresenta una costante di tipo *char*, è necessario introdurre una convenzione per identificare le costanti: tale convenzione è di racchiudere fra due apostrofi i simboli che denotano delle costanti. Per esempio, se si vuole assegnare il valore *punto di domanda* alla variabile x , si usa l'istruzione

$$x := '?'$$

Nei capitoli successivi non si farà riferimento a un insieme di simboli particolari. Sarà sufficiente che valgano le seguenti condizioni:

- 1) l'insieme dei simboli usati contiene le lettere dell'alfabeto e le cifre;
- 2) i sottoinsiemi delle lettere e delle cifre sono ordinati e coerenti, cioè:

$$\begin{aligned} c \text{ è una lettera} &\rightarrow ('A' \leq c) \wedge (c \leq 'Z') \\ c \text{ è una cifra} &\rightarrow ('0' \leq c) \wedge (c \leq '9'); \end{aligned} \quad (8.18)$$

3) l'insieme dei simboli contiene il simbolo *bianco* (spaziatura) il separatore di linea (eol significante *end of line*) e alcuni simboli speciali quali il punto, il punto e virgola, la virgola e altri che non staremo qui a elencare.

8.4. Il tipo Real

Il fatto che in un calcolatore è possibile rappresentare solo insiemi finiti di valori, pone un problema particolarmente grave per quanto riguarda i calcoli sui numeri reali. Mentre, per i numeri interi le operazioni aritmetiche portano a risultati esatti, purché essi non superino un certo valore limite, la esattezza non è più possibile per i numeri reali.

Infatti, qualsiasi intervallo reale, per quanto piccolo, contiene infiniti valori (come si suol dire, i numeri reali formano un *continuo*), mentre il tipo *real* della *aritmetica di un calcolatore* contiene solo un numero finito di valori; ciascuno di essi *rappresenta* un intervallo del continuo. La sostituzione di un numero x con il valore \tilde{x} , *rappresentante* dell'intervallo cui x appartiene, ha conseguenze che dipendono dal problema e dell'algoritmo in questione. La stima dell'errore che si commette sostituendo il continuo dei reali con un insieme finito di *rappresentanti* è campo d'indagine specifico del calcolo numerico e pone difficili problemi di approssimazione. I processi che elaborano dati di tipo *real* son detti *numerici*, dove *numerico* è sinonimo di *non esatto*.

Un qualsiasi calcolo numerico sarebbe privo di senso, qualora non si avesse un'idea del tipo e dell'entità degli errori che si possono commettere. Per questo è necessario conoscere la *rappresentazione finita* (cioè con un numero finito di cifre) usata per rappresentare i numeri reali nel calcolatore.

È ormai divenuta di uso comune negli attuali elaboratori, la così detta *rappresentazione in virgola mobile*, dove un numero reale x è rappresentato da due numeri interi e ed m , compresi in un intervallo finito, nel seguente modo:

$$x = m \cdot B^e, \quad -E < e < E, \quad -M < m < M \quad (8.19)$$

I valori di B , di E e di M variano da calcolatore a calcolatore. B è chiamato *base* della rappresentazione, ed è per lo più una potenza di due. Uno stesso valore x può essere rappresentato da più coppie di interi. Una forma canonica (o forma normale) si ottiene dalla condizione aggiuntiva

$$\frac{1}{B} \leq m < 1. \quad (8.20)$$

Usando la rappresentazione canonica, la densità dei rappresentanti per intervallo decresce con un andamento esponenziale. Per esempio, se $B = 10$, il numero dei rappresentanti dell'intervallo $0,1 \div 1$ è uguale al numero dei rappresentanti dell'intervallo $10\,000 \div 100\,000$. La scelta dei valori di B , di E e di M e la corrispondente distribuzione non uniforme dei rappresentanti hanno conseguenze non facilmente valutabili e le operazioni elementari relative ai valori di tipo *real* non possono essere esatte; tuttavia, esse devono soddisfare un insieme di condizioni minimali, qualunque sia l'*aritmetica* usata.

Una formulazione generale di tali condizioni è fornita dagli assiomi seguenti A1, A2, A3, ..., A9.

A1. Il tipo *real* (indicato con R) è un sottoinsieme finito dell'insieme \mathbb{R} dei numeri reali

$$R \subset \mathbb{R}$$

A2. A ogni numero $x \in R$ è associato un unico numero $\tilde{x} \in R$; \tilde{x} è chiamato il *rappresentante* di x .

A3. Ogni $\tilde{x} \in R$ rappresenta molti $x \in R$, ma l'intervallo che \tilde{x} rappresenta è *coerente* (cioè, se $x_1 < x_2$, $\tilde{x}_1 = r$ e $\tilde{x}_2 = r$, allora $\tilde{x} = r$ per ogni $x_1 \leq x \leq x_2$). Inoltre vale

$$x \in R \Rightarrow \tilde{x} = x;$$

in particolare, 0 e 1 devono essere rappresentati esattamente; cioè dev'essere $0 \in R$, $1 \in R$, e quindi $\tilde{0} = 0$ e $\tilde{1} = 1$.

A4. Vi è un valore massimo *max*, tale che $x = \Omega$ (Ω indica il valore *indefinito*) per tutti gli $|x| > max$. L'insieme dei numeri reali $|x| > max$ viene detto *insieme di overflow* e verrà indicato con O . $R - O$ è coerente.

Dagli assiomi precedenti segue immediatamente che, per $x, y \in R - O$:

$$\begin{aligned} x < y &\Rightarrow \tilde{x} \leq \tilde{y} \\ x = y &\Rightarrow \tilde{x} = \tilde{y} \\ x > y &\Rightarrow \tilde{x} \geq \tilde{y} \end{aligned} \quad (8.21)$$

A5. R è simmetrico rispetto allo 0, cioè:

$$(-x)^\sim = -(\tilde{x}) \quad (8.22)$$

Esporremo ora le proprietà alle quali deve soddisfare l'aritmetica di un calcolatore per essere impiegata nei calcoli numerici. Gli

operatori aritmetici base sono l'addizione, la sottrazione, la moltiplicazione e la divisione, indicate rispettivamente con i simboli $\oplus, \ominus, \otimes, \oslash$. Assumeremo sempre $x, y \in R$.

A6. *Commutatività* dell'addizione e della moltiplicazione:

$$x \oplus y = y \oplus x, \quad x \otimes y = y \otimes x \quad (8.23)$$

$$x \geq y \geq 0 \rightarrow (x \ominus y) \oplus y = x \quad (8.24)$$

A7. *Simmetria* rispetto allo 0 delle operazioni base:

$$x \ominus y = x \oplus (-y) = -(y \ominus x) \quad (8.25)$$

$$(-x) \otimes y = x \otimes (-y) = -(x \otimes y) \quad (8.25)$$

$$(-x) \oslash y = x \oslash (-y) = -(x \oslash y) \quad (8.25)$$

A9. *Monotonicità* delle operazioni base:

$$0 \leq x \leq a \text{ and } 0 \leq y \leq b$$

$$x \oplus y \leq a \oplus b \quad x \ominus b \leq a \ominus y \quad (8.26)$$

$$x \otimes y \leq a \otimes b \quad x \oslash b \leq a \oslash y$$

Quindi è possibile che per certi valori $0 \leq x \leq a$ e $0 \leq y \leq b$ valga
 $x \quad y = a \quad b$ o $x \quad y = a \quad b$, mentre non è possibile che valga
 $x \quad y > a \quad b$ oppure $x \quad y > a \quad b$.

Dai precedenti assiomi si ricavano le seguenti proposizioni, che rappresentano proprietà importanti e fondamentali per un'aritmetica:

$$y \geq 0 \rightarrow x \oplus y \geq x \quad (8.27)$$

$$x \geq y \rightarrow x \ominus y \geq 0$$

$$(x \geq 0) \wedge (0 \leq y \leq 1) \rightarrow x \otimes y \leq x$$

$$0 < x \leq y \rightarrow x \oslash y \leq 1$$

$$x \ominus x = 0$$

$$x \oplus 0 = x \ominus 0 = x$$

$$x \otimes 0 = 0$$

$$x \otimes 1 = x \oslash 1 = x$$

$$x \oslash x = 1$$

Si osservi che mancano la legge associativa e la legge distributiva. Un esempio di calcolo numerico, in cui non valga la legge associativa dell'addizione, è il seguente (in un'aritmetica a quattro cifre):

$$x = 9.900 \quad y = 1.000 \quad z = -0.999$$

$$1. (x \oplus y) \oplus z = 10.90 \oplus (-0.999) = 9.910$$

$$2. x \oplus (y \oplus z) = 9.900 \oplus 0.001 = 9.901$$

Un esempio in cui non valga invece la legge distributiva (sempre in un'aritmetica a quattro cifre) è:

$$x = 1100, \quad y = -5.000 \quad z = 5.001$$

1. $(x \otimes y) \oplus (x \otimes z) = -5500. \oplus 5501. = 1.000$
2. $x \otimes (y \oplus z) = 1100. \otimes 0.001 = 1.100$

Le operazioni che richiedono maggiore attenzione sono l'addizione e la sottrazione. Infatti, la causa principale degli errori di calcolo numerico risiede nella sottrazione di numeri di valore quasi eguale; in tal caso, infatti, le cifre più significative si eliminano fra loro e la differenza risultante perde un certo numero di cifre significative o anche tutte. Questo fenomeno è detto *cancellazione*.

Altra causa d'errori è la divisione per valori molto piccoli, poiché il risultato può facilmente superare il valore di *overflow*. Quindi deve essere evitata non solo la divisione per 0, ma anche la divisione per valori prossimi allo 0.

Come misura della *precisione* di un'aritmetica in virgola mobile, si può assumere il numero ε , definito da:

$$\varepsilon = \min_{x > 0} (x | (1 + x)^{-1} \neq 1) \quad (8.28)$$

cioè ε è il più piccolo numero positivo tale che $1 \neq (1 + \varepsilon)$. Se in un calcolatore la precisione della rappresentazione dei valori di tipo *real* è di n cifre decimali, vale: $\varepsilon = 10^{-n}$.

Benché, da un punto di vista puramente matematico, i numeri interi siano un sottoinsieme dei numeri reali, per ragioni pratiche si suole considerare il tipo *real* e il tipo *integer* come due tipi *disgiunti*. Per poter stabilire il tipo di una costante in base alla sua rappresentazione, si conviene che un numero è di tipo *integer* se, e solo se, la sua rappresentazione non contiene alcuna cifra frazionaria né alcun fattore di scala. Tutti i numeri con cifre frazionarie (o con un fattore di scala) devono essere classificati nel tipo *real*. Inoltre, nel calcolo numerico, valgono le regole seguenti.

1) In una espressione con valori reali un qualsiasi operando di tipo *real* può essere sostituito con un operando di tipo *integer*. Una conversione esplicita *integer* \rightarrow *real* non è quindi necessaria. Nondimeno, il programmatore deve tener presente che tale conversione avviene implicitamente: infatti essa viene eseguita direttamente dal compilatore in tutti quei calcolatori che usano rappresentazioni distinte per il tipo *real* e per il tipo *integer*.

2) Se una grandezza con valori reali ne rimpiazza una che può assumere solo valori interi, è necessario specificare esplicitamente una

funzione di conversione *real* \rightarrow *integer*. Adotteremo come funzione di conversione standard, la più usata dagli attuali calcolatori:

$$\text{trunc}(x)$$

il valore da essa fornito è il numero intero che si ottiene *troncando* le cifre frazionarie di x . Per esempio:

$$\text{trunc}(5.8) = 5, \quad \text{trunc}(-4.3) = -4$$

Possiamo ora definire la funzione di arrotondamento:

$$\text{round}(x) = \begin{cases} \text{trunc}(x + 0.5) & \text{se } x \geq 0 \\ \text{trunc}(x - 0.5) & \text{se } x < 0 \end{cases} \quad (8.29)$$

Esempio 8.1. Soluzione di un'equazione di secondo grado. Concludiamo il capitolo con un esempio che illustra, nell'ambito della programmazione, gli effetti derivanti dall'uso di un'aritmetica in virgola mobile, in particolare l'effetto della cancellazione. Siano x_1 e x_2 le due soluzioni dell'equazione di secondo grado

$$a * x^2 + b * x + c = 0 \quad a \neq 0$$

Una traduzione diretta della nota formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (8.30)$$

dà luogo alla sequenza di istruzioni

$$\begin{aligned} d &:= \text{sqrt}(\text{sqr}(b) - 4 * a * c); \\ x_2 &:= -(b + d)/(2 * a); \quad x_1 := (d - b)/(2 * a) \end{aligned} \quad (8.31)$$

Per esempio, se $a = 1.000$, $b = -200.0$, $c = 1.000$, eseguendo i calcoli in un'aritmetica con 4 cifre, si hanno i risultati

$$\begin{aligned} d &= \text{sqrt}(40000 - 4.000) = 200.0 \\ x_1 &= 400.0/2.000 = 200.0 \\ x_2 &= 0.000/2.000 = 0.000 \end{aligned}$$

Però, i risultati esatti fino alla quarta cifra sono:

$$x_1 = 200.0 \quad \text{e} \quad x_2 = 0.005$$

Se si assume come misura della bontà di un risultato la precisione relativa, x_2 dev'essere considerato un risultato *completamente errato*. Un metodo di soluzione che permette di superare le difficoltà connesse all'uso di un'aritmetica in virgola mobile di precisione finita, si basa sulla relazione:

$$x_1 * x_2 = c/a \quad (\text{Vieta}) \quad (8.32)$$

Si può calcolare una delle due soluzioni, e precisamente quella di valore maggiore, con la formula usuale. La seconda soluzione si ottiene dalla prima in base alla (8.32) mediante una divisione, operazione che conserva la precisione relativa. Si ottiene così il programma:

```
d := sqrt(sqr(b) - 4 * a * c);           (8.33)
if b ≥ 0 then x1 := -(b + d)/(2 * a)
else x1 := (d - b)/(2 * a);
x2 := c/(x1 * a)
```

Il problema trattato è uno dei tanti esempi, in cui non si possono applicare in modo automatico i *metodi di soluzione* della matematica, poiché si opera con i limiti di un calcolatore reale.

8.5. Problemi

8.1. Quali delle seguenti espressioni sono sintatticamente corrette? Qual'è il loro tipo? Si facciano le seguenti dichiarazioni di variabili:

var	<i>x, y, z: real;</i>	<i>i, j, k: integer</i>
<i>x + y * i</i>	<i>i mod (j + y)</i>	<i>i + j - k</i>
<i>i div j + x</i>	<i>x + y < i + j</i>	<i>k - trunc(x * i)</i>
<i>i * x + j * y</i>	<i>x < y ∧ y < z</i>	<i>x = i</i>

Determinare in ogni caso il numero delle funzioni di trasferimento implicite *integer → real*.

8.2. Completare i programmi (7.18), (7.19), (7.20), (7.21), (7.24) e (7.30) con le dichiarazioni delle variabili necessarie.

8.3. Scrivere un programma che calcoli la somma

$$1 - 1/2 + 1/3 - 1/4 + \dots + 1/9999 - 1/10000$$

nei seguenti quattro modi:

- 1) sommando i termini da sinistra verso destra;
- 2) sommando i termini da destra verso sinistra;

- 3) sommando separatamente i termini positivi e quelli negativi, da sinistra verso destra;
- 4) come in 3), ma da destra verso sinistra.

Valutare i vantaggi, gli svantaggi e il tempo di calcolo delle quattro versioni diverse di tale programma. Paragonare i risultati ottenuti eseguendo il programma (nelle 4 versioni) su di un calcolatore; suggerimento: le istruzioni della forma $a := b*(-1)$ possono essere scritte nella forma $a := -b$. Il risultato, con una precisione di 30 cifre decimali, è
0.69309718305994529691723271362

8.4. Scrivere un programma che moltiplichi n volte la variabile complessa z per la costante complessa $c = 0.6 + 0.8i$.

Una variabile complessa viene rappresentata mediante due variabili reali x e y . Si osservi che la costante c ha valore assoluto 1. Se anche il valore iniziale di z è 1 (per esempio $z_0 = 5/13 + 12/13i$), allora il valore di z dopo n moltiplicazioni per c è:

$$z_n = z_0 * c^n \quad \text{e} \quad |z^n| = 1$$

Quindi, se si calcola, nel programma, il valore

$$|z_n| = \sqrt{x_n^2 + y_n^2}$$

dopo n moltiplicazioni, ciò può essere usato per valutare la precisione del calcolatore. Scegliere $n=5000$.

Programmi basati su relazioni di ricorrenza

9.1. Successioni

Nei capitoli precedenti abbiamo introdotto le principali strutture dei programmi, i principali oggetti del calcolo e gli operatori che agiscono su tali oggetti. Nel presente capitolo esamineremo più da vicino i programmi che rappresentano semplici ripetizioni, in generale della forma:

while C **do** I (9.1)

Innanzitutto ricordiamo che una ripetizione ha senso solo se, dopo un numero finito di ripetizioni, la condizione C non è più soddisfatta. Ne segue che I deve modificare almeno una variabile che compare nell'espressione C . Se usiamo la notazione V per indicare l'insieme delle variabili, cioè se consideriamo ogni variabile del programma come una componente dell'insieme V , l'istruzione (9.1) è fornita dallo schema generale (di programma):

```
 $V := v_0;$ 
while  $p(V)$  do  $V := f(V)$ 
```

(9.2)

dove p è un predicato (funzione booleana) e f è una funzione.

Indichiamo con V_i il valore delle variabili V dopo i ripetizioni dell'istruzione I . Allora V percorre la successione di valori

$v_0, V_1, V_2, \dots, V_n$ (9.3)

e valgono le seguenti relazioni:

- | | | |
|----------|-----------------------|--------------------|
| per ogni | 1. $v_i = f(v_{i-1})$ | for all $i > 0$ |
| per | 2. $v_i \neq v_j$ | for all $i \neq j$ |
| | 3. $\neg p(v_n)$ | |
| per ogni | 4. $p(v_i)$ | for all $i < n$ |
- (9.4)

All'inizio della ripetizione V deve avere un ben determinato valore V_0 . La disattenzione di questa regola fondamentale è uno degli errori più frequenti nella programmazione.

Le relazioni (9.4) mostrano che l'istruzione **while** è la forma più adatta a rappresentare programmi il cui compito sia espresso da una relazione di ricorrenza. Spiegheremo questa affermazione con dei semplici esempi.

Esempio 9.1. Calcolo del fattoriale: la funzione

$$f(n) = 1 * 2 * 3 * \dots * n = n! \quad (n \geq 0) \quad (9.5)$$

può essere calcolata facendo uso della relazione di ricorrenza

$$f(n) = n * f(n-1) \quad \text{per } n > 0 \quad (9.6)$$

e della condizione iniziale

$$f(0) = 1$$

Per calcolare $f(n)$ introduciamo le variabili x e k , facendo in modo che, dopo i ripetizioni dell'istruzione I , i loro valori siano definiti dalle relazioni

$$\left. \begin{array}{l} x_i = k_i * x_{i-1} \\ k_i = k_{i-1} + 1 \end{array} \right\} \quad \text{per } i > 0 \quad (9.7)$$

e dalla condizione iniziale

$$x_0 = 1, \quad k_0 = 0$$

Il programma che si ricava dalle relazioni (9.7) e dallo schema (9.2) è:

```
var F, K: integer; {n ≥ 0}
begin F := 1; K := 0;
  while K ≠ n do
    begin {F = K!}
      K := K + 1; F := K * F
    end
    {F = n!}
end
```

(9.8)

Poiché k percorre la successione dei numeri naturali e vale $n \geq 0$, il programma termina. Si osservi che l'ordine con cui vengono eseguite le due istruzioni ripetute è determinante. Infatti, se l'ordine fosse invertito, il programma corrisponderebbe alle relazioni ricorsive:

$$\begin{aligned} x_i &= k_{i-1} * x_{i-1} \\ k_i &= k_{i-1} + 1 \end{aligned} \quad (9.9)$$

e non alle (9.7).

Esempio 9.2. Inverso di un numero reale: consideriamo le due successioni a_0, a_1, \dots e c_0, c_1, \dots , assegnate dalle relazioni di ricorrenza

$$\left. \begin{aligned} a_i &= a_{i-1} * (1 + c_{i-1}) \\ c_i &= c_{i-1}^2 \end{aligned} \right\} \text{for } i > 0 \quad (9.10)$$

e dalla condizione iniziale

$$a_0 = 1, \quad c_0 = 1 - x \quad 0 < x < 1$$

Si può dimostrare che

$$a_n = \frac{1 - c_n}{x} \quad (9.11)$$

allora, da $|c_n| = c_0^{2^n}$ e da $|c_0| < 1$ segue:

$$\lim_{n \rightarrow \infty} a_n = \frac{1}{x} \quad (9.12)$$

Traccia della dimostrazione: usare le relazioni

$$\begin{aligned} 1) \quad a_n &= (1 + c_{n-1}) * \cdots * (1 + c_1) * (1 + c_0) \\ 2) \quad \frac{1 + c_{i-1}}{1 - c_i} &= \frac{1}{1 - c_{i-1}}. \end{aligned}$$

Dallo schema (9.2) e dalle relazioni di ricorrenza (9.10) si ottiene il programma (9.13), che calcola un valore approssimato di $1/x$ usando solo addizioni e moltiplicazioni.

```
var A, C: real; {0 < x < 1}
begin A := 1; C := 1 - x;
  while abs(C) > ε do
    begin {A * x = 1 - C, 0 < C < 1}
      A := A * (1 + C); C := sqr(C)
    end
    {(1 - ε)/x ≤ A < 1/x}.
  end.
```

(9.13)

Il programma (9.13) termina quando $C \leq ε$. Poiché $c_0 < 1$ e $c_n = c_0^{2^n}$, è garantita l'esistenza di un numero n tale che sia $c_n < ε$ e che, per tutti gli $i < n$, sia $c_i > ε$. Si osservi che anche in questo programma l'ordine di esecuzione delle due istruzioni ripetute è essenziale.

Esempio 9.3. Radice quadrata: consideriamo le due successioni a_0, a_1, \dots e c_0, c_1, \dots assegnate dalle relazioni di ricorrenza

$$\left. \begin{aligned} a_i &= a_{i-1} * (1 + \frac{1}{2}c_{i-1}) \\ c_i &= c_{i-1}^2 * \frac{1}{4}(3 + c_{i-1}) \end{aligned} \right\} \text{for } i > 0 \quad (9.14)$$

e dalle condizioni iniziali

$$a_0 = x, \quad c_0 = 1 - x \quad 0 < x < 1$$

Con una manipolazione formale algebrica si dimostra che

$$a_n = \sqrt{x * (1 - c_n)} \quad (9.15)$$

Poiché $|c_0| > 0$, segue inoltre che

$$\lim_{n \rightarrow \infty} c_n = 0 \quad \text{and} \quad \lim_{n \rightarrow \infty} a_n = \sqrt{x} \quad (9.16)$$

Traccia della dimostrazione: usare le relazioni:

$$\begin{aligned} a_n &= (1 + \frac{1}{2}c_{n-1}) * (1 + \frac{1}{2}c_{n-2}) * \cdots * (1 + \frac{1}{2}c_0) * x \\ (1 + \frac{1}{2}c_{i-1}) &= \frac{\sqrt{1 - c_i}}{\sqrt{1 - c_{i-1}}} \end{aligned}$$

e

$$\begin{aligned} \frac{x}{a_n} &= \frac{1}{(1 + \frac{1}{2}c_0) \cdots (1 + \frac{1}{2}c_{n-1})} = \frac{\sqrt{1 - c_{n-1}}}{(1 + \frac{1}{2}c_0) \cdots (1 + \frac{1}{2}c_{n-2}) \sqrt{1 - c_n}} = \cdots \\ &= \frac{\sqrt{1 - c_1}}{(1 + \frac{1}{2}c_0) \sqrt{1 - c_n}} = \frac{\sqrt{1 - c_0}}{\sqrt{1 - c_n}} = \frac{\sqrt{x}}{\sqrt{1 - c_n}} \end{aligned}$$

Dallo schema (9.2) e dalle relazioni di ricorrenza (9.14) si ottiene:

```
var A, C: real; {0 < x < 1}
begin A := x; C := 1 - x;
  while abs(C) > ε do
    begin {A^2 = x * (1 - C), C ≥ 0}
      A := A * (1 + 0.5 * C);
      C := sqr(C) * (0.75 + 0.25 * C)
    end
    {x * (1 - ε) ≤ A^2 < x}
  end.
```

(9.17)

che da una valutazione dell'errore.

Il programma termina in base alla condizione (9.16), che afferma che c tende a 0.

9.2. Serie

Le istruzioni della forma (9.1) sono adeguate non solo ai calcoli sulle successioni numeriche ma anche ai calcoli sulle serie. Consideriamo la serie

$$s_0, s_1, s_2, \dots \quad (9.18)$$

costruita a partire dalla successione di termini

$$t_0, t_1, t_2, \dots \quad (9.19)$$

ponendo

$$s_i = t_0 + t_1 + \dots + t_i \quad (9.20)$$

Se la successione è definita dalla relazione di ricorrenza

$$t_i = f(t_{i-1}) \quad \text{per } i > 0 \quad (9.21)$$

per la serie vale:

$$\begin{aligned} s_i &= s_{i-1} + t_i \quad \text{per } i > 0 \\ s_0 &= t_0 \end{aligned} \quad (9.22)$$

Il programma che, dopo i ripetizioni, assegna alla variabile S il valore s_i , è fornito dallo schema:

```
T := t_0; S := T;
while p(S, T) do
begin T := f(T); S := S + T
end
```

(9.23)

e valgono le seguenti relazioni:

1. $t_i = f(t_{i-1}) \quad \text{per } i > 0$
 2. $t_i \neq t_j \quad \text{per } i \neq j$
 3. $s_i = s_{i-1} + t_i \quad \text{per } i > 0$
 4. $\neg p(s_n, t_n)$
 5. $p(s_i, t_i) \quad \text{per tutti gli } i < n$
- (9.24)

Esempio 9.4. Approssimazione di $\exp(x)$: i termini delle somme parziali

$$s_i = 1 + x + \frac{x^2}{2!} + \dots + \frac{x^i}{i!} \quad (9.25)$$

sono definiti dalla relazione

$$t_j = t_{j-1} * x/j \quad j > 0 \quad (9.26)$$

e dalla condizione iniziale $t_0 = 1$.

Com'è noto, vale

$$\lim_{n \rightarrow \infty} s_n = \exp(x) \quad (9.27)$$

e la serie converge a $\exp(x)$ per tutti i numeri reali x , cioè i termini decrescono in modo tale che la loro somma converga a un limite fissato. Questo fatto viene utilizzato dal programma (9.28), costruito in base allo schema (9.23) e alla relazione di ricorrenza (9.26).

```
var T, S: real; K: integer;
begin T := 1; S := T; K := 0;
while T > ε do
begin {S = 1 + x + ... + x^K/K!, T = x^K/K! > ε}
      K := K + 1; T := T * x/K; S := S + T
end
end.
```

(9.28)

Il valore finale S approssima $\exp(x)$ con un errore pari a:

$$\sum_{i=K+1}^{\infty} (x^i/i!)$$

Di solito il calcolo viene arrestato quando il rapporto fra l'ultimo termine calcolato t_i e la somma parziale s_i viene ad avere un valore inferiore a un $ε$ prefissato. Tale criterio di terminazione è usato nell'esempio seguente.

Esempio 9.5. Approssimazione di $\sin(x)$: i termini delle somme parziali

$$s_i = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{2i-1} * \frac{x^{2i-1}}{(2i-1)!} \quad (9.29)$$

sono definiti dalle relazioni

$$\begin{aligned} t_j &= -t_{j-1} * \frac{x^2}{k_j * (k_j - 1)} \\ k_j &= k_{j-1} + 2 \end{aligned} \quad (9.30)$$

e dai valori iniziali $t_0 = x$ e $k_0 = 1$.

Com'è noto

$$\lim_{n \rightarrow \infty} s_n = \sin(x) \quad (9.31)$$

Il programma che calcola un valore approssimato di $\sin(x)$, ottenuto sostituendo le (9.30) nello schema generale (9.23), è:

```
var S, T: real; K: integer;
begin T := x; K := 1; S := T;
  while abs(T) > ε * abs(S) do
    begin K := K + 2; T := -T * sqr(x)/(K * (K - 1));
      S := S + T
    end
  end.
```

(9.32)

Si osservi che nei programmi (9.13), (9.17), (9.28) e (9.32) il numero dei termini della successione o della serie calcolati non può essere determinato facilmente in base alle condizioni di terminazione. I termini della serie (o delle successioni) dei precedenti esempi hanno valore assoluto tendente a 0. Quindi il numero delle ripetizioni dipende sicuramente da ϵ . Ma esso dipende anche dalla rapidità di convergenza della successione; questa, a sua volta, è una funzione dell'argomento x . Perciò, quando si usano delle relazioni di ricorrenza per costruire dei programmi, occorre sempre far molta attenzione, anche se l'analisi matematica garantisce la convergenza. Nell'esempio (9.4) la rapidità di convergenza è alta per valori di x piccoli. Per $x < 0$ si usa la relazione

$$\exp(-x) = 1/\exp(x) \quad \text{per } x < 0 \quad (9.33)$$

mentre per valori di $x > 1$ si usa la relazione

$$\exp(i + y) = \exp(i) * \exp(y) \quad \text{per } x > 1 \quad (9.34)$$

dove $x = i + y$ e $i = \text{trunc}(x)$. Infatti $\exp(i)$ può essere calcolato con maggior rapidità eseguendo ripetute moltiplicazioni di e .

Occorre anche fare attenzione quando i termini della serie hanno segni alternati. Nell'esempio (9.5), la serie converge rapidamente solo per piccoli valori di x . Per argomenti compresi nell'intervalllo $(\pi, 2\pi)$, si usa la formula

$$\sin(x + \pi) = -\sin(x) \quad (9.35)$$

e per quelli esterni all'intervalllo $(0, 2\pi)$, si usa la relazione

$$\sin(x + 2n\pi) = \sin(x) \quad (9.36)$$

Se si usano inoltre le relazioni

$$\begin{aligned} \sin(x) &= \sin(\pi - x) \quad \text{per } \frac{\pi}{2} \leq x \leq \pi \\ \sin(x) &= \cos\left(\frac{\pi}{2} - x\right) \quad \text{per } \frac{\pi}{4} \leq x < \frac{\pi}{2} \end{aligned} \quad (9.37)$$

è possibile calcolare i valori di $\sin(x)$ limitandosi a considerare l'intervalllo $0 \leq |x| \leq \pi/4$, dove la rapidità di convergenza è sufficientemente alta per gli scopi pratici e dove il numero dei termini è abbastanza piccolo da mantenere entro limiti trascurabili gli errori di arrotondamento dovuti all'uso di un'aritmetica con precisione finita.

Il programma (9.32) è adatto per mostrare l'utilità di un'altra *regola-base della programmazione*. All'interno del gruppo delle istruzioni ripetute viene calcolato il valore di x^2 ; questo calcolo viene ripetuto, benché il valore di x (e quindi anche quello di x^2) rimanga inalterato. Un tale spreco di calcolo può essere eliminato calcolando x^2 una sola volta (ciò va fatto fuori dall'istruzione di **while**) e assegnando il valore x^2 a una variabile ausiliaria h , che sostituirà x^2 dovunque necessario. La formulazione generale è fornita dalla *regola base*: se un valore $f(x)$ viene utilizzato in una istruzione ripetuta R e se il valore di x non viene modificato durante l'esecuzione di R , si introduce una variabile ausiliaria h e si assegna ad h il valore $f(x)$; si sostituisce poi h a $f(x)$ in R . Cioè lo schema

```
while P do
  begin ... f(x) ... end
```

(9.38)

viene sostituito dallo schema:

```
h := f(x);
while P do
  begin ... h ... end
```

(9.39)

9.3. Problemi

9.1. Modificare i programmi (9.28) e (9.32) in modo che le funzioni $\exp(x)$ e $\sin(x)$ siano calcolate in modo efficiente, utilizzando le relazioni da (9.33) a (9.37).

9.2. Scrivere un programma che calcoli una approssimazione di $\cos(x)$, in base allo schema generale (9.23).

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

9.3. In base allo schema generale (9.23), scrivere un programma che calcoli una somma finita della serie:

$$\int_0^x \exp(-u^2) du = x - \frac{x^3}{3 \cdot 1!} + \frac{x^5}{5 \cdot 2!} - \frac{x^7}{7 \cdot 3!} + \dots$$

N.B.: si osservi che la serie converge lentamente per $x > 1$; quali sono le conseguenze da tener presenti usando un calcolatore con precisione finita? (Vedere, per esempio, l'andamento dei singoli termini per $x = 5, 6$).

9.4. Scrivere, in base allo schema generale (9.2), un programma che calcoli i numeri di Fibonacci nei seguenti due modi:

1) secondo la definizione ricorsiva

$$f_{i+1} = f_i + f_{i-1} \quad i > 0$$

$$f_0 = 0, \quad f_1 = 1$$

2) secondo la formula

$$f_i = \text{round}(c^i / \sqrt{5})$$

dove

$$c = (1 + \sqrt{5})/2.$$

Usare il valore approssimato: $\sqrt{5} = 2,236068$; determinare il minimo i per il quale i valori f_i calcolati nei due modi sono diversi.

9.5. Scrivere un programma, secondo lo schema generale (9.2), che calcoli il logaritmo di un numero x in base 2 ($1 \leq x < 2$). Impiegare le relazioni di ricorrenza:

$$a_i = \begin{cases} a_{i-1}^2 & \text{if } a_{i-1}^2 < 2 \\ \frac{1}{2}a_{i-1}^2 & \text{if } a_{i-1}^2 \geq 2 \end{cases}$$

$$b_i = \frac{1}{2}b_{i-1}$$

$$s_i = \begin{cases} s_{i-1} & \text{if } a_{i-1}^2 < 2 \\ s_{i-1} + b_i & \text{if } a_{i-1}^2 \geq 2 \end{cases}$$

per $i > 0$ e $a_0 = x$, $b_0 = 1$, $s_0 = 0$.

Fermare il calcolo quando $b_n \leq \varepsilon$. Trovare gli invarianti della ripetizione e verificare in base a essi che $\lim s_n = \log(x)$ è la terminazione dell'algoritmo.

9.6. Verificare i risultati nei programmi (9.13) e (9.17), in base alle condizioni di verifica necessarie dedotte dagli invarianti indicati.

9.7. Sostituire nei programmi (9.13), (9.17), (9.28) e (9.32) la grandezza ε con zero. I programmi risultanti terminano ancora usando un'aritmetica in virgola mobile con precisione finita? Quali assiomi (v. cap. 8) sono particolarmente importanti per assicurare tale condizione?

9.8. Eseguire i programmi (9.13), (9.17), (9.28) e (9.32) su di un calcolatore e introdurre un contatore del numero delle ripetizioni. Determinare la rapidità di convergenza dei programmi per differenti valori di x .

differenti (lettore e perforatore) e poiché in un programma ogni variabile di tipo *file* è abbinata a un dispositivo ben preciso, è possibile eseguire di volta in volta una sola delle due operazioni. Un *file* che può essere solamente letto è detto *file d'ingresso (input file)*, mentre un *file* che può essere solamente scritto è detto *file di uscita (output file)*.

3) *stampanti*: il *file* corrispondente è un *file d'uscita*.

Il concetto di *file* serve per fare *astrazione* da questi particolari mezzi di memoria e per formulare in generale le caratteristiche e gli operatori a essi comuni. Nei paragrafi successivi descriveremo gli operatori più importanti, senza specificare quale legame esista tra una variabile di tipo *file* e un certo mezzo di memoria.

10.2. Generazione di un *file*

Nella dichiarazione di una variabile di tipo *file* (come per le variabili scalari) vengono contemporaneamente stabiliti il nome, il tipo e la struttura. Il numero delle componenti (*lunghezza del file*) rimane però libero e può variare durante l'esecuzione del processo di calcolo; questa proprietà *dinamica* è una caratteristica particolarmente importante della struttura di un *file*. In ogni caso, le componenti non possono venir introdotte o prelevate a piacere, ma solo in un ordine strettamente sequenziale, mediante gli operatori e le procedure standard che esporremo in questo paragrafo e nel successivo.

La procedura *put (f)* serve per aggiungere una componente alla fine di un *file f*. La componente aggiunta a *f* è il valore (o contenuto) del *buffer* $f\uparrow$. L'effetto di *put (f)* è descritto formalmente dalla (10.4).

$$\{(f = \alpha) \wedge (f\uparrow = x)\} \quad put(f) \quad \{f = \alpha \cdot \langle x \rangle\} \quad (10.4)$$

Per permettere una realizzazione efficiente delle variabili e degli operatori di tipo *file*, è utile che il valore della variabile di *buffer* sia indefinito dopo l'esecuzione dell'operatore *put*, in modo che *put (f)* non eserciti alcun effetto collaterale su *f*.

*Esempio 10.1. Generazione di un *file*:* generare un *file* la cui componente *i*-esima possieda il valore i^2 e contenga tutti i quadrati perfetti minori di *n*.

Dalle relazioni ricorsive

$$\left. \begin{array}{l} a_i = a_{i-1} + b_i \\ b_i = b_{i-1} + 2 \end{array} \right\} \text{for } i > 1 \quad (10.5)$$

ponendo $a_1 = b_1 = 1$, si ottiene $a_1 = i^2$. Il programma cercato è:

```
var A, B: integer;
f: file of integer;
begin A := 1; B := 1;
repeat {A = (B + 1)^2/4}
      f\uparrow := A; put(f);
      B := B + 2; A := A + B
until A ≥ n
end. (10.6)
```

10.3. Ispezione di un *file*

Dopo che un *file* è stato preparato, generato, scritto, esso è pronto a essere ispezionato, letto, esaminato.

Si procede ancora in un modo strettamente sequenziale: le componenti vengono lette nel medesimo ordine col quale sono state scritte. Quindi durante la lettura vi è sempre una certa *posizione di ispezione del file*. Se consideriamo un *file* concreto, per esempio, un nastro magnetico, questa posizione corrisponde alla posizione della testina di lettura. Durante la scrittura, la testina è sempre posizionata alla fine del *file* astratto, per cui non occorre indicarne esplicitamente la posizione; è invece necessario poter esprimere formalmente la posizione di lettura. Il modo più semplice è assegnare un nome, per esempio f_s , alla parte del *file* a sinistra della posizione di lettura e un nome, per esempio f_d , alla parte del *file* a destra.

Verrà sempre la relazione invariante:

$$f = f_s \& f_d \quad (10.7)$$

Si noti che f_s , f_d e $\&$ non sono impiegati direttamente in alcun linguaggio di programmazione. Tali oggetti ed operatori sono qui introdotti solamente per definire formalmente gli operatori-*file* (per esempio *put (f)*) e per illustrarne il significato.

La prima componente di f_d è l'unica che può essere esaminata direttamente. Per mezzo della funzione ausiliaria

$$\text{first}(\langle x_1, x_2, \dots, x_n \rangle) = x_1 \quad (10.8)$$

possiamo descrivere le due importanti procedure *file reset (f)* e *get (f)*.

L'istruzione *reset (f)* situa la posizione di lettura all'inizio del

Il tipo strutturato "file"

10.1. Il concetto di file

I tipi scalari trattati in precedenza hanno le seguenti caratteristiche: i loro valori rappresentano unità indivisibili e il campo dei valori indicato da un tipo scalare possiede un ordinamento (da cui il nome *scalare*). Nella elaborazione di grosse masse di dati è però necessario poter trattare anche interi insieme di valori con dei nomi che li indichino collettivamente; tali insiemi, indicati da un unico nome, sono chiamati *strutturati*. Le strutture possibili sono di diversi tipi: esse differiscono principalmente per il tipo dei valori che compongono la struttura, per il modo in cui essi vengono indicati e per le operazioni che permettono di estrarli.

Le variabili i cui valori sono aggregazioni di componenti, sono dette *variabili strutturate*; per definire il tipo di una variabile strutturata sono necessarie:

- 1) l'indicazione del tipo della struttura;
- 2) l'indicazione del tipo delle componenti.

Il tipo di struttura più semplice è la successione, o *sequenza*. L'esempio più noto di variabile con struttura sequenziale è la scheda. Nella elaborazione dei dati, il termine adottato per indicare le sequenze è *sequential file*, che nel seguito abbrevieremo con *file*, lasciando implicito l'attributo *sequenziale*. Per indicare che un tipo *F* ha struttura di tipo *file* e componenti di tipo *T*, si usa la scrittura:

$$\text{type } F = \text{file of } T \quad (10.1)$$

Tutte le componenti sono perciò dello stesso tipo.

I valori di *F* sono gli elementi del semigruppo libero costruito sul

tipo *T* delle componenti, semigruppo che può essere definito formalmente per mezzo dell'operazione di *concatenazione*. Siano

$$\alpha = \langle x_1, x_2 \dots x_m \rangle \quad \text{and} \quad \beta = \langle y_1, y_2 \dots y_n \rangle$$

due sequenze, o *file*. L'accostamento (o concatenazione) dei due *file* viene indicato nel modo seguente:

$$\alpha \cdot \beta = \langle x_1 \dots x_m, y_1, y_2 \dots y_n \rangle \quad (10.2)$$

Il tipo *F* definito dalla (10.1) è allora assegnato dai seguenti assiomi:

- 1) $\langle \rangle$ è un (valore di) *F*, la sequenza vuota;
- 2) se *f* è un *F* e *t* è un *T*, allora *f* · $\langle t \rangle$ è un *F*;
- 3) non vi sono altri valori di *F*.

Una variabile *f* di tipo *file*, viene indicata secondo le convenzioni esposte nel capitolo 8:

$$\text{var } f : F \quad \text{or} \quad \text{var } f : \text{file of } T \quad (10.3)$$

Il suo valore è sempre una successione di valori di *T*. Per ragioni che vedremo in seguito, si conviene che una qualsiasi dichiarazione di un *file* introduca (contemporaneamente alla variabile di tipo *file*) una variabile speciale appartenente al tipo *T* delle componenti, detta *buffer* e generalmente indicata con \uparrow . Essa viene utilizzata per togliere o per aggiungere componenti a *f*.

Nell'uso pratico dei calcolatori, i *file* giocano un ruolo particolarmente importante. Essi sono adeguati a rappresentare i dati memorizzati in apparecchiature con parti meccaniche; in tal caso, infatti, l'accesso sequenziale è il più adatto o addirittura l'unico possibile, poiché le informazioni sono inviate al meccanismo di lettura o di scrittura in un ordine strettamente sequenziale. Le operazioni che un calcolatore può eseguire su di una memoria dipendono dalle caratteristiche fisiche di quest'ultima. I mezzi di memoria sotto elencati sono generalmente trattati come memorie sequenziali e sono largamente usati:

- 1) *nastri magnetici e dischi* (memorie su tamburo magnetico): le operazioni possibili sono *scrivere*, *leggere*, *cancellare* (riposizionare e riscrivere);
- 2) *schede e nastri di carta*: le operazioni possibili sono *leggere* e *scrivere*; poiché le due operazioni vengono eseguite da dispositivi



file; ciò corrisponde, per esempio, al riavvolgimento di un nastro magnetico.

$$\{f = \alpha\} \quad \text{reset}(f) \quad \{(\bar{f} = \langle \rangle) \wedge (\bar{f} = \alpha) \wedge (f^\uparrow = \text{first}(\bar{f}))\} \quad (10.9)$$

Il *buffer* f^\uparrow possiede ora il valore della *prima* componente di f (se ne esiste una). Per tenere la componente successiva si utilizza l'operatore $\text{get}(f)$ che fa avanzare di un posto la posizione di lettura:

$$\begin{aligned} & \{(\bar{f} = \alpha) \wedge (\bar{f} = \langle x \rangle \cdot \beta)\} \quad \text{get}(f) \\ & \{(\bar{f} = \alpha \cdot \langle x \rangle) \wedge (\bar{f} = \beta) \wedge (f^\uparrow = \text{first}(\bar{f}))\} \end{aligned} \quad (10.10)$$

Si osservi che vale $f^\uparrow = \text{first}(f_D)$ sia dopo l'esecuzione di $\text{reset}(f)$ sia dopo l'esecuzione $\text{first}(f)$, però, è definita solo se f^\uparrow non è vuoto. È perciò indispensabile poter stabilire se f_D è vuoto. A tal fine, introdurremo il predicato (funzione booleana) standard $\text{eof}(f)$ (abbreviazione di *end of file*) con il seguente significato:

$$\text{eof}(f) \equiv \bar{f} = \langle \rangle \quad (10.11)$$

Si osservi che le istruzioni $\text{reset}(f)$ e $\text{get}(f)$ permettono di esaminare la variabile di *buffer* f^\uparrow solo quando $\text{eof}(f)$ non è soddisfatto.

Come quarta operazione base introduciamo la procedura standard $\text{rewrite}(f)$. Essa viene utilizzata per preparare la generazione di un *file* sostituendo al valore attuale la sequenza vuota:

$$\text{rewrite}(f) \quad \{f = \langle \rangle\} \quad (10.12)$$

I quattro operatori base non possono essere applicati in una successione arbitraria. In particolare, è prescritto che le operazioni di scrittura non possono seguire alle operazioni di lettura in modo arbitrario. La tabella (10.I) riporta le successioni permesse di operatori-*file*.

TABELLA 10.I.

operazione	operazione successiva permessa
put	put, reset, (rewrite)
reset	get, (reset, rewrite)
get	get, reset, rewrite
rewrite	put, (rewrite, reset)

La struttura sequenziale del *file* comporta che, molto spesso, un'istruzione di ripetizione è la struttura più naturale per la ispezione di un *file*. Sia I un'istruzione che operi sul valore f : gli schemi generali (10.13) e (10.14) sono le strutture caratteristiche di elaborazione del *file* f ; il programma (10.14) può essere utilizzato solo per *file* non vuoti.

```
while  $\neg \text{eof}(f)$  do
  begin  $S$ ;  $\text{get}(f)$ 
  end
```

(10.13)

```
repeat  $S$ ;  $\text{get}(f)$ 
until  $\text{eof}(f)$ 
```

(10.14)

Esempio 10.2. Determinare la lunghezza L di un file: si usa lo schema (10.13) come base e assegnando alla variabile S il valore $L := L + 1$ si ottiene il programma (10.15)

```
var  $L$ : integer;
begin  $L := 0$ ;
  while  $\neg \text{eof}(f)$  do
    begin { $L$  = numero di componenti letti}
       $L := L + 1$ ;  $\text{get}(f)$ 
    end
  end.
```

(10.15)

Esempio 10.3. Valor medio e varianza: data una serie di misure x_i , memorizzata in un *file* f non vuoto di numeri reali, calcolarne il valor medio e lo scarto quadratico medio (m e s^2) definiti mediante le:

$$m = \frac{1}{n} \sum_i x_i, \quad s^2 = \frac{1}{n} \sum_i (x_i - m)^2 \quad (10.16)$$

Si usa due volte lo schema (10.14):

```
var m, s; real; n: integer;
begin m := 0; n := 0; reset (f);
repeat n := n+1; m := m+f↑; get (f)
until eof (f);
m := m/n; s := 0; reset (f);
repeat s := s+sqr (f↑ - m); get (f)
until eof (f);
s := sqrt (s/n)
end
```

(10.17)

10.4. File costituiti da un testo

Un *file* formato da caratteri è detto *textfile*. Fra i tipi di dati più usati, i *textfile* hanno una posizione centrale, poiché, per la maggior parte, i dati di ingresso e di uscita dei programmi sono costituiti da *textfile*. I nastri di carta perforati con le apposite macchine da scrivere, le schede perforate con le macchine perforatrici e i risultati stampati con le telescriventi, sono esempi di *textfile*. Un processo di calcolo, che utilizzi questi *supporti* dei dati (nastri e schede perforate, telescrivente), può essere considerato come un processo di trasformazione di un *textfile* (*input*) in un altro *textfile* (*output*).

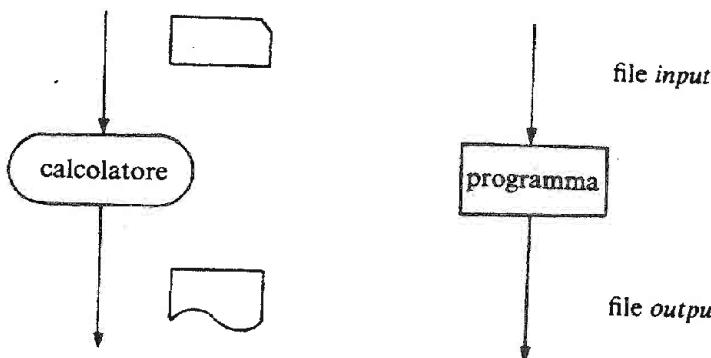


Figura 10.1.

I supporti dei dati, descritti sopra, sono degli oggetti *standard*, che è utile rappresentare con un *tipo standard* e con due *variabili standard*; essi sono sottointesi in ogni calcolatore, e sono definiti dalle *indicazioni implicite*:

```
type text = file of char
var input, output: text
```

(10.18)

Le due variabili indicano i dispositivi di ingresso e i dispositivi di uscita standard dei calcolatori, e devono soddisfare alle seguenti condizioni:

- 1) *input* può solo essere letto da parte del programma, ma non generato (o rigenerato); si può usare solo l'operatore *get*.
- 2) *output* può solo esser generato da parte del programma, ma non letto (o riposizionato); si può usare solo l'operatore *put*.

Poiché i testi sono suddivisi in linee (o righe), è necessario esprimere la struttura a righe; si danno principalmente due possibilità:

- 1) l'insieme di simboli definito dal tipo *char* contiene un simbolo speciale, che indica la fine di una linea; come separatore, si può usare il simbolo *blank*;
- 2) ogni linea viene *impaccata* in una sottosequenza di simboli, e il testo è trattato come un *file* di linee.

Per il caso 1) non sono necessarie ulteriori precisazioni; per il caso 2), invece, poiché l'insieme dei caratteri di molti calcolatori non comprende nessun simbolo di fine-linea, è necessario un esame ulteriore. Più precisamente, se un testo può essere definito come un *file of file of char*, diventa necessaria una descrizione circostanziata delle operazioni di elaborazione relative. Quindi, è necessaria una trattazione particolare del tipo *text*, che prevede l'introduzione di nuovi operatori standard per generare e per riconoscere le righe. Tali operatori sono:

writeln (f) aggiungi a *f* un simbolo di fine riga.
readln (f) *f* viene posto eguale al primo simbolo della riga successiva.
eoln (f) funzione booleana, vera dopo che *get (f)* ha avanzato la posizione di lettura oltre l'ultimo simbolo di una riga (*end of line*).

Per la elaborazione di testi in cui la struttura a righe non sia rilevante, assumiamo che *f* rappresenti lo spazio bianco (*blank*), quando *eoln (f)* è vero. Per il programmatore, che non usa il predicato *eoln*, ciò significa che ogni fine linea è uno spazio bianco.

Poiché i *textfile* sono usati molto spesso (in particolare i *file standard* *input* e *output*), si usano delle apposite abbreviazioni, definite nella tabella 10.II.

TABELLA 10.II.

Notazione per esteso	Notazione abbreviata
$c := \text{input}^\dagger; \text{get}(\text{input})$	$\text{read}(c)$
$\text{output}^\dagger := e; \text{put}(\text{output})$	$\text{write}(e)$
$\text{read}(c_1); \dots; \text{read}(c_m)$	$\text{read}(c_1, \dots c_m)$
$\text{write}(e_1); \dots; \text{write}(e_n)$	$\text{write}(e_1, \dots e_n)$

Inoltre quando si tratta di un *file standard* *input* o *output*, il nome standard del *file* è sottinteso, e può essere tralasciato; di solito, i *file standard* sono chiamati anche *default values* (valori di difetto), intendendo, con ciò, che vale:

$$\begin{aligned}\text{read}(X) &\equiv \text{read}(\text{input}, X) \\ \text{write}(X) &\equiv \text{write}(\text{output}, X)\end{aligned}\quad (10.19)$$

Esempio 10.4. Rappresentazione grafica di una funzione: si può stampare il grafico di una funzione a valori reali, costruendo opportune linee di stampa. Basta, infatti, calcolare la funzione in più punti equidistanti x_0, x_1, \dots, x_n e stampare nelle posizioni di coordinate $(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ il simbolo * (le altre posizioni di ogni simbolo saranno dei *blank*). La direzione dell'asse *y* è quella delle linee di stampa, e la distanza $d = x_i - x_{i-1}$ deve corrispondere alla distanza di interlinea. La posizione del simbolo * relativo alla coordinata x_i viene determinata calcolando il valore $y_i = f(x_i)$, moltiplicandolo per un opportuno fattore di scala *s* e arrotondandolo poi con il valore intero più vicino; il valore così contenuto deve indicare il numero degli spazi bianchi (*blank*) da anteporre all'asterisco. Nel programma (10.20), i seguenti valori sono usati come esempio:

$$\begin{aligned}f(x) &= \exp(-x) * \sin(2\pi x), \quad 0 \leq x < 4 \\ d &= 1/32 \quad (32 \text{ linee per intervallo } [x, x+1]) \\ s &50 \quad (50 \text{ spazi per intervallo } [y, y+1]) \\ h &65 \quad (\text{distanza dell'asse } x \text{ dal margine})\end{aligned}$$

```
const d = 0.03125; s = 50; h = 65; c = 6.28318; lim = 128;
var x, y:real; i, n:integer;
begin i := 0;
repeat x := i * d; y := exp(-x) * sin(c*x);
  n := round(s * y) + h
  repeat write(' ');
    n := n - 1
  until n = 0;
  writeln('*'); i := i + 1
until i = 128
end.
```

La suddivisione in linee influenza direttamente la struttura dei programmi che elaborano i *textfile*. Analogamente a quanto fatto in (10.13), si può formulare lo schema generale di tali programmi, fornito dalla struttura ciclica riportata in (10.21). La ripetizione più *esterna* elabora la singole linee, mentre quella più interna elabora i singoli caratteri di una stessa linea. Cioè, l'istruzione *I1* viene eseguita all'inizio di ogni linea, l'istruzione *I3* alla fine, e l'istruzione *I2* all'interno di ogni riga.

```
while not eof(f) do
  begin I1;
    while not eoIn(f) do
      begin read(f, ch); I2(ch)
        end;
    I3; readIn(f)
  end
```

Per i *file* non vuoti e non contenenti linee vuote, si può usare uno schema analogo al (10.14) illustrato in (10.22).

```
repeat I1;
  repeat read(f, ch); I2(ch)
    until eoIn(f);
  I3; readIn(f)
until eof(f)
```

Esempio 10.5. Trascrivere il file *input* in un file *output*, aggiungendo uno spazio bianco all'inizio di ogni linea; utilizzare lo schema della (10.21) e le abbreviazioni della tabella 10.II:

```
var ch: char;
begin
  while  $\neg \text{eof}(\text{input})$  do
    begin write(' '); {printer control}
      while  $\neg \text{eoIn}(\text{input})$  do
        begin read(ch); write(ch)
        end;
      writeln; readIn
    end.
end. (10.23)
```

10.5. Problemi

10.1. Siano assegnati due file d'ingresso *f* e *g*, che contengono le successioni numeriche ordinate f_1, f_2, \dots, f_m e g_1, g_2, \dots, g_n . tali che

$$f_{i+1} \geq f_i \text{ and } g_{j+1} \geq g_j \text{ for all } i, j$$

Fondere i due file in un unico file d'uscita *h*, in modo che:

$$h_{i+1} \geq h_i \text{ per } 1 \leq i < m+n$$

Scrivere il programma corrispondente.

10.2. Ampliare il programma (10.20) in modo che, contemporaneamente al grafico di $f(x)$, venga stampato anche l'asse delle *x*.

10.3. Scrivere un programma che copi un file *f* in un file *g*, sostituendo ogni coppia di spazi bianchi consecutivi con un unico spazio bianco, eccettuati quelli che si trovano all'inizio di ogni riga.

Le variabili del tipo strutturato *array* sono formate, come le variabili del tipo strutturato *file*, da insiemi di *componenti dello stesso tipo* scalare. Le seguenti proprietà distinguono però i due tipi di struttura:

- 1) ogni componente di un *array* ha un nome specifico ed è accessibile direttamente;
- 2) il numero degli elementi viene stabilito al momento dell'introduzione della variabile *array* e poi non viene più modificato.

Tali caratteristiche rendono necessarie delle convenzioni per:

- 1) indicare le singole componenti di un *array*;
- 2) indicare un tipo *array* e il numero delle sue componenti.

L'indicatore della componente di un *array* è costituito dal nome dell'*array* e dal così detto *indice*, che individua univocamente la componente. Ciò che caratterizza la struttura *array* è l'impiego di indici con valori numerabili appartenenti a un tipo scalare indicato in precedenza, detto *tipo* (o dominio) *dell'indice*. Valgono le seguenti convenzioni:

- 1) nella indicazione di un *array* vengono specificati sia il tipo delle componenti che il tipo dell'indice; vi è una corrispondenza univoca fra i valori dell'indice e le componenti dell'*array*; l'indicazione ha la forma:

type *A* = array [*tipo dell'indice*] of *tipo delle componenti* (11.1)

Esempi di indicazioni di variabili *array* (il cui tipo rimane *anonimo*) sono:

```
var a: array [1 ... 20] of real           (11.2)
var b: array [colori] of colori
```

La variabile *a* possiede 20 componenti di tipo *real* con valori dell'indice 1, 2, ..., 20, e la variabile *b* possiede quattro componenti di tipo *colore* (v. cap. 8) con valori dell'indice *rosso*, *giallo*, *verde* e *blu*;

2) la componente di un *array A* individuata dall'indice *i* viene indicata con *A_i* o con *A[i]*;

Esempi (v. [11.2]):

<i>a</i> [10]	<i>a</i> [<i>i+j</i>]
<i>b</i> [<i>rosso</i>]	<i>b</i> [<i>succ(giallo)</i>]

(11.3)

Ogni *array* rappresenta un'applicazione dal dominio dell'indice al tipo delle componenti: a ogni valore del dominio dell'indice corrisponde un unico valore del tipo delle componenti (cioè il valore della componente individuata dall'indice). La scrittura matematica usata per indicare le applicazioni è:

```
a: {1 ... 20} → real
b: colori → colori
```

(11.4)

Due *array* sono considerati *eguali* quando le componenti corrispondenti hanno lo stesso valore:

$$a = b \Leftrightarrow a_i = b_i \text{ per ogni } i \quad (11.5)$$

La condizione che ogni componente di un *array* possa essere nominata (ed estratta) direttamente comporta che nella esecuzione di un programma il tempo di accesso non deve dipendere dalla componente. Ciò restringe fortemente la scelta dei mezzi di memoria adeguati a rappresentare gli *array*. Non sono adatte in primo luogo le memorie sequenziali, quali i nastri di carta o magnetici, e in secondo luogo i dischi. Per la rappresentazione di un *array* è necessario usare memorie per le quali il tempo di accesso non dipende dalla scelta della cella di memoria. Gli esempi più noti di un tal tipo di memoria ad *accesso random* sono la memoria a nuclei magnetici e la memoria a semiconduttori integrati, che non contengono parti

meccaniche. Ma poiché i costi per unità di memoria sono incompatibilmente più alti rispetto alle memorie sequenziali, quando è necessario elaborare grossi volumi di dati vengono sempre usate queste ultime. Quando però un insieme di dati può essere contenuto nella memoria centrale, rapida e ad accesso diretto, per lo più si usa la struttura *array*. La memoria centrale è chiamata *memoria principale* mentre le memorie ad accesso sequenziale vengono chiamate *memorie secondarie*.

Esempio 11.1. Ricerca di una componente in un array: dati *x* e *A* [1], ..., *A* [*n*], individuare un indice *i* tale che *A* [*i*] = *x*; assegnare alla variabile *q* il valore *true* se un tale indice *i* esiste e il valore *false* altrimenti.

```
var i: 0 .. n; q: Boolean; {n > 0}
A: array [1 .. n] of T;
begin assegnamento di valore ad A
  i := 0;
  repeat i := i + 1; q := A[i] = x
    {A[j] ≠ x for j = 1 ... i - 1}
  until q ∨ (i = n)
end.
```

(11.6)

Esempio 11.2. Ricerca di una componente in un array ordinato: è lo stesso problema dell'esempio 11.1, ma i valori *A* [*i*] sono ordinati in modo che sia *A* [*i*] < *A* [*j*] per *i* < *j*. Benché il programma (11.6) sia utilizzabile anche in questa situazione, si può costruire un programma più efficiente basato sull'ordinamento dell'*array*. A tal fine si considerino le componenti dell'*array* come i nodi di un albero, in cui da ogni nodo si dipartono due rami (albero binario). La figura 11.1 mostra un albero con 15 nodi (*n* = 15).

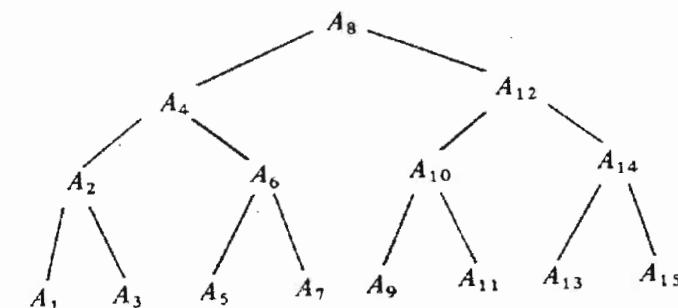


Figura 11.1.

Considerando la componente con indice $k = (n+1)/2$ si danno tre possibilità:

- 1) $A[k] = x$: si è trovato il valore cercato;
- 2) $A[k] < x$: nessuna delle componenti della branca sinistra può essere eguale a x , poiché il loro indice ha un valore $j < k$ e quindi $A[j] < A[k]$; la ricerca deve proseguire sulla branca destra e si seguirà lo stesso metodo di ricerca;
- 3) $A[k] > x$: in base allo stesso ragionamento la ricerca proseguirà solo nella branca sinistra.

Dalle precedenti considerazioni si ricava il programma:

```
var i, j, k: integer; q: Boolean;
A: array [1..n] of T;
begin assegnamento di valore ad A
  i := 1; j := n; q := false;
  repeat k := (i + j) div 2;
    if A[k] = x then q := true else
      if A[k] < x then i := k + 1 else j := k - 1
    until q  $\vee$  (i > j)
end.
```

(11.7)

Il metodo di ricerca corrispondente al programma (11.7) viene detto *ricerca binaria (binary search)*. Si noti che il numero dei confronti necessari è almeno 1 e al più $\log_2 n$. In media, tale numero si aggira intorno a $\log n$, e quindi è sensibilmente migliore rispetto al programma (11.6).

Esempio 11.3. Prodotto scalare: siano assegnati i valori x_1, \dots, x_n e y_1, \dots, y_n . Si calcoli il prodotto scalare:

$$s = \sum_{i=1}^n x_i * y_i \quad (11.8)$$

La sommatoria può essere rappresentata nella forma della relazione ricorsiva:

$$s_i = s_{i-1} + x_i * y_i, \quad s_0 = 0 \quad (11.9)$$

Secondo lo schema del programma (9.21), tale relazione dà luogo al programma:

```
var s: real; i: integer;
x, y: array [1..n] of real;
begin
```

(11.10)

```
begin {assegnamento dei valori iniziali a x e y}
  s := 0; i := 0;
  repeat { s =  $\sum_{j=1}^i x[j] * y[j]$  }
    i := i + 1; s := s + x[i] * y[i]
  until i = n
end.
```

Nell'esempio 11.3, come nell'esempio 11.1, le due variabili *array* vengono esaminate in modo puramente sequenziale, ma, a differenza dell'esempio 11.1, vengono considerate *tutte* le componenti in ogni esecuzione. L'ordine di esecuzione è però irrilevante; le n moltiplicazioni possono essere eseguite secondo una arbitraria successione, o addirittura simultaneamente. Questo stato di cose si verifica assai spesso operando con strutture *array*. Perciò, è utile introdurre una notazione speciale, che avrà la forma di una clausola ripetitiva simile alle clausole *while* e *repeat*. Siano I una istruzione, V una variabile, a e b due espressioni dello stesso tipo scalare di V . La istruzione

for $V := a$ **to** b **do** I (11.11)

significa che le due istruzioni

$V := x; I$ (11.12)

debbono essere ripetute, tante volte quanti sono i valori x appartenenti all'intervallo $a \dots b$. Seguendo le convenzioni adottate nei più comuni linguaggi di programmazione, i successivi valori di x vengono scelti secondo un ordine crescente prefissato. L'istruzione (11.12) è da considerarsi equivalente alla sequenza di istruzioni (11.13)

begin $V := v_1; I; V := v_2; I; \dots; V := v_n; I$ **end** (11.13)

dove $v_1 = a$, $v_n = b$ e $v_i = \text{succ}(v_{i-1})$ per $i = 2, 3, \dots, n$. Chiaramente, la funzione successore deve essere definita sull'insieme dei valori della variabile V (che quindi non può essere di tipo *real*). La istruzione (11.13) può essere rappresentata dallo schema (11.14).

```

if  $a \leq b$  then
begin  $V := a$ ;  $S$ ;
  while  $V < b$  do
    begin  $V := \text{succ}(V)$ ;  $S$ 
    end
end

```

(11.14)

Lo schema mostra, fra l'altro, che l'istruzione **for** non comporta alcuna computazione se $a > b$. Nel caso $a < b$ la complessità dello schema fa prevedere che le regole di verifica siano in ogni caso complicate come quelle delle istruzioni **while** e **repeat**.

11.1. Regole di verifica per l'istruzione **for**

P e $Q(V)$ sono proposizioni qualsiasi.

1) Premesse:

- a) $\{(V = a) \wedge P\} I \{Q(a)\}$
 - b) $\{Q(\text{pred}(x))\} I \{Q(x)\}$ per ogni x tale che $a < x \leq b$
- (11.15)

2) conseguenze:

- (a) $\{(a \leq b) \wedge P\} \text{ for } V := a \text{ to } b \text{ do } S \quad \{Q(b)\}$
 - (b) $\{(a > b) \wedge P\} \text{ for } V := a \text{ to } b \text{ do } S \quad \{P\}$
- (11.16)

L'esempio 11.3 può servire per illustrare suddette regole di verifica. Il programma (11.10) può essere scritto (tralasciando i commenti) nella forma:

```

begin  $s := 0$ ;
  for  $i := 1$  to  $n$  do  $s := s + x[i] * y[i]$ 
end

```

(11.17)

Sostituendo a P , $s = 0$ e a $Q(i)$, $s = \sum_{j=1}^i x_j * y_j$, le due premesse assumono la forma:

- (a) $\{(i = 1) \wedge (s = 0)\} s := s + x[i] * y[i] \left\{ s = \sum_{j=1}^1 x_j * y_j \right\}$
 - (b) $\left\{ s = \sum_{j=1}^{i-1} x_j * y_j \right\} s := s + x[i] * y[i] \left\{ s = \sum_{j=1}^i x_j * y_j \right\}$
- (11.18)

La condizione b) può essere facilmente verificata per tutti gli $i = 2, \dots, n$. La conseguenza è allora, per $n > 0$:

$$\{s = 0\} \text{ for } i := 1 \text{ to } n \text{ do } s := s + x[i] * y[i] \quad \left\{ s = \sum_{j=1}^n x_j * y_j \right\}$$
(11.19)

È evidente che la proposizione $Q(V)$ ha lo stesso ruolo degli invarianti nelle regole di verifica delle istruzioni **while** e **repeat**. Tuttavia è sempre necessaria l'indicazione esplicita della *variabile di controllo* V , poiché nella clausola **for** è implicitamente contenuto un assegnamento a tale variabile. Il grande vantaggio della istruzione **for** è che non occorre verificare la terminazione della ripetizione da essa rappresentata.

Naturalmente l'uso dell'istruzione **for** non si limita ai programmi che operano su strutture *array*; il suo massimo impiego si colloca però in tale ambito. Per l'uso adeguato dell'istruzione **for**, vale la seguente regola: quando bisogna ripetere un'istruzione, l'impiego della istruzione **for** è opportuno se il numero delle ripetizioni necessarie è noto a priori; in caso contrario occorre usare l'istruzione **while** o **repeat**.

I seguenti tre esempi di programmi illustrano il modo appropriato di usare le strutture *array* e le istruzioni **for**.

Esempio 11.4. Trovare il massimo $x[j]$: si tratta di individuare l'indice j tale che $x_j = \max(x_m, \dots, x_n)$.

```

var  $j, k : m \dots n$ ;
   $x : \text{array}[m \dots n]$  of  $T$ ;
begin  $j := m$ ;
  for  $k := m + 1$  to  $n$  do
    if  $x[k] > x[j]$  then  $j := k$ 
end

```

(11.20)

La proposizione $Q(k)$ da usare per la verifica è:

$$x[j] \geq x[i] \text{ per tutti gli } i = m, \dots, k$$
(11.21)

*Esempio 11.5. Ordinare un array: le componenti di un *array* debbono essere permutate in modo tale che i loro valori siano disposti secondo un ordine crescente. Il metodo di soluzione più immediato è il seguente:*

- 1) individuare l'elemento massimo secondo il programma (11.20);
- 2) scambiare x_j con x_1 ;
- 3) ripetere i passi 1) e 2) con gli insiemi $x_2, \dots, x_n, x_3, \dots, x_n$, ecc., fino a quando l'insieme da esaminare contiene solo x_n .

Una prima formulazione di tale procedimento è la seguente:

```
for h := 1 to n - 1 do          (11.22)
begin {Q(h - 1), if h > 1}
1: Trovare il massimo elemento  $x_j = \max(x_h, \dots, x_n)$ ;
2: scambiare  $x_h$  con  $x_j$ 
{Q(h)}
end
```

L'istruzione 1) può essere sostituita dal programma (11.20), mentre l'istruzione 2) può essere formulata come una successione di tre assegnamenti:

$$u := x[h]; x[h] := x[j]; x[j] := u \quad (11.23)$$

dove u è una variabile ausiliaria. Le proposizioni da usare nella verifica sono:

P : vuota

$$Q(h): x_1 \geq x_2 \geq \dots \geq x_h \geq x_i \quad \text{for all } i > h \quad (11.24)$$

Si ottiene infine il programma (11.25):

```
var h, j, k: 1..n;           (11.25)
x: array [1..n] of T;  u: T;
begin
  for h := 1 to n - 1 do
    begin j := h;
      for k := h + 1 to n do
        if x[k] > x[j] then j := k;
      u := x[h]; x[h] := x[j]; x[j] := u
    end
  end.
end.
```

In questo programma una istruzione **for** viene usata all'interno di un'altra istruzione **for**. Il numero delle esecuzioni dell'illustrazione "if $x[k] > x[j]$ then ..." è:

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n}{2}(n - 1) \quad (11.26)$$

Il tempo di calcolo di un tale metodo di ordinamento cresce proporzionalmente al quadrato del numero delle componenti da ordinare. Se tale numero è alto, occorre impiegare metodi di ordinamento più efficienti.

Le componenti delle variabili strutturate non sono necessariamente scalari: esse possono essere a loro volta strutturate. Se un *array* A ha per elementi altri *array*, viene detto *multidimensionale*; se gli elementi degli *array* componenti sono scalari, A viene detto *matrice*. La dichiarazione di un *array* multidimensionale avviene secondo lo schema (11.1).

Esempio:

var M : *array* [$a..b$] of *array* [$c..d$] of T (11.27)

la dichiarazione indica che M è costituita da $b - a + 1$ componenti (spesso dette righe della matrice M) con indici $a, a + 1, \dots, b$; ogni riga è a sua volta costituita da $d - c + 1$ componenti di tipo T con indici $c, c + 1, \dots, d$. Per indicare l' i -esima componente di M (i -esima riga della matrice), si usa la notazione:

$M[i] \quad a \leq i \leq b$ (11.28)

mentre:

$M[i][j] \quad a \leq i \leq b, \quad c \leq j \leq d$ (11.29)

indica una componente elementare (della riga i) di tipo T . Di solito, al posto di (11.27) e di (11.29) si usano le seguenti abbreviazioni:

var M : *array* [$a..b, c..d$] of T (11.30)
 $M[i,j]$

Esempio 11.6. Moltiplicazione di due matrici: date le due matrici a valori reali A ($m p$) e B ($p n$), calcolare la matrice prodotto C ($m n$) definita da:

$$C_{ij} = \sum_{k=1}^p A_{ik} * B_{kj} \quad \text{per } i = 1, \dots, m \text{ e } j = 1, \dots, n. \quad (11.31)$$

Si può usare direttamente il programma (11.17):

```

var i: 1..m; j: 1..n; k: 1..p; s: real;
A: array [1..m, 1..p] of real;
B: array [1..p, 1..n] of real;
C: array [1..m, 1..n] of real;
begin {assegnamento dei valori iniziali ad A e B}
  for i := 1 to m do
    for j := 1 to n do
      begin s := 0;
        for k := 1 to p do s := s + A[i, k] * B[k, j];
        C[i, j] := s
      end
  end.
end. .

```

(11.32)

Il programma (11.32) è un esempio di più ripetizioni innestate l'una dentro l'altra. Poiché programmi di tal tipo comportano sempre un tempo di calcolo elevato, è utile una più attenta analisi da questo punto di vista. È facile vedere che l'istruzione "for $j \dots$ " viene eseguita m volte, l'istruzione "for $k \dots$ " viene eseguita $m \cdot n$ volte e l'assegnamento " $s := s + \dots$ " viene eseguito $m \cdot n \cdot p$ volte. Nell'ipotesi che n, m, p siano molto elevati ($\gg 1$), il numero di assegnamenti a s (e quindi il tempo di calcolo) aumenta in modo sproporzionale. Ciò dimostra come, nei programmi con più ripetizioni *innestate*, l'istruzione di ripetizione più *interna* dev'essere scelta in modo che il relativo tempo di calcolo sia il più piccolo possibile. Il tempo di calcolo per la moltiplicazione cresce come n^3 , assumendo che sia $m = n = p$.

11.2. Problemi

11.1. Sia data la matrice

$$A = \begin{pmatrix} 2 & 1 & 3 \\ 3 & 3 & 1 \\ 1 & 2 & 1 \end{pmatrix}$$

a) Eseguire l'istruzione

```

for i := 1 to 3 do
  for j := 1 to 3 do C[i, j] := A[A[i, j], A[j, i]]

```

(11.33)

Quali sono i valori del risultato C ?

- b) L'ordine nel quale sono scelti gli indici i e j gioca un qualche ruolo?
- c) Sostituire in (a) la variabile C con la variabile A ed eseguire il nuovo programma. Quali sono i valori del nuovo risultato A ?

d) Ripetere il punto c) scegliendo la successione inversa delle coppie di indici (i, j) :

(3, 3), (3, 2), ..., (1, 2), (1, 1)

Paragonare i risultati ottenuti con quelli del punto c).

11.2. Verificare la seguente versione del programma di ricerca binaria:

```

i := m; j := n;
repeat k := (i + j) div 2;
  if A[k] ≤ x then i := k + 1;
  if A[k] ≥ x then j := k - 1
until i > j

```

(11.34)

Confrontare il numero dei paragoni (test) necessari con quello del programma (11.8); si osservi che la condizione di terminazione è più semplice nel programma (11.34).

11.3. Una matrice a valori complessi è rappresentata mediante due matrici a valori reali: $Z = X + iY$. Scrivere un programma che calcoli la parte reale X e la parte immaginaria Y del prodotto di due matrici complesse (A, B) e (C, D):

$$X + iY = (A + iB) * (C + iD) \quad (11.35)$$

Suggerimento:

$$(A + iB) * (C + iD) = (AC - BD) + i(AD + BC) \quad (11.36)$$

Si calcolino le tre matrici

$$R = A * D, \quad S = B * C, \quad T = (A + B) * (C - D)$$

da cui

$$X = T + R - S \quad \text{e} \quad Y = R + S$$

Calcolare il numero delle addizioni e delle moltiplicazioni necessarie e paragonarlo con quello corrispondente all'uso diretto della formula (11.35).

11.4. Rappresentare i coefficienti di un polinomio

$$P_n(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n \quad (11.37)$$

come componenti di un array a . Scrivere un programma che calcoli $P_n(x)$ per un dato valore di x . Suggerimento: usare la fattorizzazione di Horner:

$$P_n(x) = (\cdots (a_0x + a_1) * x + \cdots + a_{n-1}) * x + a_n \quad (11.38)$$

11.5. Trovare un programma che calcoli il minimo e il massimo fra n numeri, rappresentati dalla variabile A :

var A: array [1..n] of integer

Suggerimento: è possibile trovare un programma che impieghi al più $3/2 \cdot n$ confronti.

11.6. Data una variabile-array

```
var M: array [1 .. n, 1 .. n] of integer
```

assegnare agli elementi della matrice M i numeri naturali $1, 2, 3, \dots, n^2$, in modo che M rappresenti un *quadrato magico*, cioè tale che:

$$\sum_{k=1}^n M[k, k] = \sum_{k=1}^n M[k, n-k+1] = C \quad (11.39)$$

e che:

$$\sum_{k=1}^n M[i, k] = \sum_{k=1}^n M[k, i] = C \quad i = 1, \dots, n \quad (11.40)$$

dove $C = n/2*(n^2 + 1)$. Si assuma n dispari. Suggerimento: assegnare i numeri $1, \dots, n^2$ alle componenti $M[i, j]$ in successione, partendo da $M[n+1/2, n]$ e aumentando i e j di 1 (modulo n) ogni $n - 1$ passi, decrementando ogni volta j di 1 all' n -esimo passo e lasciando i invariato; a ogni componente si assegna un'unica volta un unico valore.

11.7. Siano date due matrici X e Y con $2n$ righe e $2n$ colonne. Scrivere un programma per calcolare il prodotto di matrici $Z = X * Y$, usando le relazioni seguenti (*Winograd*) per determinare i prodotti scalari:

$$\sum_{k=1}^{2n} x_k * y_k = \sum_{k=1}^n (x_{2k} + y_{2k-1}) * (x_{2k-1} + y_{2k}) \quad (11.41)$$

$$\underbrace{- \sum_{k=1}^n x_{2k} * x_{2k-1}}_{\bar{x}} - \underbrace{\sum_{k=1}^n y_{2k} * y_{2k-1}}_{\bar{y}}$$

Suggerimento: si calcolino i $(2n)^3$ prodotti scalari della forma

$$\sum_{k=1}^{2n} x_{ik} * y_{kj}$$

in tal modo sono necessari solo $2n$ valori di x e di y . Il metodo usuale di moltiplicazione delle matrici necessita sempre di $(2n)^3$ addizioni e moltiplicazioni. Determinare il tempo di calcolo del metodo ora esposto, in funzione di n .

12.1. Concetti e terminologia

Accade molto spesso che una sequenza di istruzioni compaia nella stessa forma in più parti di un medesimo programma. Per risparmiare un inutile lavoro di scrittura, i linguaggi di programmazione prevedono l'uso dei *sottoprogrammi*; permettono cioè di assegnare a una sequenza di istruzioni un nome che può essere utilizzato come abbreviazione della stessa a tutti gli effetti e che può essere inserito al suo posto nel programma. Nella terminologia dell'ALGOL, un sottoprogramma, a cui è stato assegnato un nome, è detto una *procedura*; se il nome rappresenta un risultato, che può essere inserito in un'espressione o in un'istruzione, viene chiamato *funzione*. L'indicazione del nome di un sottoprogramma viene chiamata *indicazione della procedura* (o *della funzione*) corrispondente; l'impiego di una procedura in un programma, viene detto *chiamata della procedura* (quello di una funzione, viene detto *chiamata della funzione*).

La notazione che useremo in seguito, per indicare le procedure e le relative istruzioni, è definita dai diagrammi sintattici dell'appendice A, ed è illustrata nell'esempio 12.1.

Esempio 12.1. Indicazione e chiamata di una procedura: la sequenza di istruzioni

$$t := r \bmod q; \quad r := q; \quad q := t \quad (12.1)$$

può essere denotata dall'indicazione della procedura:

$$\begin{aligned} &\text{procedure } P; \\ &\text{begin } t := r \bmod q; r := q; q := t \text{ end} \end{aligned} \quad (12.2)$$

In tal modo, la sequenza di istruzioni (12.1) può essere sostituita dalla chiamata di procedura

P (12.3)

L'indicazione di una procedura consiste di due parti: il *titolo della procedura* e il *corpo della procedura*. Nel titolo della procedura (prima riga nella 12.2) vengono indicati il nome (l'identificatore) della procedura e altre eventuali variabili; il corpo della procedura (seconda riga nella 12.2) è formato dalla sequenza di istruzioni denotata dal nome della procedura.

Queste semplici convenzioni, per abbreviare i testi dei programmi, non avrebbero grande importanza, se non implicassero anche altri concetti importanti. Infatti, le procedure sono uno dei pochi strumenti importanti per una buona tecnica di programmazione; la padronanza nell'uso delle procedure influisce in modo decisivo sullo stile e sulla qualità del lavoro di un programmatore.

L'uso delle procedure non serve solo per abbreviare il lavoro di scrittura, ma anche, e in modo essenziale, per articolare, suddividere e strutturare un programma in componenti fra loro coerenti. Una struttura adeguata è determinante per la comprensibilità di un programma, soprattutto quando esso è complicato e il testo ha dimensioni che non permettono di scorrerlo con un unico sguardo. Un'articolazione appropriata in sottoprogrammi è indispensabile per una documentazione comprensibile e per una verifica facile; perciò, spesso è utile indicare una sequenza di istruzioni con una procedura (cioè, assegnarle un nome) anche se essa compare in un sol punto del programma e l'introduzione della procedura non porta a un testo più breve. L'estrazione dal programma di una sua parte, può servire a indicare esplicitamente le variabili da essa influenzate e anche a porre in risalto le condizioni che debbono essere soddisfatte per ottenere un certo risultato intermedio; è conveniente inserire queste informazioni, che riguardano il significato e l'effetto della procedura, nel titolo della procedura.

La strutturazione accurata di un programma diventa necessaria e significativa, quando il programma è lungo. Ma, per forza di cose, nelle lezioni e nei testi introduttivi alla programmazione, gli esempi sono relativi a programmi piuttosto brevi; ne segue che non si fa cenno al problema oppure che la necessità di una buona strutturazione può essere giustificata solo con esempi poco significativi. Quindi è bene ribadire qui che un programmatore deve essere in grado di sviluppare programmi complessi, di grandi dimensioni e corretti. Infatti ogni calcolatore dispone, oggi giorno, di un com-

plesso sistema operativo, formato da programmi di molte migliaia di istruzioni; individuare, comprendere e verificare il loro significato, è possibile solo sulla base di una loro suddivisione in parti semplici e coerenti; l'uso delle procedure ha così un ruolo centrale nella tecnica della programmazione.

Altri due concetti fondamentali sottolineano il ruolo delle procedure, dal punto di vista della strutturazione dei programmi. Spesso, certe variabili (solitamente dette *variabili ausiliarie*) vengono impiegate soltanto all'interno di una certa sequenza di istruzioni, mentre non hanno influenza nel resto del programma. La comprensibilità di un programma aumenta in modo essenziale quando questa *localizzazione* di alcune variabili è posta in chiara evidenza. In ogni caso, i *campi di influenza* delle variabili (cioè, dove il loro valore influenza l'esecuzione) devono risultare con chiarezza dalla struttura del programma. La struttura più adeguata a porre in evidenza il *campo di influenza delle variabili locali*, è fornita dalle procedure.

Spesso, accade che una sequenza di istruzioni compaia in parti diverse non esattamente nella stessa forma, ma all'incirca nella stessa forma. Particolare attenzione merita il caso in cui la differenza consiste unicamente nell'uso di operandi diversi e può essere eliminata con una sostituzione sistematica dei nomi degli operandi. In questo caso è possibile estrarre dalle sequenze di istruzioni uno *schema di procedura comune*; gli operandi da sostituire vengono chiamati *parametri della procedura*.

12.2. Il concetto di locale

Se un oggetto — una costante, una variabile, una procedura, una funzione o un tipo — è significativo solo all'interno di una determinata parte del programma, viene chiamato *locale*. Spesso conviene rappresentare questa parte mediante una procedura; gli oggetti locali vengono allora indicati nel titolo della procedura. Dato che le procedure stesse possono essere locali, può accadere che più indicazioni di procedura siano innestate l'una nell'altra.

Esempio 12.2. Dichiarazione di procedura con indicazione delle variabili locali:

```
procedure P;  
  var t: integer;  
  begin t := r mod q; r := q; q := t end (12.4)
```

Nell'ambito della procedura si possono quindi riconoscere due tipi di oggetti: gli oggetti *locali* e gli oggetti *non locali*. Questi ultimi sono oggetti definiti nel programma (o nella procedura) in cui è inserita la procedura (*ambiente* della procedura). Se sono definiti nel programma principale, sono detti *globali*. In una procedura, il campo di influenza degli oggetti locali corrisponde al corpo della procedura. In particolare, terminata l'esecuzione della procedura, le variabili locali saranno ancora disponibili per indicare dei nuovi valori; chiaramente, in una chiamata successiva della stessa procedura, i valori delle variabili locali saranno diversi da quelli della chiamata precedente.

È essenziale che i nomi degli oggetti locali non debbano dipendere dall'ambiente della procedura. Ma, in tal modo, può accadere che un nome *x*, scelto per un oggetto locale della procedura *P*, sia identico a quello di un oggetto definito nel programma ambiente di *P*. Questa situazione però è corretta solo se la grandezza non locale *x* non è significativa per *P*, cioè non viene applicata in *P*. Adotteremo quindi la *regola fondamentale* che *x* denoti entro *P* la grandezza locale e fuori da *P* quella non locale.

Esempio 12.3. Procedura con "conflitto di nomi" (per d):

```
var a, b, d, e: integer; {variabili globali} (12.5)
procedure Multiply; {procedura globale}
  var c, d: integer; {variabili locali}
begin {e := a * b, cf. (7.18)}
  c := a; d := b; e := 0;
  while d ≠ 0 do
    begin if odd(d) then e := e + c;
           c := 2 * c; d := d div 2
    end
  end;
begin {programma principale} a:=5; b:=7; d:=10; Multiply
  {a = 5, b = 7, d = 10, e = 35}
end.
```

Si osservi che il programma principale può essere visto come una dichiarazione di procedura in cui manca soltanto il nome del programma. È conveniente trattare qui ogni programma come una procedura il cui ambiente è il sistema operativo del calcolatore. Gli oggetti standard del sistema operativo (che sono oggetti non locali per il programma) sono fissati rigidamente e i nomi standard possono essere quindi usati per indicare gli oggetti locali del programma, senza creare alcuna difficoltà o ambiguità, pur di non

impiegare nello stesso programma anche gli oggetti standard in questione. In altre parole, il programmatore deve preoccuparsi di evitare conflitti di nomi solo per i nomi degli oggetti standard del sistema operativo che egli intende utilizzare nel programma.

12.3. Parametri di procedura

Una sequenza di nomi da applicare in posti diversi e su operandi differenti viene spesso formulata come una *procedura*; gli operandi in questione vengono trattati come dei *parametri*. I nomi degli operandi vengono indicati nel titolo della procedura e si chiamano *parametri formali*; essi vengono impiegati nel corpo della procedura. Gli oggetti da sostituire al posto dei parametri formali, prima di ogni esecuzione, sono detti *parametri attuali* e devono essere specificati prima di ogni chiamata di procedura o di funzione; il tipo dei parametri attuali deve essere identico a quello dei corrispondenti parametri formali, indicato nel titolo della procedura, dove viene stabilito anche il *modo di sostituzione*. Si distinguono tre diversi modi di sostituzione dei parametri.

- 1) Si calcola il valore del parametro attuale e lo si sostituisce al posto del corrispondente parametro formale. Questo modo è detto *sostituzione per valore* (*value substitution*) ed è applicato nella maggior parte dei casi.
- 2) Il parametro attuale è una variabile; se essa ha un indice, viene valutato il valore dell'indice; la variabile così identificata viene sostituita al posto del parametro formale corrispondente. Questo modo viene chiamato *sostituzione per referenza* (*reference substitution*) e viene impiegato quando il parametro rappresenta un risultato della procedura.
- 3) Il parametro formale viene sostituito dai parametri attuali senza eseguire alcuna valutazione. Questo modo viene detto *sostituzione per nome* (*substitution by name*) ed è usato solo raramente.

Gli effetti dei tre precedenti modi di sostituzione sono illustrati dagli esempi seguenti, dove si analizza l'effetto della chiamata di procedura *p* (*a [i]*) sulla variabile *a*.

Esempio 12.4.

```
var i: integer;
  a: array [1 .. 2] of integer;
procedure P(x: integer);
begin i := i + 1; x := x + 2
end; (12.6)
```

```

begin {programma principale}
  a[1] := 10; a[2] := 20; i := 1;
  P(a[i])
end.

```

- 1) *Sostituzione per valore*: x è una variabile con valore iniziale 10; risultato: $a = (10, 20)$.
- 2) *Sostituzione per referenza*: $x \equiv a[1]$; l'istruzione $x := x + 2$ ora significa $a[1] := a[1] + 2$; risultato: $a = (10, 22)$.
- 3) *Sostituzione per nome*: $x \equiv a[i]$; l'istruzione $x := x + 2$ diventa $a[i] := a[i] + 2$; risultato: $a = (10, 22)$.

Per poter distinguere i tre casi, introdurremo le seguenti *convenzioni*:

- 1) poiché la sostituzione per valore è la più frequente, è quella impiegata automaticamente, qualora non vi siano altre indicazioni esplicite;
- 2) la sostituzione per referenza (sostituzione di una variabile) verrà indicata con la specificazione `var`;
- 3) tralasciamo qui la sostituzione per nome, poiché metodi equivalenti saranno trattati nel paragrafo 13.

È possibile formulare il programma (12.5) secondo queste regole, indicando gli argomenti x e y e il risultato z come parametri della seguente procedura.

Esempio 12.5. Procedura con parametri:

```

var a, b, c, d, e, f: integer;                                (12.7)
procedure Multiply(x, y: integer; var z: integer);
  {x, y, z sono parametri formali}
begin z := 0;
  while x ≠ 0 do
    begin if odd(x) then z := z + y;
           y := 2 * y; x := x div 2
    end
  end;
begin {programma principale}
  a := 5; b := 7; d := 11; e := 13;
  Multiply(a, b, c); Multiply(d - b, e - a, f)
  {a = 5, b = 7, c = 35, d = 11, e = 13, f = 32}
end.

```

Il programma (12.7) mostra che, nel caso della sostituzione per valore, i parametri formali denotano delle variabili locali che non

conservano più alcuna relazione con i parametri attuali, in seguito all'assegnamento dei valori iniziali fatto con la chiamata della procedura. Perciò possiamo stabilire le seguenti *regole-base di scelta del modo di sostituzione*:

- 1) se un parametro rappresenta un argomento — e non un risultato — di una procedura, si sceglie la normale sostituzione per valore;
- 2) se un parametro rappresenta un risultato, occorre usare la sostituzione per referenza.

Poiché la sostituzione per referenza è una pericolosa fonte di errori di programmazione difficili da scoprire, ne accenniamo brevemente a conclusione di questo paragrafo. Il motivo fondamentale degli errori è che una medesima variabile può presentarsi sotto diverse denominazione. A questa possibilità occorre fare particolare attenzione nel caso di variabili strutturate. Il seguente breve programma illustra le possibili conseguenze del fenomeno.

Esempio 12.6. Uso errato della sostituzione per referenza:

```

type matrix = array [1..2, 1..2] of integer;                (12.8)
procedure mult (var x, y, z: matrix);
begin z[1, 1] := x[1, 1] * y[1, 1] + x[1, 2] * y[2, 1];
  z[1, 2] := x[1, 1] * y[1, 2] + x[1, 2] * y[2, 2];
  z[2, 1] := x[2, 1] * y[1, 1] + x[2, 2] * y[2, 1];
  z[2, 2] := x[2, 1] * y[1, 2] + x[2, 2] * y[2, 2];
end

```

Consideriamo le matrici

$$A = \begin{pmatrix} 2 & 1 \\ -1 & 3 \end{pmatrix} \quad \text{e} \quad B = \begin{pmatrix} 3 & -1 \\ 1 & 2 \end{pmatrix}$$

le chiamate $M(A, B, C)$, $M(A, B, A)$ e $M(A, B, B)$ con i medesimi argomenti A e B , danno luogo ai seguenti risultati:

$$C = \begin{pmatrix} 7 & 0 \\ 0 & 7 \end{pmatrix}.$$

$$A = \begin{pmatrix} 7 & -5 \\ 0 & 6 \end{pmatrix}.$$

$$B = \begin{pmatrix} 7 & 0 \\ -4 & 6 \end{pmatrix}.$$

Da questo breve esempio, si deduce che *ogni parametro*, usato in una sostituzione per referenza, *deve essere disgiunto da tutti gli altri*, cioè deve esserne diverso anche nelle sue componenti (per tipi strutturati). Si osservi infine che il programma (12.8) porta sempre a risultati corretti se per x e per y si usa la sostituzione per valore.

12.4. Funzioni e procedure parametriche

Una procedura (o funzione) F può essere usata come parametro di un'altra procedura (o funzione) G , se F dev'essere calcolata durante l'esecuzione di G ; F può inoltre rappresentare differenti funzioni (o procedure), per differenti chiamate in G . Un esempio ben noto è il calcolo dell'integrale G di una funzione F . L'algoritmo di integrazione può essere rappresentato come una funzione G avente F come parametro.

Esempio 12.7. Integrazione con il metodo di Simpson: per calcolare l'integrale

$$s = \int_a^b f(x) dx \quad (12.9)$$

si usa come approssimazione la somma finita di *valori campione*:

$$\begin{aligned} s_k = & \frac{h}{3} (f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + \dots + \\ & + 2f(a+(n-2)h) + 4f(a+(n-1)h) + f(a+nh)) \end{aligned} \quad (12.10)$$

dove $n = 2^k$ e $h = b - a/n$; $n + 1$ è il numero dei valori campione e h è la distanza tra due punti di campionatura adiacenti. L'approssimazione di s si ottiene dalla successione s_1, s_2, s_3, \dots , che converge (se f ha un andamento "buono") nell'ipotesi di usare un'aritmetica precisa (v. fig. 12.1).

Dunque in ogni passo viene raddoppiato il numero dei valori campioni. Per ottenere la somma s_k non è necessario calcolare ogni volta il valore della funzione f in tutti i 2^{k+1} punti di campionatura, basta invece inserire volta per volta i valori campione già ottenuti nel passo precedente, senza calcolarli nuovamente. A tal fine, la somma dei 2^{k+1} valori campione viene spezzata in tre termini:

$$s_k = s_k^{(1)} + s_k^{(2)} + s_k^{(4)} \quad (12.11)$$

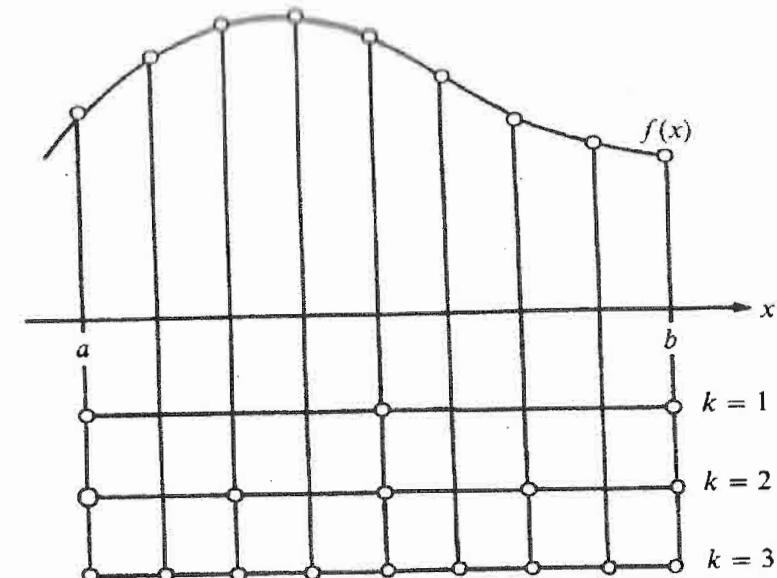


Figura 12.1.

i quali rappresentano le somme dei valori-campione con peso 1, 2 e 4 rispettivamente. È possibile calcolare i termini suddetti con le relazioni di ricorrenza (12.12).

$$s_k^{(1)} = \frac{1}{2} s_{k-1}^{(1)} \quad (12.12)$$

$$s_k^{(2)} = \frac{1}{2} s_{k-1}^{(2)} + \frac{1}{4} s_{k-1}^{(4)}$$

$$s_k^{(4)} = \frac{4h}{3} (f(a+h) + f(a+3h) + \dots + f(a+(n-1)h))$$

per $k > 1$ e

$$s_1^{(1)} = \frac{h}{3} (f(a) + f(b)) \quad (12.13)$$

$$s_1^{(2)} = 0$$

$$s_1^{(4)} = \frac{4h}{3} f\left(\frac{a+b}{2}\right)$$

Utilizzando lo schema di programma (9.21), a partire dalle suddette relazioni si ottiene il programma di integrazione (12.14), che contiene la funzione f come parametro.

```
function Simpson (a, b: real; function f: real): real;
var i, n: integer;
s, ss, s1, s2, s4, h: real;
```

$\{f(x)\}$ è una funzione a valori reali con un unico parametro reale il cui intervallo di definizione è $a \leq x \leq b\}.$

```
begin n := 2; h := (b - a) * 0.5; (12.14)
  s1 := h * (f(a) + f(b)); s2 := 0;
  s4 := 4 * h * f(a + h); s := s1 + s2 + s4;
repeat ss := s; n := 2 * n; h := h/2;
  s1 := 0.5 * s1; s2 := 0.5 * s2 + 0.25 * s4;
  s4 := 0; i := 1;
repeat s4 := s4 + f(a + i * h); i := i + 2
until i > n;
s4 := 4 * h * s4; s := s1 + s2 + s4
until abs(s-ss) < ε;
Simpson := s/3
end
```

La funzione *Simpson* può essere usata come operando in una espressione aritmetica; per esempio:

$$u := \text{Simpson}(0, \pi/2, \sin) \quad (12.15)$$

denota l'assegnamento:

$$u = \int_0^{\pi/2} \sin(x) dx$$

Tuttavia, al posto del terzo parametro, può comparire solo il nome di una funzione e non, per esempio, un'espressione. Così, per calcolare

$$u = \int_0^{\pi/2} \frac{dx}{(a^2 \cos^2 x + b^2 \sin^2 x)^{\frac{1}{2}}} \quad (12.16)$$

per mezzo della funzione *Simpson*, è necessario indicare una seconda funzione *F*:

```
function F(x: real): real;
begin F := 1/sqr(sqrt(sqr(a * cos(x)) + sqr(b * sin(x)))) end (12.17)
```

dopo di che (12.16) può essere denotata dall'istruzione (12.18):

$$u := \text{Simpson}(0, \pi/2, F) \quad (12.18)$$

12.5. Problemi

12.1. Formulare i programmi (7.20), (7.24), (9.17), (9.28), (10.18), (11.25) e (11.32) come procedure o funzioni, con una opportuna scelta dei parametri.

12.2. Si consideri la seguente indicazione di funzione:

```
function f(x, y: real): real;
begin if x ≥ y then f := (x + y)/2 else
  f := f(f(x + 2, y - 1), f(x + 1, y - 2))
end
```

Qual è il valore di $f(1, 10)$?

Come si può rappresentare e calcolare in un modo semplice il valore $f(a, b)$?

12.3. Eseguire i seguenti tre programmi e stabilire i valori dei parametri attuali delle istruzioni di WRITE:

(a) var a, b, c: integer; (12.20)

procedure P(x, y: integer; var z: integer); (12.21)

begin z := x + y + z; write(x, y, z)

end;

begin a := 5; b := 8; c := 3;

P(a, b, c); P(7, a+b+c, a); P(a * b, a div b, c)

end.

(b) var i, j, k: integer;

procedure P(var i: integer);

begin i := i + 1; write(i, j, k)

end {P};

procedure Q(h: integer; var j: integer);

var i: integer;

procedure R;

begin i := i + 1

end {R};

begin i := j;

if h = 0 then P(j) else if h = 1 then P(i) else R;

write(i, j, k)

end {Q};

begin i := 0; j := 1; k := 2; Q(0, k); Q(1, i); Q(2, j)

end.

(c) procedure P(procedure R; b: Boolean);

var x: integer;

```

procedure Q;
begin x := x + 1
end {Q};
begin x := 0; if b then P(Q, false) else R;
  write(x)
end {P};
begin P(P, true)
end.

```

12.4. L'integrale ellittico (v. (12.16))

$$I = \frac{2}{\pi} \int_0^{\pi/2} \frac{dx}{(a^2 \cos^2 x + b^2 \sin^2 x)^{\frac{1}{2}}} \quad (12.23)$$

si può calcolare, con il metodo di Gauss, per mezzo dei valori limiti delle due successioni convergenti

$$s_0, s_1, s_2, \dots \text{ or } t_0, t_1, t_2, \dots$$

definite dalle seguenti relazioni di ricorrenza

$$\begin{aligned} s_i &= (s_{i-1} + t_{i-1})/2 \\ t_i &= \sqrt{s_{i-1} * t_{i-1}} \end{aligned} \quad i > 0 \quad (12.24)$$

Ponendo $s_0 = a$ e $t_0 = b$, vale infatti: $\lim_{i \rightarrow \infty} s_i = I$ (metodo delle medie aritmetica e geometrica);

Rappresentare I , mostrando una opportuna indicazione di funzione.

12.5. Il metodo di integrazione secondo Romberg consiste nell'approssimare l'integrale

$$\int_a^b f(x) dx \quad (12.25)$$

con la successione convergente

$$t_{0,0}, t_{1,0}, t_{2,0}, \dots \quad (12.26)$$

definita dalle relazioni ricorsive

$$t_{m,k} = \frac{1}{4^m - 1} (4^m * t_{m-1,k+1} - t_{m-1,k}) \quad (12.27)$$

per $m > 0$, e dalle condizioni iniziali

$$t_{0,k} = \frac{b-a}{n} (\frac{1}{2}f_0 + f_1 + \dots + f_{n-1} + \frac{1}{2}f_n) \quad (12.28)$$

Stabilire una procedura avente per parametri a, b ed f che calcoli, per mezzo della successione (12.26), l'integrale (12.25) con una certa precisione relativa. Suggerimento: il programma deve calcolare il valore della funzione f una

sola volta in ogni punto di campionatura. A ogni passo il numero dei campioni viene raddoppiato; allo scopo si introduca una variabile di tipo array T tale che

$$T[i] = t_{k-i,i} \quad i = 0, \dots, k \quad (12.29)$$

La ripetizione viene interrotta dopo al più p passi, cioè $k = 0, \dots, p$.

12.6. Uno zero di una funzione reale $f(x)$ è, per definizione, un valore x_0 tale che

$$(f(x_0 - \varepsilon) < 0) = (f(x_0 + \varepsilon) > 0) \quad (12.30)$$

per ε arbitrariamente piccolo. Scrivere una funzione con parametri a, b ed f che calcoli uno zero di f quando valga

$$(f(a) < 0) = (f(b) > 0) \quad (12.31)$$

Suggerimento: dividere ripetutamente in due l'intervallo contenente lo zero (si noti l'analogia con (11.34)).
Quanti valori di f occorre calcolare per a, b, ε assegnati?

13

Trasformazioni della rappresentazione dei numeri

Il concetto astratto di numero non dipende dalla particolare rappresentazione dei numeri. Le operazioni sui numeri possono essere definite per mezzo di un insieme di assiomi, indipendenti dalla loro rappresentazione; se, però, si vuole eseguire un'operazione, bisogna scegliere la rappresentazione numerica in cui poter scrivere i valori degli operandi e del risultato.

La decisione di definire le operazioni sui numeri mediante algoritmi, la cui formulazione è indipendente dalla rappresentazione dei numeri, permette di scegliere la rappresentazione più adatta alle caratteristiche del processore; i calcolatori digitali usano la rappresentazione binaria dei numeri, cioè con due sole cifre.

Per un uomo, di solito abituato fin da bambino a usare la rappresentazione decimale, la rappresentazione binaria non è adatta; per questo i caratteri disponibili sui dispositivi di ingresso e di uscita comprendono tutte le cifre decimali, ed è compito del calcolatore trasformare (prima di eseguire il programma) la *rappresentazione esterna* decimale nella *rappresentazione interna* binaria e viceversa (prima di stampare il risultato).

Nei programmi che seguono, si uscirà un insieme di caratteri *char* (v. anche [8.3]) contenente le cifre '1', ..., '9' e avente le seguenti proprietà:

$$\begin{aligned} ord('1') - ord('0') &= 1 \\ \dots \dots \dots & \\ ord('9') - ord('0') &= 9 \end{aligned} \quad (13.1)$$

Dunque la funzione standard *ord* è già una trasformazione di cifre in valori numerici, ma solo di singole cifre e non di sequenze.

di cifre. Nei paragrafi successivi riporteremo dei programmi che trasformano intere sequenze di cifre in valori di tipo *integer* e viceversa.

Al posto delle funzioni standard *ord* e *char* utilizzeremo le funzioni *num* e *rep* definite nel modo seguente:

function *num* (*x*:char):integer
begin *num* := *ord* (*x*) - *ord* ('0') **end**

e:

```
function rep (x:integer):char;
begin {0 ≤ x ≤ 9} rep := chr (x + ord ('0')) end (13.3)
```

Si userà inoltre la funzione booleana così definita:

13.1. Ingresso (lettura) di numeri interi positivi nella rappresentazione decimale

Si tratta di determinare il valore rappresentato da una successione di cifre posta in un *file* *f*. Si può assumere che la sequenza di cifre termini con un simbolo che non è una cifra e che quindi permette di riconoscere la fine del *file* rappresentante il valore numerico in questione, *num* ($f_0 \dots f_{n-1}$). Tale valore è determinato dalla (13.5), dove *f_i* indica il valore *num* (*f_i*).

$$x = \text{num}(f_0 \dots f_{n-1}) = 10^{n-1} * \bar{f}_0 + \dots + 10^1 * \bar{f}_{n-2} + 10^0 * \bar{f}_{n-1} \quad (13.5)$$

Il programma di trasformazione (13.7) è costituito da una istruzione di ripetizione, che legge una cifra alla volta, e lo stato del file dopo i passi è:

$$f_0 f_1 \dots f_{i-1} \quad f_i \dots f_{n-1} f_n \quad (13.6)$$

7

e $f \uparrow = f_i$. Il valore x corrisponde, passo per passo, al numero rappresentato da f .

procedure *read*(**var** *x*: *integer*); (13.7)

```

repeat  $x := B * x + \text{num}(D\uparrow)$ ;  $\text{get}(D)$ 
until  $\neg \text{digit}(D\uparrow)$ 
end

```

13.2. Uscita (stampa) di interi positivi nella rappresentazione decimale

Si tratta di trasformare un numero intero positivo o nullo in una sequenza di cifre che lo rappresentino in forma decimale. La sequenza di cifre deve essere caricata in un file f . Il punto di partenza sarà ancora la relazione (13.5). In particolare, si noti che, posto $x = \text{num}(f_0 \dots f_{n-1})$, valgono le relazioni

$$\begin{aligned} \text{num}(f_0 \dots f_{n-2}) &= x \text{ div } 10 \\ \text{num}(f_{n-1}) &= x \text{ mod } 10 \end{aligned} \quad (13.8)$$

Ma la successione di cifre ottenuta con successive divisioni per 10 è $f_{n-1}, f_{n-2}, \dots, f_0$; per invertirne l'ordine è allora necessario introdurre uno spazio di memoria intermedio (*buffer*). In tal modo si ottiene il programma:

```

procedure write (e: integer; var f: text);
var x, u: integer; i: 0..n;
a: array [1..n] of integer;
begin {0 ≤ e < 10^n} i := 0; x := e;
repeat  $a_1 * 10^0 + \dots + a_i * 10^{i-1} + \dots * 10^i = e$ 
u := x div 10;
i := i + 1; a[i] := x - 10 * u; x := u
until x = 0;
repeat f\uparrow := rep(a[i]); put(f); i := i - 1
until i = 0;
end

```

13.3. Uscita (stampa) di numeri razionali fratti nella rappresentazione decimale

Si tratta di trasformare un valore razionale x ($0 \leq x < 1$) in una sequenza di cifre decimali tale che:

$$\begin{aligned} x &= \text{num}(f_0 \dots f_k) = \\ &= f_0 * 10^{-1} + \bar{f}_1 * 10^{-2} + \dots + \bar{f}_{k-1} * 10^{-k} = \\ &= \frac{1}{10} \left(\bar{f}_0 + \frac{1}{10} \left(\bar{f}_1 + \dots + \frac{1}{10} \bar{f}_{k-1} \dots \right) \right), \end{aligned} \quad (13.10)$$

dove $\bar{f}_i = \text{num}(f_i)$ (valore numerico rappresentato dalla cifra f_i). Come per il programma (13.7) la successione di cifre viene costruita nell'ordine desiderato (e quindi non è necessario il *buffer* intermedio) moltiplicando in ogni passo il resto ottenuto (nel passo iniziale x stesso) per 10; la cifra desiderata sarà la parte intera del prodotto e il resto, da usare nel passo successivo, sarà la parte decimale. Si ottiene in tal modo il programma:

```

procedure write fraction (e: real; var f: text);
var x: real; i, u: integer;
begin x := e; f\uparrow := '.'; put(f); i := 0;
repeat { $\text{num}(f_0 \dots f_{i-1}) + x * 10^{-i} = e, 0 \leq x < 1$ }
x := 10 * x; u := trunc(x); f\uparrow := rep(u); put(f);
i := i + 1; x := x - u
until i = n
end

```

(Come condizione di terminazione non si prende $x = 0$, bensì il numero n delle cifre desiderate).

13.4. Trasformazione delle rappresentazioni in virgola mobile

Come si è detto nel capitolo 8, la rappresentazione dei numeri frazionari, comunemente usata nei calcolatori, è la rappresentazione in virgola mobile, dove il numero x è rappresentato da una coppia ordinata di interi $(m, e)_B$, tale che

$$x = m * B^e \quad \frac{1}{B} \leq m < 1 \quad (13.12)$$

B è un numero naturale (piccolo), detto *base* della rappresentazione in virgola mobile. Esempi utili a illustrare tale rappresentazione (e il perché del nome *virgola mobile*) sono (con base 10):

$$\begin{aligned} (.34567, 2) &= 34.567 \\ (.34567, 4) &= 2345.7 \\ (.34567, 0) &= .34567 \\ (.34567, -2) &= .0034567 \end{aligned}$$

I calcolatori utilizzano la rappresentazione binaria, per cui è vantaggioso scegliere, come base B , una potenza di 2, $B = 2^k$. In tal modo, aumentare o diminuire di 1 l'esponente e , significa moltiplicare o dividere il coefficiente m per 2^k ; per far ciò, basta spostare la mantissa m di k posizioni a sinistra o a destra (*shift*).

Di solito, nelle documentazioni esterne al calcolatore, si usa la rappresentazione in base 10; pertanto, nelle operazioni di ingresso e di uscita dei numeri reali, si pone il problema di cambiare la rappresentazione dalla base B alla base 10:

$$(m, e)_B \leftrightarrow (m', e')_{10} \quad (13.13)$$

I problemi seguenti riguardano il caso $B=2$, ma possono venir facilmente adattati a un arbitrario valore di B . La variabile b rappresenta l'esponente binario (base 2) e la variabile d quello decimal (base 10). Il metodo più semplice per eseguire la trasformazione desiderata

$$(m, b)_2 \rightarrow (m', d)_{10} \quad (13.14)$$

consiste nella seguente azione: moltiplicare m per 2^b e poi normalizzare, cioè dividere o moltiplicare per 10 fino a quando il risultato m' non verifica la relazione $0.1 \leq m' < 1$. Il numero delle divisioni (o delle moltiplicazioni) fornisce l'esponente d (o $-d$). Ma i risultati intermedi, in tale processo, possono diventare troppo grandi; richiederemo che l'ulteriore condizione $0.1 \leq m \leq 1$ valga durante l'intero processo. Perché ciò sia possibile, le moltiplicazioni per 2 e le divisioni per 10 dovranno essere *alternate*. Dalle considerazioni esposte si ricava il programma (13.15), nel quale i casi $b \geq 0$ e $b < 0$ sono trattati in due modi diversi, per evidenti ragioni.

```
procedure convert (var b, d: integer; m: real);
begin d:=0;
  if b ≥ 0 then
    while b ≠ 0 do
      begin {x = m * 2^b * 10^d, 0,1 ≤ m < 1}
        m:=2*m; b:=b-1;
        if m ≥ 1 then begin m:=m/10; d:=d+1 end
      end
    else repeat {x = m * 2^b * 10^d, 0,1 ≤ m < 1}
      m:=m/2; b:=b+1;
      if m < 0,1 then begin m:=10*m; d:=d-1 end
      until b=0
  end
```

(13.15)

Considerato che i calcolatori hanno una precisione finita, per cui in ogni operazione si può avere un errore di arrotondamento, il programma ora esposto ha il difetto che, quando il numero di moltiplicazioni e di divisioni diventa elevato, tali errori si possono accu-

molare e rendere non attendibile il risultato finale; un secondo inconveniente, sarebbe che le operazioni di ingresso e di uscita dei dati numerici richiederebbero un tempo di calcolo troppo elevato. Una possibilità di ridurre drasticamente il numero delle operazioni necessarie consiste nel memorizzare in una tabella, che sia sempre disponibile in memoria, i valori dei moltiplicatori 2^b , rappresentati da coppie di interi $(u, v)_b$ tali che (v. tab. 13.I):

$$u [b] * 10^{v[b]} = 2^b, \quad 0.1 \leq u [b] < 1 \quad (13.16)$$

TABELLA 13.I.

b	$u [b]$	$v [b]$
-2	0,25	0
-1	0,5	0
0	0,1	1
1	0,2	1
2	0,4	1
3	0,8	1
4	0,16	2

Il programma che lavora sulla base di tale tabella è:

```
m := m*u [b]; d := v [b];
if m ≤ 0,1 then begin m := 10*m; d := d - 1 end
```

(13.17)

Risparmiare il tempo di calcolo a spese dello spazio di memoria (per la tabella), o viceversa, è una scelta che dipende dalla situazione; la scelta della soluzione corretta va fatta in base al costo del tempo di calcolo spese e al costo dello spazio di memoria occupato. È facile vedere che, nell'esempio precedente, lo spazio di memoria necessario per contenere la tabella, diventa eccessivo quando gli esponenti variano in un intervallo ampio. Gli intervalli usuali per b sono dell'ordine: $|b| \leq 2^9$; in tal caso la tabella consiste di 2^{10} elementi. È quindi necessario trovare una soluzione di compromesso, consistente nel memorizzare solo una parte della tabella, in modo che sia possibile calcolare con una certa facilità gli elementi della parte rimanente.

Il programma (13.15) rappresenta una tale soluzione di compromesso e utilizza la sottotabella costituita dagli elementi definiti dalla (13.18).

$$u[b] * 10^{v[b]} = 2^{b}, \quad 0.1 \leq u[b] < 1 \quad (13.18)$$

Si osservi che questo programma è basato sullo stesso principio sul quale è basato il programma del prodotto esposto nel capitolo 7 (7.20). Ci limiteremo a riportare solo il programma relativo al caso di esponenti positivi:

```
procedure convert (var b, d: integer; m: real);
  var i: integer;
begin {x = m*2^b, b ≥ 0} i := 0; d := 0;
  while b > 0 do
    begin {m*10^d*2^b*2^i = x}
      if odd(b) then begin m := m*u[i]; d := d + v[i];
        Normalise
      end;
      b := b div 2; i := i + 1
    end
  end
end
```

(13.19)

13.5. Problemi

13.1. Formulare due procedure, di cui:

- a) la prima legge da un file d'ingresso un numero (positivo o negativo) scritto nella rappresentazione decimale (con parte intera, punto decimale e parte frazionaria), lo trasforma, e lo assegna alla variabile reale v (parametrica) (v. 13.5);
- b) la seconda rappresenti il valore di un parametro reale x come una successione di cifre decimali con punto decimale, posta in un file d'uscita. (v. (13.9) e (13.11)).

13.2. Formulare una procedura di conversione analoga alla (13.15), che esegua la conversione:

$$(m, d)_10 \rightarrow (m', b)_2$$

13.3. Formulare una procedura di conversione analoga alla (13.19), che accetti tanto valori positivi dell'esponente b , quanto valori negativi.

13.4. Costruire un programma che calcoli i valori esatti di $1/n$ per $n = 2, 3, \dots, 50$ e li carichi come sequenze di cifre in un file d'uscita. Inoltre:

- a) ogni successione di cifre deve terminare non appena il primo periodo della parte decimale compare per intero;
- b) per individuare le cifre del periodo deve essere inserito uno spazio bianco davanti alla prima di esse.

Il file d'uscita corrispondente alle precedenti richieste sarà:

```
.5 0
.3
.25 0
.2 0
.1 6
.142857
.....
.02 0
```

Suggerimento: si costruisca una prima procedura, che risolva solo il punto

- (a). Inoltre non si impieghino grandezze reali.

111

sola volta in ogni punto di campionatura. A ogni passo il numero dei campioni viene raddoppiato; allo scopo si introduca una variabile di tipo array T tale che

$$T[i] = t_{k-i,i} \quad i = 0, \dots, k \quad (12.29)$$

La ripetizione viene interrotta dopo al più p passi, cioè $k = 0, \dots, p$.

12.6. Uno zero di una funzione reale $f(x)$ è, per definizione, un valore x_0 tale che

$$(f(x_0 - \varepsilon) < 0) = (f(x_0 + \varepsilon) > 0) \quad (12.30)$$

per ε arbitrariamente piccolo. Scrivere una funzione con parametri a, b ed f che calcoli uno zero di f quando valga

$$(f(a) < 0) = (f(b) > 0) \quad (12.31)$$

Suggerimento: dividere ripetutamente in due l'intervallo contenente lo zero (si noti l'analogia con (11.34)).

Quanti valori di f occorre calcolare per a, b, ε assegnati?

14

Elaborazione di testi mediante le strutture "file" e "array"

Nel presente capitolo tratteremo alcuni problemi, che consistono nell'esame e nella trasformazione di configurazioni di simboli; essi sono esempi di problemi tipici della programmazione, e costituiscono degli utili esercizi di applicazione dei concetti introdotti nei capitoli precedenti.

14.1. Delimitazione della lunghezza delle linee di un *textfile*

Un *textfile* è formato da sottosequenze di caratteri, delimitate da spazi bianchi (simboli *blank*) o da simboli di fine-linea. Ogni sottosequenza, racchiusa fra due *blank* consecutivi, è detta *parola*; il numero dei simboli in essa contenuti viene detto *lunghezza della parola*. Ogni sottosequenza, racchiusa fra due simboli di fine-linea consecutivi, è detta *linea*; il numero dei simboli in essa contenuti è detto *lunghezza della linea*. Si voglia costruire un programma che legge un *file f* e che genera un *file g*, contenente la medesima sequenza di parole e formato da linee di lunghezza non superiore a un valore limite *Lmax*. Il numero totale dei simboli in *f* e in *g* deve essere lo stesso e, chiaramente, per ogni parola deve valere $p \leq p_{max} \leq L_{max}$ (p = lunghezza della parola). Inoltre, supponiamo che *f* contenga almeno una parola e, di conseguenza, almeno una riga.

Da tale impostazione del problema, segue che *g* deve essere una copia di *f*, nella quale certi spazi (*blank*) vengono sostituiti da simboli di fine-riga e viceversa.

È chiaro che l'aggiunta di una parola, in *g*, non può avvenire senza una memorizzazione intermedia di alcuni caratteri. Infatti, le parole non possono essere divise, per cui è lecito aggiungere una parola solo se vi è ancora lo spazio sufficiente nella linea di *g*. Per-

ciò occorre conoscere la lunghezza della parola da aggiungere: solo dopo averla letta completamente, si può decidere se bisogna introdurre in *g* un *blank*, oppure un simbolo di fine-riga; eseguita questa operazione, si può aggiungere la parola in esame al *file g*. È quindi necessario introdurre un *buffer* per la memorizzazione intermedia delle parole (il *buffer* dovrà quindi contenere *pmax* simboli).

Il *file f* è formato da una sequenza di parole e di separatori, per cui la struttura appropriata per il programma è la ripetizione di un'istruzione *I*, che contenga le istruzioni per leggere dal *file f* una parola, seguita dal corrispondente separatore, e per trascriverla nel *file g*, preceduta da un separatore (v. fig. 14.1).

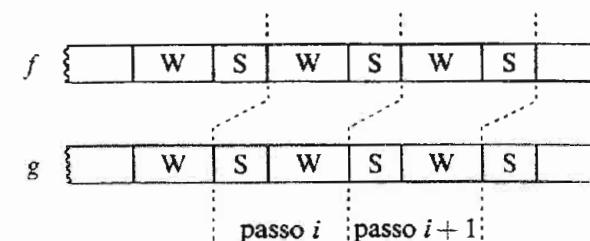


Figura 14.1.

Il caso più generale in cui a una parola possono seguire più separatori può essere ricondotto al più semplice schema della figura 14.1, considerando lecite anche parole di lunghezza 0. Da questo punto di vista, ogni separatore immediatamente successivo a un altro separatore viene considerato una parola di lunghezza 0, ottenendo in tal modo una sequenza alternata di parole e di separatori.

Tornando al programma, esso si ottiene inserendo l'istruzione *I* nello schema generale (10.13); ma, prima, conviene introdurre due procedure, così definite:

1) *readword*: legge una parola dal *buffer B* (memoria intermedia); dopo l'esecuzione di *readword*, la variabile *ch* contiene l'ultimo simbolo letto, successivo alla parola, e *p* è uguale alla lunghezza della parola;

2) *writeword*: aggiunge al *file g* la parola contenuta nel *buffer B*. *L* indicherà la lunghezza attuale dell'ultima linea di *g* in costruzione; perciò, *L* dev'essere incrementata di 1 ogni volta che viene aggiunto un simbolo a *g*. Se (per semplificare il programma) si introducono al posto di *f* e di *g* i *file standard input* e *output*, il programma risulta:

```

begin readword;
  if  $L + w < Lmax$  then
    begin write(' ');  $L := L + 1$ 
  end else
    begin writeIn;  $L := 0$ 
  end;
  writeword
end

```

(14.1)

Prima di specificare nei dettagli le due procedure ausiliarie, è necessario delineare la situazione che si presenta all'inizio e alla fine della elaborazione del *file*. Dalla figura 14.2 risulta evidente che, tanto all'inizio come alla fine, debbono venir introdotte alcune istruzioni non collegate all'istruzione *I*, che non vanno perciò inglobate nell'iterazione.

```

var w: 0 .. wmax; {lunghezza di parola}
 $L: 0 .. Lmax$ ; {lunghezza di riga}
ch: char; {ultimo simbolo letto}
Z: array [1 .. wmax] of char; {buffer}
procedure readword;
begin w := 0;
  while (ch ≠ '') ∧ (ch ≠ eol) do
    begin w := w + 1; Z[w] := ch; read(ch)
    end
  end;
procedure writeword;
  var i: 1 .. wmax;
begin for i := 1 to w do write(Z[i]);  $L := L + w$ 
end;
begin  $L := 0$ ; read(ch); readword; writeword;
  while ¬eof(input) do
    begin read(ch); readword;
      if  $L + w < Lmax$  then
        begin write(' ');  $L := L + 1$ 
      end else
        begin write(eol);  $L := 0$ 
      end;
      writeword
    end;
    write(eol)
end.

```

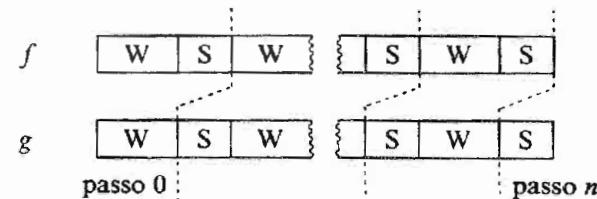
(14.2)


Figura 14.2.

Nel passo 0 viene letto un simbolo di separazione in più rispetto ai simboli scritti, mentre nel passo finale viene scritto un simbolo di separazione in più rispetto ai simboli letti. Il programma finale è riportato in (14.2). Si può verificare che le due procedure ausiliarie sono corrette anche nel caso di parole vuote.

Dall'esempio precedente si deducono le considerazioni seguenti:

- 1) se si vuole trasformare una sequenza di configurazioni (contenuta in un *file* *f*) in una sequenza di configurazioni corrispondenti, bisogna innanzitutto formulare il passo di trasformazione di una singola configurazione; l'istruzione così ottenuta va inserita in una istruzione **while** o **repeat**; inoltre, all'inizio e alla fine del *file* (o della relativa elaborazione), si verificano delle condizioni particolari, che vanno elaborate con due istruzioni distinte, prima e dopo la ripetizione;
- 2) per esprimere delle azioni, che hanno una loro unità interna, si introducono delle procedure corrispondenti; la loro specificazione può esser rimandata a una fase successiva; anche la struttura dei dati, impiegati da tali procedure, può essere precisata in una fase successiva (non appena la specificazione delle procedure le renderà necessarie).

14.2. Modifica (editing) di una linea in un testo

Consideriamo il seguente problema: siano dati una linea *z* formata dalla successione di simboli

$$z = z_1 z_2 \dots z_n \quad n > 0$$

un sostituendo *x*, assegnato da

$$x = x_1 x_2 \dots x_k \quad k > 0 \quad (14.3)$$

e un sostituto (di *x*) *y*, assegnato da:

$$y = y_1 y_2 \dots y_m \quad m \geq 0$$

Si voglia rimpiazzare nella linea z il sostituendo x con il sostituto y . Una variabile booleana q rappresenterà il valore di verità della proposizione x è contenuto in z . Nel caso normale tale valore dev'essere *true*. Può inoltre accadere che x compaia più volte in z : per definire in modo univoco tale situazione, si assume che sia specificata una *posizione di linea p*; il sostituendo x viene cercato a partire da tale posizione, proseguendo da sinistra verso destra, e viene sostituito non appena incontrato. Si assegna poi alla posizione di linea p l'indice del simbolo successivo alla sequenza sostituita. Se non si trova alcun x fra p e la fine-linea, la ricerca prosegue fra l'inizio di linea e la posizione p (ricerca circolare).

Esempi: $x = AB$, $y = UVW$

z prima	e dopo la sostituzione
EFABGH	EFUVWGH
↑	↑
EFABCDABCD	EFABCDUVWCD
↑	↑
EFABCDABCD	EFUVWCDABCD
↑	↑

Questo metodo di sostituzione è spesso usato, per esempio, dai calcolatori collegati a un terminale, che memorizzano i dati e i programmi dell'utente nella forma di *textfield*.

Il problema posto può essere diviso in due passi distinti:

- 1) la ricerca del sostituendo x nella linea z ($z = axb$);
- 2) la sostituzione di y a x ($z = ayb$).

Passo 1: individuare un indice i tale che

$$x_j = z_{i+j-1} \quad \text{per tutti } i, j = 1, \dots, k \quad (14.4)$$

Il programma che assegna alla variabile q il valore di verità di "è stato trovato x e i è l'indice cercato" è:

```
i := p;
repeat {Q(i)}
  q := x = (zi ... zi+k-1);
  i := i + 1; if i > n then i := 1
until q V (i = p) (14.5)
```

dove

$$\begin{aligned} Q(i): (x_1 \dots x_k) &\neq (z_j \dots z_{j+k-1}) \\ \text{per tutti } i, j = &\begin{cases} p \dots i-1 \text{ se } i \geq p \\ p \dots n-1 \dots i-1 \text{ se } i < p \end{cases} \end{aligned}$$

Il confronto di due sequenze di simboli può essere ulteriormente suddiviso in una successione di confronti di singoli simboli:

```
j := 1;
repeat {P(j)}
  q := (x[j] = z[i+j-1]); j := j + 1
until not q V (j > k) (14.6)
```

dove: $P(j): Q(i) \wedge (x_h = z_{i+h-1})$ per ogni $h = 1 \dots j-1$.

Tuttavia questa versione è ancora incompleta, poiché nel caso in cui

$$\begin{aligned} x_h &= z_{i+h-1} \quad \text{per tutti gli } h = 1 \dots h' \\ i+h'-1 &= n \text{ e } h' < k \end{aligned} \quad (14.7)$$

vengono confrontati i simboli x_{h+1} e z_{n+1} ; ma z_{n+1} non esiste e non è definito; inoltre, nella impostazione del problema, non si è deciso se interrompere in questo caso con risultato negativo, oppure se proseguire il confronto dall'inizio di linea (circolarmente).

Sceglieremo qui la prima alternativa:

```
j := 1;
repeat if i + j > n then q := false else
  q := x[j] = z[i+j-1]; j := j + 1
until not q V (j > k) (14.8)
```

Spesso, per semplificare questa istruzione, si usa una soluzione che rende inutile il test esplicito di fine-linea; tale soluzione consiste nel *marcare* la fine-linea con un simbolo che non può comparire nel sostituendo (per esempio con *eof*). In tal modo, si può applicare il programma (14.6) al posto di (14.8).

Passo 2: al posto dei simboli $z_i \dots z_{i+k-1}$ vanno ora sostituiti i simboli $y_1 \dots y_m$. A tal fine, la sequenza di simboli $z = axb$ viene prima trasformata in $z' = ab$, spostando b di k posizioni verso sinistra; in seguito, essa può essere portata nella forma desiderata, $z'' = ayb$, spostando b di m posizioni verso destra e inserendo y

nello spazio intermedio lasciato libero. Tale procedimento può essere semplificato spostando b una sola volta, distinguendo i due casi:

- 1) $m < k$: il sostituto è più breve del sostituendo; b viene spostato di $k - m$ posizioni verso sinistra;
- 2) $m > k$: il sostituto è più lungo del sostituendo; b viene spostato di $m - k$ posizioni verso destra.

Il programma complessivo, costituito da un passo di ricerca e da un passo di sostituendo, è riportato per esteso in (14.9):

```

procedure Substitute;
  var i,j: 1..n+1; q:Boolean; d:integer;
begin {passo 1: arco x nella linea z}
  i := p;
  repeat j := 1;
    repeat q := (x[j] = z[i+j-1]); j := j + 1
    until not q or (j > k);
    i := i + 1; if i > n then i := 1
  until q or (i = p);
  if q then
    begin {passo 2: sostituisce y a x}
      d := m - k; p := i;
      if d < 0 then
        begin j := p + k;
          while j ≤ n do
            begin z[j+d] := z[j]; j := j + 1
          end
        end else
        if d > 0 then
          begin j := n;
            while j ≥ p + k do
              begin z[j+d] := z[j]; j := j - 1
            end
          end;
        n := n + d; j := 1;
        while j ≤ m do
          begin z[p] := y[j]; p := p + 1; j := j + 1
        end
      end
    end
  end
end
(14.9)

```

14.3. Riconoscimento di linguaggi regolari

Un problema che si presenta di frequente nella stesura di programmi che interpretano o che traducono un testo, è il riconoscimento di particolari configurazioni di caratteri in un *textfield*. Per *configurazione* si intende una sequenza di caratteri costruita secondo delle regole assegnate. La complessità di un algoritmo, che verifica se una data sequenza di simboli soddisfa alle regole, dipende chiaramente dalla loro complessità. Le regole, che definiscono le configurazioni lecite, in genere vengono dette *sintassi*; il problema del riconoscimento è detto invece *analisi sintattica*.

Nel seguito esporremo una classe di regole (cioè uno schema di regole) che generano configurazioni di simboli riconoscibili da algoritmi estremamente semplici, e caratterizzate da una certa regolarità e semplicità di struttura; perciò, le regole si chiamano anche *espressioni regolari* e gli insiemi delle sequenze di simboli, da esse generate, si chiamano *linguaggi regolari*. Introduciamo innanzitutto la seguente notazione:

- 1) i caratteri minuscoli indicano simboli di un *vocabolario base* V (ogni configurazione sarà una sequenza di simboli di V);
- 2) i caratteri maiuscoli indicano le espressioni regolari, o, alternativamente, gli insiemi delle sequenze da esse definite;
- 3) i caratteri greci minuscoli indicano arbitrarie sequenze di simboli dell'alfabeto V ; λ indica la sequenza vuota;
- 4) l'insieme di tutte le sequenze ottenute come accostamento di una sequenza di A con una sequenza di B viene indicato come il prodotto AB :

$$AB = \{\alpha\beta \mid \alpha \in A \text{ e } \beta \in B\} \quad (14.10)$$

- 5) l'unione dei due insiemi A e B viene indicata con $A|B$:

$$A|B = \{\gamma \mid \gamma \in A \text{ o } \gamma \in B\} \quad (14.11)$$

- 6) l'insieme delle sequenze costituite dall'accostamento di un numero arbitrario di elementi di A viene indicato con A^* :

$$A^* = \{\lambda\} \cup A \cup AA \cup AAA \cup \dots$$

Le espressioni regolari sono definite secondo lo schema seguente:

- 1) ogni simbolo dell'alfabeto base (vocabolario) è una espressione regolare;

- 2) ogni prodotto di due espressioni regolari AB è un'espressione regolare;
- 3) ogni unione di due espressioni regolari $A|B$ è un'espressione regolare;
- 4) se A è un'espressione regolare, anche A^* è un'espressione regolare;
- 5) sono espressioni regolari solo quelle ottenute dalle regole 1) ÷ 4).

Esempi di espressioni regolari sull'alfabeto base $V = \{a, b, c, d, e, f\}$ con gli insiemi di simboli corrispondenti sono:

1) a	$\{a\}$
2) $ab bc$	$\{ab, bc\}$
3) ab^*c	$\{ac, abc, abbc, abbbc, \dots\}$
4) $a((b c)a)^*$	$\{a, aba,aca,ababa,abaca,acaba,acaca\}$
5) $a(b c)^*d$	$\{ad, abd, acd, abcd, acbd, accd, \dots\}$

(14.12)

Il problema posto, del riconoscimento di configurazioni di simboli, può ora essere formulato nel modo seguente:

sia A un insieme di sequenze descritto da un'espressione regolare E ; costruire un algoritmo di riconoscimento $\mathcal{P}(E)$, che stabilisce, per ogni sequenza α , di simboli del vocabolario V ($\alpha \in V^*$), se α appartiene all'insieme A ($\alpha \in A$?).

L'idea più immediata, è di sfruttare la struttura dell'espressione regolare, come base per determinare la struttura dell'algoritmo di riconoscimento. Un metodo sistematico per eseguire tale traduzione deve logicamente posare su una regola di trasformazione degli elementi strutturali delle espressioni regolari. Un'immagine visiva delle regole-base, utile da questo punto di vista, è fornita dal loro dia-

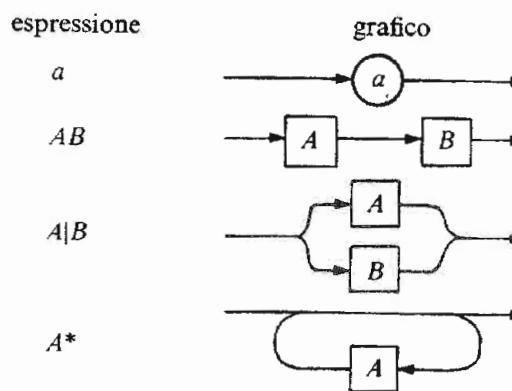


Figura 14.3.

gramma sintattico (v. fig. 14.3); dove a sua volta,

indica il diagramma sintattico della sotto-espressione A .

Per esempio, l'applicazione di queste corrispondenze all'espressione 4 in (14.12) porta al grafico della figura 14.4.

Un algoritmo di riconoscimento adeguato deve decidere se la

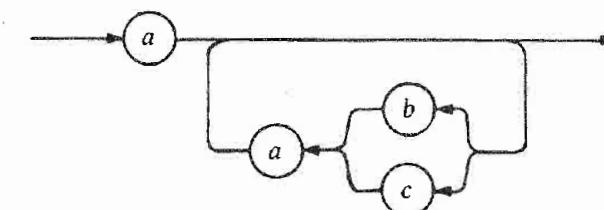


Figura 14.4.

sequenza di simboli letta sino a quell'istante è accettabile, esaminando un certo numero (il più piccolo possibile) degli ultimi simboli letti; un algoritmo di riconoscimento, basato su un diagramma sintattico, decide il cammino da seguire confrontando gli ultimi simboli letti con i prossimi simboli del diagramma (se nessun cammino corrisponde ai simboli letti, la sequenza non è riconosciuta). In particolare, per un linguaggio regolare, si può sempre costruire un algoritmo di riconoscimento che sceglie il cammino, confrontando, di volta in volta, il prossimo simbolo del diagramma con il simbolo appena letto. Per verificare questa affermazione, bastano delle semplici considerazioni:

- a) il problema di decidere il cammino da seguire, si presenta solo quando si incontra una ramificazione; per poter decidere univocamente, basta che *ogni ramo inizi con un simbolo diverso*;
- b) ogni espressione regolare (e il diagramma sintattico corrispondente) può essere trasformata in un'espressione (in un diagramma) equivalente, che soddisfa alla condizione precedente, usando le seguenti regole di trasformazione:

$$1. \quad aA|aB = a(A|B) \quad (14.13)$$

$$2. \quad (aA)^*aB = aB|aA \quad (aA)^*B = a(\lambda|A(aA)^*)B = a(Aa)^*B \quad (14.14)$$

$$3. \quad (aA)^*(aB|C) = (aA)^*aB|(aA)^*C = a(Aa)^*B|(aA)^*C \quad (14.15)$$

Esempio:

$$\begin{aligned} (abc|abd)(efgf)^*ef &= a(bc|bd)(efgf)^*ef = \\ &= ab(c|d)(efgf)^*ef = ab(c|d)e(fgfe)^*f = \\ &= ab(c|d)ef(gfes)^* \end{aligned} \quad (14.16)$$

Dopo aver trasformato l'espressione regolare (o il diagramma corrispondente) nella forma desiderata, detta *deterministica*, è possibile tradurla direttamente in un programma. Le regole di traduzione, che trasformano una espressione regolare deterministica in un programma di riconoscimento, sono basate sulle tre *convenzioni* seguenti:

1) Il programma di riconoscimento ha la forma

```
var ch: char; {ch = ultimo simbolo letto}
begin read(ch); P(X)
end
```

 (14.17)

dove la sequenza di simboli da riconoscere è posta nel *file input* e $P(x)$ rappresenta il programma ottenuto dalle regole di traduzione;

2) l'istruzione *next* è definita come abbreviazione dell'istruzione *read(ch)*;

3) l'istruzione *text(x)* esamina se $ch = x$ (se $ch \neq x$, la sequenza di simboli non appartiene all'insieme indicato; in tal caso l'istruzione può provocare, per esempio, l'interruzione del processo di riconoscimento).

Le regole di traduzione sono le seguenti:

$$1. P(a) = \text{test}(a); \text{read}(ch) \quad (14.18)$$

$$2. P(AB) = P(A); P(B) \quad (14.19)$$

$$3. P(aA|B) = \text{if } ch = a \text{ then} \\ \quad \begin{aligned} &\text{begin read}(ch); P(A) \text{ end} \\ &\text{else } P(B) \end{aligned} \quad (14.20)$$

$$4. P((aA)^*) = \text{while } ch = a \text{ do} \\ \quad \begin{aligned} &\text{begin read}(ch); P(A) \text{ end} \end{aligned} \quad (14.21)$$

È poi necessario considerare un'eccezione all'ultima regola, quando all'espressione A non seguono altre espressioni. In tal caso la regola dev'essere:

$$P(A^*) = \text{while } \neg \text{eof}(\text{input}) \text{ do } P(A). \quad (14.22)$$

Inoltre è spesso utile l'uso della regola ulteriore:

$$5. \quad \begin{aligned} &P(aA(bA)^*) = \\ &\quad \text{test}(a); \text{repeat next; } P(A) \text{ until } ch \neq b \\ &\quad \text{test}(a); \text{repeat next; } P(A) \text{ until } ch \neq b \end{aligned} \quad (14.23)$$

Esempi:

$$1. \quad \begin{aligned} &P(aa^*b) = \\ &\quad \text{test}(a); \text{next;} \\ &\quad \text{while } ch = a \text{ do next;} \\ &\quad \text{test}(b); \text{next} \end{aligned} \quad (14.24)$$

(Osservare che il programma di riconoscimento completo si ottiene inserendo tale sequenza di istruzioni nello schema (14.17)).

$$2. \quad \begin{aligned} &P(a(bc|bd)^*e) = P(a(b(c|d))^*e) = \\ &\quad \text{test}(a); \text{next;} \\ &\quad \text{while } ch = b \text{ do} \\ &\quad \begin{aligned} &\text{begin read}(ch); \text{if } ch = c \text{ then next else} \\ &\quad \quad \begin{aligned} &\text{begin test}(d); \text{next} \\ &\text{end} \\ &\text{end;} \\ &\text{test}(e); \text{next} \end{aligned} \end{aligned} \quad (14.25)$$

$$3. \quad \begin{aligned} &P(eab(cb)^*(dab(cb)^*)^*e) = \\ &\quad \text{test}(e); \\ &\quad \text{repeat next; test}(a); \\ &\quad \begin{aligned} &\text{repeat next; } \\ &\quad \quad \begin{aligned} &\text{test}(b); \text{next} \\ &\text{until } ch \neq c \\ &\text{repeat next; } \\ &\quad \quad \begin{aligned} &\text{test}(d); \text{next} \\ &\text{until } ch \neq d; \\ &\text{test}(e); \text{next} \end{aligned} \end{aligned} \end{aligned} \quad (14.26)$$

Nota: l'espressione ha la forma $eA(dA)^*$ (dove $A = aB(cB)^*$ e $B = b$, per cui si può impiegare due volte la regola di traduzione 5).

L'istruzione *text(x)* non ha alcun effetto se $ch = x$. Se invece si verifica il caso contrario, la sequenza contenuta nel *file input* non appartiene all'insieme definito dall'espressione regolare e il processo di riconoscimento può venire interrotto con risultato negativo. La situazione, in cui la prosecuzione del processo di cal-

colo non ha più senso, può essere risolta con una opportuna *istruzione di salto*. Per esempio, la procedura *test* può essere scritta nella forma (14.27):

```
procedure test (x: char);
begin if ch ≠ x then goto 13
end
```

(14.27)

dove "goto *M*" rappresenta l'istruzione di salto e *M* è la cosiddetta *etichetta* (label) dell'istruzione *I*, alla quale è indirizzato il salto.

L'associazione di un'etichetta a *I* viene indicata nel modo seguente:

13:*I* (14.28)

(a *I* è associato il *label* 13).

Il riconoscimento di configurazioni, nella programmazione, riguarda un campo di applicazioni piuttosto ristretto; qui, è stato esposto solo come introduzione al più generale e importante capitolo della *elaborazione di configurazioni*. Se l'insieme della configurazione da elaborare è un linguaggio regolare, esso può essere descritto da un'espressione regolare, la cui struttura serve a riprodurre anche la struttura dei passi di elaborazione. A tal fine, si introduce la seguente *definizione*: sia *E* un'espressione regolare e Ω indichi l'elaborazione da eseguire, dopo aver riconosciuto una configurazione $\alpha \in E$; chiameremo *E* Ω *espressione (regolare) di elaborazione*.

La regola di traduzione di una espressione di elaborazione in un segmento di programma è

$\mathcal{P}(E\Omega) = \mathcal{P}(E); \Omega$ (14.29)

Un caso particolare di elaborazione, che si presenta abbastanza spesso, è la *traduzione* di una sequenza di simboli; in questo caso, Ω indica la sequenza di simboli da fornire in uscita, dopo il riconoscimento di un ingresso $\alpha \in E$, e la regola di trasformazione diventa:

$\mathcal{P}(E\Omega) = \mathcal{P}(E); \text{write } (\Omega)$ (14.30)

La teoria delle espressioni regolari e degli automi di traduzione garantisce che il linguaggio ottenuto in tal modo è ancora un linguaggio regolare.

Una applicazione è fornita dal seguente esempio di elaborazione: sia

$E = (0|1)(0|1)^*$ (14.31)

la corrispondente espressione di elaborazione avrà la forma:

$$E' = (0\Omega_0|1\Omega_1)(0\Omega_2|1\Omega_3)$$

Se poniamo:

$$\begin{aligned} \Omega_0 &= n := 0 \\ \Omega_1 &= n := 1 \\ \Omega_2 &= n := 2*n \\ \Omega_3 &= n := 2*n + 1 \end{aligned} \quad (14.32)$$

la variabile *n* indicherà il valore rappresentato dalla sequenza di cifre binarie da elaborare.

Il programma di elaborazione risulterà:

```
 $\mathcal{P}(A') =$ 
if ch = '0' then n := 0 else
  begin test ('1'); n := 1
  end ;
next;
while (ch = '0') v (ch = '1') do
  begin if ch = '0' then n := 2*n else n := 2*n + 1;
  next
end ;
```

(14.33)

L'elaborazione dei linguaggi regolari (detti anche *lineari*) è la base per i compilatori dei linguaggi di programmazione. Benché gli usuali linguaggi di programmazione non siano formati da insiemi regolari di sequenze (frasi del linguaggio), possono essere generati con un piccolo ampliamento delle regole di costruzione delle espressioni regolari. L'elenco, esposto sopra, di schemi di programma e di regole di trasformazione delle espressioni in segmenti di programma, aventi una struttura assegnata, è un esempio tipico di come sistemi di elaborazione complessi possono essere costruiti sistematicamente a partire da pochi tipi di elementi base. Abbiamo in tal modo esposto una delle tecniche più importanti, per costruire sistemi di programmazione chiari e di facile verifica.

14.4. Problemi

14.1. Stabilire se i due programmi seguenti risolvono in modo corretto il problema, posto nel paragrafo [14.1], di copiare un *textfile* limitando la lunghezza massima delle linee.

```

(a) var i, L, w: integer; ch: char;
    Z: array [1 .. wmax] of char;
begin L := 0;
repeat w := 0; read (ch);
    while ch ≠ ' ' do
        begin w := w + 1; Z[w] := ch; read(ch)
            end;
    if w > 0 then
        begin if L + w < Lmax then
            begin write (' '); L := L + 1
            end else
            begin writeIn; L := 0
            end;
        for i := 1 to w do write(Z[i]); L := L + w
        end
    until eof (input)
end.

```

(14.34)

```

(b) var i, L, w: integer; ch: char;
    Z: array [1 .. wmax] of char;
begin L := Lmax;
repeat if ch = ' ' then read(ch) else
begin w := 0;
repeat w := w + 1; Z[w] := ch; read(ch)
until ch = ' ';
if L + w < Lmax then
begin write(' '); L := L + 1
end else
begin writeln; L := 0
end;
for i := 1 to w do write(Z[i]); L := L + w
end
until eof(input);
writeln
end.

```

14.2. Si vogliono indicare certi simboli di una linea di stampa di lunghezza L , stampando dei simboli ' \dagger ' nelle posizioni corrispondenti della linea di stampa successiva. Tali posizioni siano assegnate dai valori

$$p_1, p_2, p_3, \dots, p_n \quad n \geq 0, \quad 1 \leq p_i \leq L$$

valga inoltre

$$p_i \leq p_j \text{ per ogni } i < j$$

Esempio: $p = (6, 11, 19, 32)$

Linea 1: Questo problema è un gioco da bambini
Linea 2: ↑↑↑↑↑↑

Esaminare i seguenti cinque tentativi di soluzione. Quali sono corretti? Quali i casi in cui quelli non corretti sbagliano?

```

for i := 1 to L do
begin if i = p[k] then
        begin write('*'); k := k + 1
        end
        else write(' ')
end

```

```

(b) k := 1; p[n + 1] := 0;
    . . .
    for i := 1 to L do
        begin if i = p[k] then
            begin write('*');
            repeat k := k + 1 until i ≠ p[k]
        end
        else write(' ')
    end

```

```

(c) i := 1;
    for k := 1 to n do
        begin repeat write(' ');
           until i = p[k];
           write('*'); i := i + 1
        end

```

```

(d)  $i := 1; k := 1;$ 
    repeat
        while  $i < p[k]$  do
            begin write(' ');  $i := i + 1$ 
            end;
        if  $i = p[k]$  then
            begin write('*');  $i := i + 1$ 
            end;
         $k := k + 1$ 
    until  $k \geq n$ 

```

```

(c) i := 1; k := 1;
    while k ≤ n do
        begin while i < p[k] do
            begin write(' '); i := i + 1
            end;
            while p[k] = i do k := k + 1;
            write('*')
        end

```

14.3. Costruire un programma che conta il numero di simboli distinti contenuto in un assegnato *textfile*. Porre il risultato in una variabile *N* del tipo

```
var N: array [char] of integer
```

e tale che *N* [*c*] rappresenti il numero dei simboli *c* contenuti nel *file*. Si introduca inoltre una variabile *D* che indichi il numero delle componenti di *N* diverse da 0.

14.4. *A* e *B* siano due *array* composti da parole; ogni parola *A_i* e ogni parola *B_i* sia rappresentata da un *array* composto da simboli; inoltre, per ogni *i* = 1 ... *M*, le componenti di *A_i* e di *B_i*

$$A_{i,1} \dots A_{i,m_i} \quad \text{and} \quad B_{i,1} \dots B_{i,n_i} \quad m_i, n_i < N$$

siano dei caratteri e le componenti

$$A_{i,m_i+1} \dots A_{i,N} \quad \text{and} \quad B_{i,n_i+1} \dots B_{i,N}$$

siano spazi bianchi. Sia inoltre assegnato un *textfile input*, costituito da una sequenza di successioni di caratteri (parole) separate da simboli di fine riga e da spazi bianchi.

Costruire un programma che legge il *textfile input* e che genera un *file output*, nel quale tutte le parole uguali a un dato *A_i* siano sostituite dal corrispondente *B_i*.

Suggerimento: usare un *buffer* di memorizzazione intermedia contenente al più *N* simboli. Introdurre una procedura *transmitword*, che paragona l'ultima parola letta e memorizzata nel *buffer*, *Z₁* ... *Z_k*, con tutti gli *A_i*, e che scrive nel *file output* o *B_i* o *Z*. Si prenda in considerazione la seguente procedura (con particolare cautela):

```
procedure Transmitword;
  var i: 0..M; j: 0..N; f: Boolean;
begin i := 0; {Z[k + 1] = ' ', 0 < k < N}
  repeat i := i + 1; j := 0;
    repeat j := j + 1; f := Z[j] ≠ A[i,j]      (14.36)
    until f ∨ (j = k)
  until ∼f ∨ (i = M);
  if f then Emit(Z) else Emit(B[i])
end
```

14.5. Sia dato un *array* *C* di parole chiave. *C_{1,1}* ... *C_{i,n}* siano dei caratteri, mentre *C_{i,n+1}* ... *C_{i,N}* siano spazi bianchi ($1 \leq n_i < N$, $1 \leq i \leq M$). Come nell'esercizio 14.4, sia dato un *textfile input*, costituito da parole separate da simboli di fine linea o bianchi. Ogni linea contenga inoltre al più *L* simboli.

Costruire un programma che generi un *file output*, ottenuto dal *file input* con rotazioni cicliche delle parole di ogni linea, in modo che, in ogni linea, l'inizio di una parola chiave sia in una posizione fissata *k*. Se una linea non contiene nessuna parola chiave, allora essa non verrà scritta nel *file output*, mentre, se contiene più parole chiave, comparirà un numero di volte pari al numero delle rotazioni distinte, che portano una parola chiave nella posizione desiderata.

Usare come *buffer* una variabile di tipo *array* composta da *L* simboli.

14.6. Scrivere un programma, secondo le regole di trasformazione (14.18) + (14.23), per valutare delle espressioni aritmetiche del tipo di quelle eseguibili da una calcolatrice da tavolo. L'alfabeto base sia:

$$V = \{d + - * / =\}$$

La struttura dei testi forniti dal *file input* sia definita dall'espressione regolare

$$(d^+ ((+ | - | * | /) d^+)^*)^+$$

dove *d* rappresenta una qualsiasi cifra decimale e la notazione *B⁺* è un'abbreviazione per *BB**. Per valutazione si intende il calcolo e la stampa del valore dell'espressione, dove il simbolo di egualanza indica semplicemente la fine di un'espressione.

Nella precedente espressione regolare, tutte le operazioni aritmetiche sono considerate dello stesso rango (classe di precedenza) e la valutazione prosegue strettamente da sinistra verso destra; scrivere un secondo programma che valuti le medesime espressioni, dove però gli operatori * e / abbiano la precedenza sugli altri. La corrispondente espressione regolare, adeguata allo scopo, è:

$$(d^+ ((* | /) d^+)^* ((+ | -) d^+ ((* | /)^*)^*)^+$$

Scrivere anche le due espressioni di valutazione (regolari) dalle quali si ottengono i due programmi in questione.

Sviluppo di un programma per passi successivi

Dalle considerazioni fin qui esposte, e in particolare dagli esempi del precedente capitolo, risulta chiaro che la programmazione è un processo di invenzioni e di formulazione di algoritmi, in generale complesso, che richiede attenzione ai dettagli e la conoscenza di tecniche specifiche. Inoltre, solo in rarissimi casi, la soluzione è data da un unico programma; per lo più sono possibili molte soluzioni e la scelta del programma *migliore* dipende da molti fattori: dalle caratteristiche del problema in esame, dal calcolatore a disposizione, dall'impiego cui è destinato il programma. Se, invece, la programmazione fosse un processo puramente deterministico (cioè, che porta a un'unica soluzione obbligata), sarebbe stata automatizzata già da tempo.

Come nell'ingegneria, la costruzione di un prodotto — in questo caso di un algoritmo — avviene alternando fasi di analisi e scelte realizzative. Si comincia da un'analisi generale del problema e si sceglie una prima soluzione a grandi linee; a partire da questa, la costruzione della soluzione finale, adatta agli strumenti tecnici disponibili, avviene per gradi, specificando un numero sempre maggiore di dettagli. Ma, contrariamente a quanto accade nel campo dell'ingegneria, nel campo della programmazione i prodotti sono oggetti di natura astratta, e perciò slegati da complicazioni di natura fisica; i processi avvengono esattamente nel modo indicato nei programmi, senza effetti fisici collaterali (che, nell'ingegneria, possono essere causati dall'invecchiamento delle macchine e dai materiali di qualità inferiore usati).

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

in sottoproblemi. Ogni *passo di suddivisione* (o di *specificazione* dei sottoproblemi) deve verificare le seguenti condizioni:

- 1) la soluzione dei sottoproblemi conduce alla soluzione generale;
- 2) la successione di azioni da eseguire ha senso ed è possibile;
- 3) la suddivisione scelta dà luogo a sottoproblemi "più vicini" agli strumenti disponibili, cioè risolubili più direttamente nel linguaggio di programmazione, nel quale sarà scritta la versione finale del programma.

È proprio quest'ultima richiesta che rende impossibile uno sviluppo lineare, che parte dal problema e giunge alla soluzione attraverso la scomposizione del problema in sottoproblemi. Infatti, può accadere che un sottoproblema, deciso in un certo passo di specificazione, sia difficile da risolvere con i mezzi del linguaggio di programmazione, e che questa difficoltà si riveli solo in passaggi successivi. Occorre allora riprendere in considerazione i passaggi precedenti.

Se il processo di scomposizione del problema e il contemporaneo sviluppo e raffinamento del programma sono un processo di continuo approfondimento degli aspetti specifici, diremo che il modo di procedere è *dall'alto verso il basso (top-down)*. Ma si può seguire anche il metodo opposto, partendo dalle istruzioni del linguaggio di programmazione (o addirittura del linguaggio macchina), costruendo da queste dei sottoprogrammi assai semplici, collegando fra loro semplici sottoprogrammi in sottoprogrammi più complessi, e così via sino a ottenere il programma finale, che costituisce la soluzione completa del problema; questo procedimento, che parte dalla codificazione *profonda*, viene detto *dal basso verso l'alto (bottom-up)*. In pratica, come prima accennato, lo sviluppo di un programma non può seguire un procedimento puramente *top-down*, ma neppure puramente *bottom-down*. Tuttavia, si può affermare che, nella costruzione di un nuovo algoritmo, è *dominante il processo top-down*, mentre, nell'adattamento (a scopi diversi) di un programma già scritto, assume una maggiore importanza il metodo *bottom-up*.

Tanto la scomposizione di un problema in sottoproblemi, quanto la composizione di un programma a partire da componenti separate, dà luogo a una costruzione ben *strutturata*. È estremamente importante disporre di un linguaggio in grado di esprimere tale articolazione in sottoproblemi: solo così è possibile ottenere una versione finale del programma, che rappresenta il processo di soluzione seguito e che permette di ricostruire e di verificare il processo stesso. Una formulazione in un linguaggio non strutturato, al contrario, (caso estremo è l'insieme delle cifre binarie che rappresentano un programma nella memoria), rappresenta una versione priva di

quelle informazioni che permettono di distinguere un programma da un accostamento casuale di simboli.

Quando si scomponete un'azione in azioni più semplici, è necessario introdurre nuove variabili, che indicano gli argomenti e i risultati dei sottoprogrammi corrispondenti, e che stabiliscono il legame fra i sottoprogrammi. Queste variabili devono essere introdotte, durante lo sviluppo per passi di specificazione successivi, solo nei sottoprogrammi, per i quali si rendono necessarie. Inoltre, spesso, una descrizione più dettagliata della struttura dei dati comporta la specificazione di ulteriori dettagli. Perciò il linguaggio usato dovrebbe permettere anche di rappresentare strutture di dati gerarchiche. Da quanto detto risulta chiara l'importanza dei concetti relativi alle variabili locali (v. [12.2]) e ai tipi di dati strutturati (v. cap. 10 e 11).

Nei quattro paragrafi seguenti, illustreremo con degli esempi le considerazioni riportate sopra. Essi non riguardano (eccetto, forse, il primo) problemi comuni nella elaborazione dei dati; ma sono stati scelti perché, nonostante la loro brevità, essi illustrano nei tratti essenziali i metodi della programmazione sistematica.

15.1. Soluzione di un sistema di equazioni lineari

Considereremo ora lo sviluppo di un programma che calcoli i valori delle incognite x_1, \dots, x_n soluzioni del sistema di n equazioni lineari

$$\sum_{j=1}^n a_{ij} * x_j = b_i \quad i = 1, \dots, n \quad (15.1)$$

per valori a_{ij} e b_i assegnati. Un esempio per $n=3$ è fornito dalle equazioni (15.2):

$$\begin{aligned} x_1 + 2 * x_2 + 5 * x_3 &= 4 \\ 3 * x_1 + x_2 + 4 * x_3 &= 11 \\ -2 * x_1 + 5 * x_2 + 9 * x_3 &= -7 \end{aligned} \quad (15.2)$$

Utilizzeremo come metodi di soluzione, il metodo di eliminazione delle incognite (metodo di Gauss), che consiste nel seguente procedimento: nel primo passo si elimina l'incognita x_1 , rappresentandola come

$$x_1 = \left(b_1 - \sum_{j=2}^n a_{1j} * x_j \right) / a_{11} \quad (15.3)$$

e sostituendola nelle rimanenti $n-1$ equazioni; si ottiene così un sistema di $n-1$ equazioni in $n-1$ incognite. Lo stesso procedimento,

detto *passo di eliminazione*, viene ancora applicato al nuovo sistema di equazioni e ripetuto fino a ottenere un sistema costituito da una sola equazione (di immediata soluzione).

Descriviamo in modo più preciso e generale il k -esimo passo di eliminazione; a partire dal sistema di $n-k+1$ equazioni (ottenuto dopo $k-1$ passi)

$$\sum_{j=k}^n a_{ij}^{(k)} * x_j = b_i^{(k)} \quad i = k, \dots, n \quad (15.4)$$

si calcolano i coefficienti $a_{ij}^{(k+1)}$ e $b_i^{(k+1)}$ e si ottiene il sistema di $n-k$ equazioni:

$$\sum_{j=k+1}^n a_{ij}^{(k+1)} * x_j = b_i^{(k+1)} \quad i = k+1, \dots, n \quad (15.5)$$

dove i coefficienti $a_{ij}^{(k+1)}$ e $b_i^{(k+1)}$ vengono calcolati come combinazioni lineari della k -esima e della i -esima riga di $a^{(k)}$ e di $b^{(k)}$, e più precisamente sono forniti dalle relazioni:

$$\begin{aligned} a_{ij}^{(k+1)} &= a_{ij}^{(k)} - (a_{kj}^{(k)} / a_{kk}^{(k)}) * a_{ik}^{(k)} \\ b_i^{(k+1)} &= b_i^{(k)} - (b_k^{(k)} / a_{kk}^{(k)}) * a_{ik}^{(k)} \end{aligned} \quad (15.6)$$

per $i, j = k+1, \dots, n$. Per $j=k$ si ottiene:

$$a_{ik}^{(k+1)} = a_{ik}^{(k)} - (a_{kk}^{(k)} / a_{kk}^{(k)}) * a_{ik}^{(k)} = 0 \quad (15.7)$$

per ogni i ; cioè i coefficienti dell'incognita x_k sono tutti nulli (e quindi non occorre nemmeno calcolarli) per cui x_k viene effettivamente eliminata.

Dopo $n-1$ passi di eliminazione si ottiene il sistema di equazioni:

$$a_{nn}^{(n)} * x_n = b_n^{(n)} \quad (15.8)$$

che permette di calcolare immediatamente il valore di x_n . I valori delle rimanenti incognite vengono via via determinati sostituendo i valori x_j già calcolati, nelle equazioni del sistema corrispondente al precedente passo di eliminazione. Per esempio, x_{n-1} si ottiene sostituendo x_n nell'equazione:

$$a_{n-1,n-1}^{(n-1)} * x_{n-1} + a_{n-1,n}^{(n-1)} * x_n = b_{n-1}^{(n-1)} \quad (15.9)$$

Questo passaggio viene detto *passo di risostituzione*. Nel k -esimo

passo di risostituzione l'incognita x_k è data da:

$$x_k = \left(b_i^{(k)} - \sum_{j=k+1}^n a_{ij}^{(k)} * x_j \right) / a_{kk}^{(k)} \quad (15.10)$$

con i qualsiasi compreso fra k ed n . (Per motivi che vedremo in seguito, per lo più si sceglie $i = k$). Si osservi tuttavia che la successione dei passi di risostituzione non è arbitraria; infatti, per calcolare x_k , occorre prima conoscere i valori di x_{k+1}, \dots, x_n .

Nel caso dell'esempio (15.2) l'intero processo di soluzione è rappresentato nella figura 15.1.

Eliminazione

1.	<table border="1"> <tr><td>1</td><td>2</td><td>5</td><td>4</td></tr> <tr><td>3</td><td>1</td><td>4</td><td>11</td></tr> <tr><td>-2</td><td>5</td><td>9</td><td>-7</td></tr> </table> $a^{(1)}$	1	2	5	4	3	1	4	11	-2	5	9	-7	$b^{(1)}$	$*3/1$ $*-2/1$	
1	2	5	4													
3	1	4	11													
-2	5	9	-7													
2.	<table border="1"> <tr><td>-5</td><td>-11</td><td>-1</td></tr> <tr><td>9</td><td>19</td><td>1</td></tr> </table> $a^{(2)}$	-5	-11	-1	9	19	1	$b^{(2)}$	$*9/-5$							
-5	-11	-1														
9	19	1														
3.	<table border="1"> <tr><td></td><td></td><td></td></tr> <tr><td>-4/5</td><td>-4/5</td><td></td></tr> </table> $a^{(3)}$				-4/5	-4/5		$b^{(3)}$								
-4/5	-4/5															

Risostituzione

3. $x_3 = (-4/5)/(-4/5) = 1$
2. $x_2 = (-1 + 11 * 1)/(-5) = -2$
1. $x_1 = (+4 - 5 * 1 - 2 * (-2))/1 = 3$

Figura 15.1.

Per costruire un programma corrispondente a questo metodo di soluzione, è determinante poter calcolare $a^{(k+1)}$ e $b^{(k+1)}$ in base

solo ai coefficienti di $a^{(k)}$ e di $b^{(k)}$, senza usare quelli di apice hk . Inoltre, bisogna stabilire se, per ogni k , si possono usare le stesse variabili A e B , per rappresentare $a^{(k)}$ e $b^{(k)}$. Prima di rispondere a questa domanda, bisogna sapere quali coefficienti sono necessari nei passi di risostituzione successivi. Per questo scopo, basta ricordare che, nel k -esimo passo di risostituzione, per calcolare x_k si può usare una qualsiasi riga a_i , con $k < i \leq n$; allora, se si adopera la riga k -esima, le righe di A e di B , che devono contenere i valori $a_{ij}^{(k)}$ e $b_i^{(k)}$, restano libere per $i = k+1 \dots n$. Esse possono essere quindi usate per memorizzare i nuovi coefficienti $a_{ij}^{(k+1)}$ e $b_i^{(k+1)}$. Concludendo, bastano *una sola* variabile A e *una sola* variabile B , che rappresentano i coefficienti successivi $a^{(1)}, a^{(2)}, \dots, a^{(n)}$ e $b^{(1)}, b^{(2)}, \dots, b^{(n)}$. Lo spazio di memoria necessario, quindi, si riduce da

$$n^2 + (n-1)^2 + \dots + 2^2 + 1^2 = \frac{1}{6}(2n^3 + 3n^2 + n)$$

unità di memoria necessarie per $a^{(1)}, \dots, a^{(n)}$

$$n + (n-1) + \dots + 2 + 1 = \frac{1}{2}(n^2 + n)$$

unità di memoria necessarie per $b^{(1)}, \dots, b^{(n)}$ a soltanto $n^2 + n$ unità di memoria che contengono variabili A e B simultaneamente. Tale risparmio di memoria è la ragione per cui si preferisce impiegare il metodo di Gauss (con analoghe considerazioni è possibile memorizzare nella variabile B anche i valori x_1, \dots, x_n e in tal modo eliminare una variabile X risparmiando ulteriore spazio di memoria).

In base alle precedenti considerazioni, introdurremo le variabili A , B , X nel modo seguente:

```
var A: array [1 .. n, 1 .. n] of real;
B, X: array [1 .. n] of real
```

(15.11)

Versione 1: la prima versione del programma sarà:

```
begin "assegnamento dei valori ad A e a B";
for k := 1 to n-1 do
  begin "calcola a^{(k+1)} e b^{(k+1)} a partire da a^{(k)} e b^{(k)}
         secondo la (15.6)"
  end;
  k := n;
  repeat "calcola x_k secondo la (15.10)";
    k := k - 1
  until k = 0
end
```

(15.12)

Versone 2: si ottiene precisando l'istruzione "calcola ... secondo la (15.6)":

```
for i := k+1 to n do
begin "calcola la i-esima riga di  $a^{(k+1)}$  e  $b^{(k+1)}$  secondo la
(15.6)"
end
```

(15.13)

A questo punto, si osserva che nel calcolo di $a_{ij}^{(k+1)}$ (e di $b_i^{(k+1)}$) secondo la (15.6), il fattore $a_{kj}^{(k)}/a_{kk}^{(k)}$ (e rispettivamente $b_k^{(k)}/a_{kk}^{(k)}$) non dipende da i . Per non ripetere il calcolo di espressioni che rimangono invariate, si deve portare la divisione per $a_{kk}^{(k)}$ all'esterno della istruzione di ripetizione for. Ma dove memorizzare i quozienti? È possibile sostituire i valori $a_{kj}^{(k)}$ con $a_{kj}^{(k)}/a_{kk}^{(k)}$ (e $b_k^{(k)}$ con $b_k^{(k)}/a_{kk}^{(k)}$)? Di nuovo, occorre prima studiare l'effetto di una tale operazione sui passi di risostituzione successivi, nei quali si utilizzeranno questi coefficienti. Ricordiamo che la moltiplicazione (e la divisione) di tutti i coefficienti di un'equazione per uno stesso fattore non ne cambia le soluzioni; osserviamo inoltre che la divisione di $a_{kk}^{(k)}$ per $a_{kk}^{(k)}$ stesso fornisce il valore 1, per cui diventa superflua la divisione nel passo di risostituzione. Quindi la suddetta sostituzione è non solo possibile, ma anche vantaggiosa!

Versone 3: in base a tali considerazioni, il passo di eliminazione è dato da:

```
for k := 1 to n-1 do
begin p := 1/A[k, k];
for j := k+1 to n do A[k, j] := p * A[k, j];
B[k] := p * B[k];
for i := k+1 to n do
begin "calcola  $a^{(k+1)}$  e  $b^{(k+1)}$  secondo la (15.6)"
end
end
```

(15.14)

L'istruzione "calcola secondo (15.6)", per il fatto che $A [k, j]$ rappresenta il valore $a_{kj}^{(k)}/a_{kk}^{(k)}$ diventa:

Versone 4:

```
for i := k+1 to n do
begin for j := k+1 to n do A[i, j] := A[i, j] - A[i, k] * A[k, j];
B[i] := B[i] - A[i, k] * B[k]
end
```

(15.15)

Lo stesso processo va ancora applicato solo all'istruzione "calcola ... secondo (15.10)" che rappresenta la risostituzione. Essa risulta essere una successione di sottrazioni, nella quale la divisione per $a_{kk}^{(k)}$ è ancora superflua.

Versone 5: k-esimo passo di risostituzione

```
begin t := B[k];
for j := k+1 to n do t := t - A[k, j] * X[j];
X[k] := t
end
```

(15.16)

Si deve notare che, per definizione, l'istruzione for viene eseguita solo se il valore iniziale delle variabili di controllo è minore del limite finale; se ciò non si verifica, l'istruzione non viene eseguita.

L'intera soluzione è riassunta in (15.17). Si noti inoltre che vengono eseguiti n passi di eliminazione invece di $n-1$. L'ultimo passo produce però semplicemente la (necessaria) divisione di $b_n^{(n)}$ per $a_{nn}^{(n)}$.

Procedimento di eliminazione di Gauss {soluzione di un sistema di equazioni lineari secondo Gauss}:

```
var i, j, k: 1 .. n;
p, t: real;
A: array [1 .. n, 1 .. n] of real;
B, X: array [1 .. n] of real;
begin {assegnamento dei valori ad A e B}
for k := 1 to n do
begin p := 1.0/A[k, k];
for j := k+1 to n do A[k, j] := p * A[k, j];
B[k] := p * B[k];
for i := k+1 to n do
begin for j := k+1 to n do
A[i, j] := A[i, j] - A[i, k] * A[k, j];
B[i] := B[i] - A[i, k] * B[k]
end
end;
k := n;
repeat t := B[k];
for j := k+1 to n do t := t - A[k, j] * X[j];
X[k] := t; k := k - 1
until k = 0
{X[1] ... X[n] sono le soluzioni}
end
```

(15.17)

In questo programma, è necessario prestare particolare attenzione alla divisione, perché l'algoritmo non può essere utilizzato quando compare un divisore con valore zero. Questo pericolo potenziale è tanto maggiore quando si usa un'aritmetica con precisione finita, poiché anche i divisorì con valore vicino allo zero portano a risultati errati. Il fatto che una permutazione arbitraria di righe ($a_i^{(k)}$, $b_j^{(k)}$) o una permutazione di colonne $a_i * j^{(k)}$ lascia immutati i risultati (pur di ribattezzare i risultati X_j), permette di scegliere un divisore (detto *Pivot*) tra tutti i possibili $a_i^{(k)}$ in modo tale che l'eventuale errore sia minimo. Si sceglie sempre come *Pivot* il divisore con valore assoluto maggiore. Questa scelta del *Pivot* complica alquanto l'algoritmo, ma non è in generale impossibile. Se, infine, non è possibile trovare alcun divisore con valore significativamente diverso da zero, si dice che il sistema di equazioni è mal condizionato; se tutti i divisorì sono nulli il sistema è singolare, cioè non ha nessuna soluzione significativa. Una breve analisi dell'algoritmo mostra che l'operazione

$$A[i, j] := A[i, j] - A[i, k] * A[k, j] \quad (15.18)$$

ricorre molto frequentemente e cioè

$$(n-1)^3 + (n-2)^2 + \dots + 2^2 + 1^2 = \frac{1}{6}(2n^3 - 3n^2 + n)$$

volte. Per grandi valori di n il numero di operazioni cresce quindi come n^3 .

15.2. Determinazione del più piccolo numero eguale a due somme diverse di numeri naturali elevati alla terza potenza

Riportiamo un esempio, che illustra lo sviluppo sistematico, per passi di raffinamento successivi, di programmi, che vengono perfezionati in ogni passo dello sviluppo.

Il problema consiste nel trovare il più piccolo numero x , tale che

$$x = a^3 + b^3 = c^3 + d^3 \quad (15.20)$$

dove a, b, c, d sono numeri naturali e $a \neq c$ e $a \neq d$. Senza bisogno di profonde conoscenze della teoria dei numeri, la cosa più sensata sembra essere il calcolo dei valori "candidati per x ", secondo un ordinamento crescente, e interrompere il processo non appena due valori successivi sono identici. A tal fine, vengono considerati candidati tutti quei numeri che si possono rappresentare come somma di due potenze terze. La prima versione di un programma può essere espressa come segue:

Versione 1:

```
x := 2; {2 = 1^3 + 1^3}
repeat min := x;
      x :=
until x = min
```

prossima somma (maggiore) di due potenze

Il problema è così ridotto al calcolo della *prossima somma (maggiore) di due potenze*. Per trovare un punto di partenza, da cui affrontare il problema, è consigliabile calcolare *a mano* l'inizio di questa sequenza. Questo modo di procedere sarà qui illustrato per lo stesso problema, relativo alla somma di due quadrati, anziché di due cubi: la tabella 15.I contiene i candidati in una sequenza non ordinata formata dagli elementi $S_{ij} = i^2 + j^2$.

TABELLA 15.I

$i \backslash j$	1	2	3	4	5	6	7	8	...
1	2								
2	5	8							
3	10	13	18						
4	17	20	25	32					
5	26	29	34	41	50				
6	37	40	45	52	61	72			
7	50	53	58	65	74	85	98		
8	65	68	73	80	89	100	113	128	
9	82	85	...						
...									

Si vede immediatamente che

$$50 = 1^2 + 7^2 = 5^2 + 5^2$$

$$65 = 1^2 + 8^2 = 4^2 + 7^2$$

$$85 = 2^2 + 9^2 = 6^2 + 7^2$$

e che 50 è il minimo cercato. Per rendere automatica questa ricerca si tratta di generare i candidati nella successione

$$2, 5, 8, 10, 13, 17, \dots, 45, 50, 50 \quad (15.21)$$

Per proseguire, sono utili le seguenti osservazioni:

- 1) $S_{ij} > S_{ik}$ per tutti gli $i, j > k$;
- 2) $S_{ij} > S_{kj}$ per tutti i $j, i > k$;
- 3) per simmetria $S_{ij} = S_{ji}$, basta prendere in considerazione solo S_{ij} con $j \leq i$.

Dal punto 1), segue immediatamente che, in una riga della tabella 15.I, i candidati vengono esaminati sempre da sinistra verso destra. Quindi, è superfluo ricordare ogni riga per intero; basta che di volta in volta sia memorizzato l'ultimo candidato. Tornando al problema di partenza, la relativa tabella può quindi essere rappresentata con la variabile

`var S: array [1 .. ?] of integer` (15.22)

Inoltre, per memorizzare l'ultimo candidato generato, si introduce la variabile indice j :

`var j: array [1 .. ?] of integer` (15.23)

per la quale deve sempre valere la relazione:

$$S[k] = k^3 + j[k]^3 \quad (15.24)$$

Se, al posto di x , si determina direttamente l'indice i dell'ultimo candidato generato e se, inoltre, si introduce la funzione $p(k) = k^3$, si ottiene la seguente versione del programma:

Versione 2:

```
for k := 1 to ? do
  begin j[k] := 1; S[k] := p(k) + 1
  end;
repeat min := S[i];
```

(15.25)

1: incrementa $j[i]$ e sostituisci $S[i]$ con il successivo valore della riga i -esima;

2: assegna a i l'indice della riga contenente il più piccolo candidato non ancora esaminato.

`until S[i] = min`

Questa versione è inadeguata, perché il numero delle componenti $S[k]$, alle quali è assegnato il valore iniziale $k^3 + 1^3$, è indeterminato.

In linea di principio, anche l'istruzione 2) implica l'esame di un numero indeterminato di candidati. Se però si considera l'osservazione 2), l'esame (e quindi anche l'inizializzazione) degli elementi $S[k]$, con $k > ih$, è superflua, dove ih è il minimo indice tale che $j[ih] = 1$ (cioè $j[k] > 1$ per tutti i $k < ih$). Ne deriva la

Versione 3:

```
i := 1; ih := 2;
j[1] := 1; S[1] := 2; j[2] := 1; S[2] := p(2) + 1;
repeat min := S[i];
  1: if j[i] = 1 then
```

(15.26)

incrementa ih e inizializza $S[ih]$;
 2: incrementa $j[i]$ e sostituisci $S[i]$ con il successivo valore della riga;
 3: determina in modo che $S[i] = \min(S[1] \dots S[ih])$
`until S[i] = min`

L'esame di altri candidati nella riga i -esima deve comunque essere interrotto, appena $j[i] = 1$, e questo, non solo perché l'esame può essere limitato al "triangolo inferiore" della tabella simmetrica, ma anche (e soprattutto) perché la prosecuzione genererebbe e riconoscerebbe le coppie $a^3 + b^3$ e $b^3 + a^3$ come somme eguali di valori distinti. Però, raggiunto il limite $j[i] = 1$, si escluderebbero completamente alcune righe; quindi, è necessario introdurre, per l'indice di riga i , anche un limite inferiore il , oltre al limite superiore ih . L'esame dei candidati (righe) viene quindi ulteriormente specificato nel modo desiderato. Il limite inferiore dell'indice i viene via via elevato, appena si elimina una riga:

Versione 4:

```
i := 1; il := 1; ih := 2; ...
repeat min := S[i];
  if j[i] = 1 then il := il + 1 else
    begin 1: if j[i] = 1 then
```

(15.27)

incrementa ih e inizializza $S[ih]$;
 2: incrementa $j[i]$ e sostituisci $S[i] \dots$
`end;`
 3: determina i in modo che $S[i] = \min(S[i \dots S[ih])$
`until S[i] = min`

Si tratta ora di precisare le istruzioni 1 + 3 in (15.27). Se nell'istruzione 3 si determina il minimo con una semplice ricerca lineare (v. (11.20)), si ottiene la parte di programma (15.28):

Versione 5: (istruzione 3)

```
3; i := il; k := i;
while k < ih do
begin { S[i] = min(S[il] . . . S[k]) } k := k + 1;
  if S[k] < S[i] then i := k
end
```

(15.28)

La formulazione più semplice dell'istruzione 2 è:

$$j[i] := j[i] + 1; \quad S[i] := p(i) + p(j[i]) \quad (15.29)$$

e dell'istruzione 1 è:

```
1: if j[i] = 1 then
begin ih := ih + 1; j[ih] := 1; S[ih] := p(ih) + 1
end
```

(15.30)

Il programma così ottenuto può essere ulteriormente perfezionato, evitando di ripetere il calcolo di $p(i)$ per calcolare le somme $S[i]$. Questa modifica è "doverosa", perché è necessario calcolare una nuova potenza $p(i)$ solo quando si ha un incremento di ih e tale calcolo può essere facilmente incorporato al punto giusto nel programma considerato. Se, per memorizzare le potenze, si introduce la variabile

```
var p: array [1 . . ?] of integer
```

(15.31)

e se la funzione $p(x)$ viene sistematicamente sostituita con $p[x]$, l'ulteriore raffinamento del programma si ottiene inserendo

$$p[ih] := ih * ih * ih \quad (15.32)$$

nell'istruzione 1 (15.30). Si ottiene così la versione definitiva, riportata come programma completo in (15.33). Questo programma

$$1729 = 10^3 + 9^3 = 12^3 + 1^3$$

esaminando 61 candidati. Inoltre, i valori finali di i e di il e di ih sono $il = 10$, $ih = 12$, e il test $S[k] < S[i]$ è seguito 107 volte. Cambiando l'espressione (15.32) e i valori iniziali di S , si può applicare

questo programma anche per il calcolo del minimo numero, rappresentabile come somma di potenze quarte differenti. Però, il tempo di calcolo aumenta rapidamente: tale programma calcola il risultato

$$634318657 = 134^4 + 133^4 = 158^4 + 59^4$$

dopo aver esaminato 11660 candidati.

```
var i, il, ih, min, a, b, k: integer;
j, p, S: array [1 . . 12] of integer;
{p[k] = k^3, S[k] = p[k] + p[j[k]] for k = 1 . . ih}
begin i := 1; il := 1; ih := 2;
j[1] := 1; p[1] := 1; S[1] := 2; j[2] := 1; p[2] := 8; S[2] := 9;
repeat min := S[i]; a := i; b := j[i];
  if j[i] = i then il := il + 1 else
    begin if j[i] = 1 then
      begin ih := ih + 1; p[ih] := ih * ih * ih;
        j[ih] := 1; S[ih] := p[ih] + 1
      end;
      j[i] := j[i] + 1; S[i] := p[i] + p[j[i]]
    end;
    i := il; k := i;
    while k < ih do
      begin k := k + 1;
        if S[k] < S[i] then i := k
      end
    until S[i] = min;
    write (min, a, b, i, j[i], eol)
end.
```

(15.33)

15.3. Calcolo dei primi n numeri primi

Il problema della determinazione dei primi n numeri primi ha una certa somiglianza con l'esempio precedente, in quanto anche qui si determinano i membri di una successione di numeri passo per passo. Il criterio di terminazione è ancora più semplice e la prima versione immediatamente proposta è:

Versione 1:

```
var i, x: integer;
begin x := 1;
```

(15.34)

```

for i := 1 to n do
begin x := "prossimo numero primo"; write(x)
end
end.

```

L'unica istruzione di questo programma, che necessita ancora di precisazioni, è " $x := \text{prossimo numero primo } (x)$ ". Questa istruzione, che fa riferimento solo al fatto che i numeri richiesti siano primi, è esprimibile introducendo una variabile booleana *prim*:

Versione 2:

```

repeat x := x + 1;
  prim := "x è un numero primo"      (15.35)
until prim

```

Si può immediatamente ridurre il tempo di calcolo necessario considerando che, eccetto 2, tutti i numeri primi sono dispari. Se il numero primo 2 viene trattato come caso a parte, si può di volta in volta aumentare di 2. Il raffinamento successivo consiste nello scindere l'istruzione

```
prim := "x è un numero primo"
```

in passi più elementari. Per far ciò, dobbiamo riferirci (per la prima volta) alla definizione di numero primo: x è un numero primo se e solo se è divisibile soltanto per 1 e per se stesso, cioè se e solo se non può essere diviso senza resto per nessuno dei numeri 2, 3, ... $x - 1$. Ne risulta un'altra struttura ripetitiva, innestata nella ripetizione (15.35):

Versione 3:

```

repeat x := x + 2; k := 2;
  repeat {x non è divisibile per 2, 3, ... k}      (15.36)
    k := k + 1; prim := "x non è divisibile per k"
  until  $\neg \text{prim} \vee (k \geq \text{lim})$ 
until prim

```

Evidentemente, per la grandezza *lim* si può scegliere direttamente l'espressione $x - 1$. Molto più vantaggiosa, e altrettanto adeguata, è però la scelta $\text{lim} = \sqrt{x}$; infatti, se x fosse divisibile per un numero $k > \sqrt{x}$, x sarebbe rappresentabile come prodotto $x = k * j$; ma

allora x sarebbe divisibile anche per $j < \sqrt{x}$, il che è, evidentemente, una contraddizione. Altrettanto importante, per un ulteriore sviluppo del programma, è la considerazione seguente: ci si può limitare a considerare la divisibilità di x per i *numeri primi*, poiché se x è divisibile per k è divisibile anche per i fattori primi di k . È quindi consigliabile memorizzare i numeri primi già calcolati in una tabella *p*. Dalle precedenti considerazioni, risulta la

Versione 4:

```

repeat x := x + 2; k := 2; prim := true;
  while prim  $\wedge (k < \text{lim})$  do
    begin prim := "x non è divisibile per p [k]"
      k := k + 1
    end
  until prim;
  p[i] := x

```

(15.37)

Si deve ora determinare il valore di *lim*, dato dall'indice del più grande (e ultimo) numero primo *p* [*lim*], per il quale si deve verificare se è un divisore di x . Perciò deve essere:

$$p[\text{lim}] > \sqrt{x} \quad \text{and} \quad p[\text{lim} - 1] \leq \sqrt{x} \quad (15.38)$$

Fin qui, si è tacitamente assunto che i valori *p* [1] ... *p* [*lim*] siano già noti, cioè risultino da elaborazioni precedenti. Ma ciò accade solo se il candidato x da esaminare è minore di *p* [*i* - 1]², cioè se è sempre

$$p[i] < p[i - 1]^2 \quad (15.39)$$

Fortunatamente, la teoria dei numeri garantisce proprio la validità di questa condizione per tutti i numeri primi. Evidentemente, l'indice *lim* deve essere *controllato* ogni volta che x viene aumentato e deve essere aumentato appena $p[\text{lim}]^2 \leq x$; basta incrementare *lim* di 1, poiché, dopo l'ultimo controllo, il valore x è stato incrementato soltanto di 2, e $p_{i+1}^2 > p_i^2 + 2$ per ogni *i*. Queste considerazioni ci portano a formulare la versione 5 che fornisce una visione d'insieme dell'intero programma fin qui sviluppato:

Versione 5:

```

type index = 1 .. n;
var x: integer;

```

```

i, k, lim: index; prim: Boolean;
p: array [index] of integer; {p [i] = ith numero primo}
begin p[1] := 2; write(2); x := 1; lim := 1;
      for i := 2 to n do
begin
  repeat x := x + 2;
    if sqr(p[lim]) ≤ x then lim := lim + 1;
    k := 2; prim := true;
    while prim ∧ (k < lim) do
      begin prim := "x non è divisibile per p [k]";
            k := k + 1
          end
        until prim;
        p[i] := x; write(x)
      end
end.

```

Rimane infine da precisare ancora l'istruzione

prim := "x non è divisibile per p [k]"

in modo che possa essere formulata con gli operatori disponibili nel linguaggio di programmazione.

Se si applicano gli operatori introdotti nel paragrafo 8, l'istruzione può essere espressa nei modi seguenti:

prim := (x mod p [k]) ≠ 0 (15.41)

oppure

*prim := (x div p [k]) * p [k] ≠ x* (15.42)

Ci si potrebbe accontentare di queste soluzioni. Imponiamo tuttavia la condizione aggiuntiva che non si debbano impiegare divisioni. In questo caso l'istruzione (15.42-15.43) può essere rappresentata da un programma che sottrae ripetutamente il divisore da *x*:

```

r := x;
repeat r := r - p[k] until r ≤ 0;
prim := r < 0 (15.43)

```

Ma l'istruzione (15.43) deve essere applicata con particolare frequenza e, d'altronde, ripetere la sottrazione è piuttosto lungo;

è allora il caso di cercare una soluzione più elegante e più efficiente. Una soluzione semplice e adeguata consiste nel memorizzare, oltre ai numeri primi *p [2], p [3], ..., p [lim]*, anche i loro multipli *V [k] = m * p [k]* tali che:

$$x \leq V[k] < x + p[k] \quad \text{for } k = 2, \dots, lim \quad (15.44)$$

In questo caso, la divisibilità di *x* per *p [k]* può essere determinata semplicemente confrontando *x* con *V [k]*. Se, in considerazione del frequentissimo impiego dell'espressione *p [lim]²*, si introduce la variabile ausiliaria *square*:

$$square = p[lim]^2 \quad (15.45)$$

Il programma definitivo risulta:

Versione 6:

```

type index = 1 .. n;
var x, square: integer;
i, k, lim: index; prim: Boolean;
p: array [index] of integer;
V: array [1 .. √n] of integer;
begin p[1] := 2; write(2); x := 1; lim := 1; square := 4;
      for i := 2 to n do
begin
  repeat x := x + 2;
    if square ≤ x then
      begin V[lim] := square;
            lim := lim + 1; square := sqr(p[lim])
          end;
        k := 2; prim := true;
        while prim ∧ (k < lim) do
          begin if V[k] < x then
                V[k] := V[k] + p[k];
                prim := (x ≠ V[k]); k := k + 1
              end
            until prim;
            p[i] := x; write(x)
          end
end.

```

Questo esempio mostra chiaramente come l'ulteriore limitazione di dover lavorare con un calcolatore privo della divisione sia stata

uno stimolo a trovare una soluzione ben meditata. Infatti non è raro che calcolatori efficienti distolgano i programmati dal perfezionare gli algoritmi in modo che i risultati richiesti siano prodotti con un tempo di calcolo ragionevole.

La tabella 15.II illustra la frequenza con cui vengono impiegate alcune istruzioni, in funzione dei numeri primi da calcolare.

TABELLA 15.II.

$n =$	10	20	50	500	1000
$x := x + 2$	14	114	611	1785	3959
$lim := lim + 1$	3	6	11	17	23
$prim := (x \neq V[k])$	13	268	2340	9099	25133
$V[k] := V[k] + p[k]$	8	156	1151	3848	9287

15.4. Un esempio di algoritmo euristico

Il problema trattato in questo paragrafo è un esempio semplice, ma tipico, di una classe di problemi le cui soluzioni si ottengono in modo euristico, cioè per *tentativi e verifiche successivi*. Questo metodo consiste nello sviluppare successive ipotesi di soluzione, confrontandole, di volta in volta, con le richieste a cui deve soddisfare una soluzione.

Se una ipotesi si dimostra inaccettabile, viene lasciata cadere. Nell'esempio seguente il problema è: "costruire una sequenza di lunghezza N su un alfabeto di tre simboli (per esempio 1, 2 e 3), in modo che due sottosequenze consecutive non siano mai identiche".

Per esempio, la sequenza di lunghezza 5 "12321" è accettabile, mentre non lo sono "12323" oppure "12123". In problemi di questo tipo, è raccomandabile far crescere sistematicamente la sequenza fino alla lunghezza N , producendo sequenze man mano più lunghe; cominciando dalla sequenza vuota. Certamente non ha senso allungare una sequenza che non soddisfa la condizione posta; ne consegue la strategia di verificare, a ogni passo, la sequenza ottenuta e di allungarla se soddisfa la condizione posta (in questo caso diremo che la sequenza è "buona"), oppure di modificarla se non soddisfa la condizione. Nella prima versione del programma si introducono due variabili, una per rappresentare la lunghezza e l'altra per rappresentare la "bontà" della sequenza S prodotta. La prima variabile è indicata con m e assume i valori 0, 1, 2, ..., mentre per la seconda si può introdurre un campo di valori *buono* o *cattivo*, oppure si può

usare una variabile tipo *boolean* con un nome appropriato, per esempio *good*. Ci atteniamo qui alla seconda soluzione con la convenzione:

good = true significa: la sequenza soddisfa la condizione;
good = false significa: la sequenza non soddisfa la condizione

Versione I:

```
var m: 0 .. N; good: Boolean; S: Sequence; (15.47)
begin m:=0; good:=true; {la sequenza vuota è "buona"}
repeat if good then "allunga la sequenza S"
  else "cambia la sequenza S"
    good:="S è una sequenza buona"
until good  $\wedge$  (m=N);
print (S)
end.
```

Per modifica si intende il cambiamento o la rimozione di certe componenti della sequenza, senza che però aumenti il loro numero. Perché l'algoritmo giunga a un risultato finale, la modifica deve sempre essere fatta in modo che non si ripresenti mai una sequenza già prodotta in precedenza. Ciò implica che la modifica deve essere effettuata in modo pianificato e sistematico, e che si deve seguire un qualche ordinamento nel produrre le sequenze possibili. La scelta dei simboli '1', '2', e '3' suggerisce già un possibile ordinamento: se una sequenza $S=s_1s_2\dots$ è considerata come parte decimale del valore $|S|=0.s_1s_2\dots$, la relazione d'ordine $<$ è definita da:

$$S < S' \leftrightarrow |S| < |S'| \quad (15.48)$$

Si determinano così contemporaneamente gli algoritmi per allungare e per modificare la sequenza. Per gli allungamenti, bisogna cominciare con la soluzione minima, in modo da non escludere nessuna sequenza nei cambiamenti successivi; per allungare la sequenza S , si aggiunge il simbolo '1' ('1' < '2' < '3') a tale soluzione. Per poter descrivere nei dettagli le operazioni di allungamento e di modifica è indispensabile stabilire la struttura della variabile S che rappresenta la sequenza. Siccome le componenti della sequenza debbono poter essere esaminate e modificate, si esclude subito la struttura *file*. La scelta è quindi

```
var S: array [1 .. N] of char (15.49)
```

Per stabilire la modifica, bisogna tener presente che l'elemento $S[m]$ non può essere semplicemente sostituito con il simbolo immediatamente maggiore, quando $S[m] = '3'$. In questo caso si deve abbreviare la sequenza e cambiare il simbolo precedente (può essere anch'esso '3'!). A questo punto il lettore potrebbe analizzare da sè l'algoritmo; si generano i candidati indicati nella tabella 15.III.

TABELLA 15.III.

```
+ 1
  11
+ 12
+ 121
  1211
  1212
+ 1213
+ 12131
  121311
+ 121312
...
...
```

(Le sequenze "buone" sono indicate con il segno +). Se le operazioni di allungamento, di modifica e di verifica vengono rappresentate come procedure con i nomi *extend*, *change*, e *check*, si ottiene il seguente programma (15.50):

Versione 2:

```
var S: array [1 .. N] of char;
m: 0 .. N; good: Boolean;
procedure extend;
begin m := m + 1; S[m] := '1' end; (15.50)
procedure change;
begin if S[m] < '3' then S[m] := succ(S[m]) else
begin m := m - 1; {accorciamento di S}
  if m > 0 then
    if S[m] < '3' then S[m] := succ(S[m]) else
      begin m := m - 1; {accorciamento ulteriore}
        if m > 0 then S[m] := succ(S[m])
      end
    end
  end
end;
begin m := 0; good := {sequenza vuota}
```

```
repeat if good then extend else change;
  check
until good  $\wedge$  (m = N)  $\vee$  (m = 0);
print(S)
end.
```

La versione 2 tiene conto del fatto che sussiste la possibilità di accorciare S fino alla lunghezza 0, aggiungendo alla condizione di terminazione la condizione $m = 0$. Si noti inoltre che la procedura *change* prevede un accorciamento della sequenza di al massimo due elementi. Questa restrizione è lecita, perché le sequenze di candidati proposte si ottengono sempre aggiungendo un unico simbolo a una sequenza già *accettabile*.

Prima di passare a un esame più preciso della operazione *check*, richiamiamo l'attenzione su una variante di questo programma, che rappresenta una soluzione un po' più efficiente. Per prima cosa, si osservi che le tre operazioni fondamentali *extend*, *change* e *check* si succedono sempre nel modo illustrato nella figura 15.2.

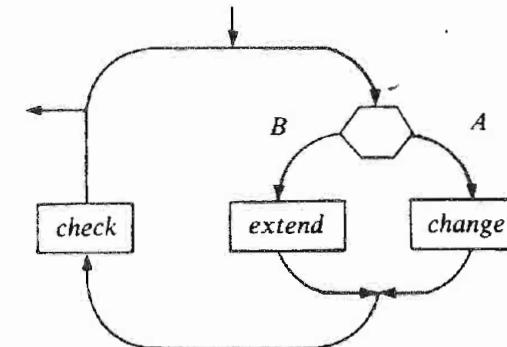


Figura 15.2.

Il cammino *A* (contenente l'operazione *change*) in media è percorso un numero di volte maggiore rispetto al cammino *B*; perciò, nella versione 2a, la ripetizione dell'operazione *change* presenta una condizione di terminazione più semplice, usando come struttura di ripetizione due cicli innestati:

Versione 2a:

```
repeat extend; check;
  while  $\neg$  good  $\wedge$  (m > 0) do
    begin change; check end
  until (m = N)  $\vee$  (m = 0) (15.51)
```

La versione 2a corrisponde al diagramma della figura 15.3, che mostra che le successioni possibili delle tre operazioni fondamentali sono le stesse del diagramma della figura 15.2.

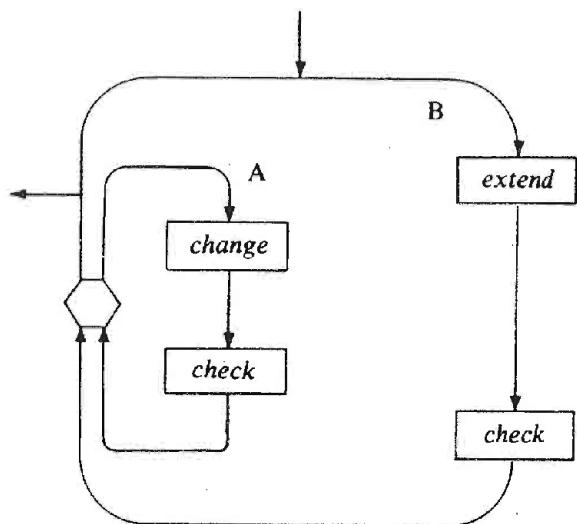


Figura 15.3.

Nella versione 2 si vorrebbe eliminare soprattutto la complessità dell'operazione *change* e quella della condizione di terminazione. La prima può essere semplificata impiegando un "trucco" frequentemente usato: la complessità della procedura *change* deriva dal fatto, che si tien conto di un caso che accade solo raramente, e forse anche mai: la riduzione della sequenza alla sequenza vuota. In questo caso, deve infatti essere impedito un assegnamento all'elemento inesistente $S[0]$; il "trucco" consiste allora nell'introdurre una componente $S[0]$ e nel permettere l'assegnamento suddetto. Si può quindi scrivere la procedura *change* come segue:

```
procedure change; (15.52)
begin while  $S[m] = '3'$  do  $m := m - 1$ ;
    $S[m] := \text{succ}(S[m])$ 
end
```

Inoltre se — in base a una analisi combinatoria — si sa che esiste una soluzione di lunghezza N , la (15.52) si può applicare perfino senza introdurre la componente $S[0]$, così che il programma principale (15.51) può essere ulteriormente semplificato:

```
repeat extend; check; (15.53)
  while  $\neg \text{good}$  do
    begin change; check end
  until  $m = N$ 
```

Si tratta ora di precisare l'operazione *check*, la cui definizione è (v. (15.47)):

$\text{good} := S$ è una sequenza buona

Si osservi che, fino a ora, non ci siamo mai riferiti alla condizione che caratterizza una sequenza buona, cioè accettabile. (Abbiamo semplicemente applicato la proprietà che allungando sequenze non accettabili non si ottengono sequenze accettabili.) La soluzione indicata è quindi molto generale! In questo esempio specifico, la procedura *check* deve verificare se esistono sottosequenze di lunghezza L adiacenti identiche con $1 \leq L \leq m/2$. Dato che un confronto di due sequenze di lunghezza L richiede esattamente L confronti (di simboli), il numero totale dei confronti è, al massimo:

$$\begin{aligned} N(m) &= (m-1)*1 + (m-3)*2 + \dots + 3*(m/2-1) + 1*(m/2) \\ &= \frac{1}{24}(m^3 + 3*m^2 + 2*m) \end{aligned} \quad (15.54)$$

per m pari, e

$$\begin{aligned} N(m) &= (m-1)*1 + (m-3)*2 + \dots + 4*\left(\frac{m-3}{2}\right) + 2*\left(\frac{m-1}{2}\right) \\ &= \frac{1}{24}(m^3 + 3*m^2 - m - 3) \end{aligned} \quad (15.55)$$

per m dispari. Il numero dei confronti cresce, quindi, proporzionalmente alla terza potenza della lunghezza m . Questo numero può essere comunque drasticamente ridotto, utilizzando il fatto che la sequenza da esaminare è stata ottenuta da una sequenza già buona, aggiungendo un unico simbolo. È quindi sufficiente confrontare solo le coppie di sottosequenze che contengono l'ultimo simbolo, cioè

$$\langle S_{m-2L+1} \dots S_{m-L}, S_{m-L+1} \dots S_m \rangle \quad (15.56)$$

per $L = 1 \dots m/2$. Il numero dei confronti è così ridotto a

$$Z'(m) = 1 + 2 + \dots + \frac{m}{2} = \frac{m}{4} \left(\frac{m}{2} + 1 \right) \quad (15.57)$$

In base a questi risultati la procedura *check* può essere scritta come segue:

```
procedure check;
  var L: integer;
begin good := true;
  for L := 1 to (m div 2) do
    good := good  $\wedge$  ((Sm-2L+1 ... Sm-L)  $\neq$  (Sm-L+1 ... Sm))
end
```

(15.58)

Ma si può ottenere una soluzione ancora più efficiente, se i confronti cessano, appena viene riconosciuta l'uguaglianza di due sequenze. Si osservi che, in genere, lo schema generale (9.2) non va applicato quando la relazione ricorsiva $v_i = f(v_{i-1})$ ha la forma $v_i = v_{i-1} \wedge q_i$. Nella versione migliorata (15.39), l'istruzione **for** è sostituita da una struttura **while**, poiché non è più noto a priori il numero delle ripetizioni.

```
procedure check;
  var L, mhalf: integer;
begin good := true; L := 0; mhalf := m div 2;
  while good  $\wedge$  (L < mhalf) do
    begin L := L + 1;
      good := (Sm-2L+1 ... Sm-L)  $\neq$  (Sm-L+1 ... Sm)
    end
  end
```

(15.59)

Precisando allo stesso modo — questa volta applicando una istruzione **repeat** — anche l'operazione di confronto di due sequenze, mediante una successione di confronti di simboli, si ottiene la versione definitiva della procedura *check*:

```
procedure check;
  var i, L, mhalf: integer;
begin good := true; L := 0; mhalf := m div 2;
  while good  $\wedge$  (L < mhalf) do
    begin L := L + 1; i := 0;
      repeat good := S[m-i]  $\neq$  S[m-L-i]; i := i + 1
      until good  $\vee$  (i = L)
    end
  end
```

(15.60)

Sono così finalmente specificate tutte le componenti del programma principale, in tutti i particolari, ed il problema posto è stato risolto nei passi (15.50), (15.52), (15.53), (15.60).

Infine useremo il problema precedente, anche per esemplificare il caso (assai frequente nella prassi della programmazione) in cui si modifica lo scopo originario e si deve adattare alle nuove esigenze il programma ottenuto. Prendiamo in considerazione la seguente estensione del problema: invece di un'unica sequenza arbitraria di lunghezza N , devono venir prodotte *tutte* le sequenze di lunghezza N che non contengono sottosequenze adiacenti identiche. Ora, se il programma originario è sufficientemente articolato in parti, di cui è possibile rappresentare separatamente il compito e il risultato, l'adattamento comporta semplicemente la modifica di alcune parti. Se un programma è articolato in un modo adeguato e chiaro, gli ampliamenti e le modifiche sono più facili; infatti, con una buona articolazione, si possono individuare e isolare facilmente le parti da modificare.

Nel caso in esame, non solo la chiara decomposizione del programma in singole parti, ma il modo sistematico di produrre i "candidati", comportano dei vantaggi. Le seguenti considerazioni portano direttamente alla soluzione (cfr. 15.50):

- 1) quando m raggiunge il valore N ($m = N$), la sequenza S va riconosciuta come risultato e stampata; in seguito, S potrà essere modificata, ma non allungata;
- 2) la condizione di terminazione può essere semplificata, poiché la relazione $m = N$ non è più rilevante; viene mantenuto solo il confronto $m = 0$.

A questo punto, è chiaro che l'algoritmo risultante, in primo luogo deve produrre solo sequenze accettabili, in secondo luogo deve esaminare tutti i candidati possibili. Nella tabella 15.IV sono riportate le sequenze accettabili nel caso $N = 3$.

TABELLA 15.IV.

1	2	3
12	21	31
+ 121	+ 212	+ 312
+ 123	+ 213	+ 313
13	23	32
+ 131	+ 231	+ 321
+ 132	+ 232	+ 323

Dalla tabella risulta evidente che, tra le 12 soluzioni, alcune sono tra loro simili, nel senso che derivano una dall'altra con una permutazione ciclica dei simboli base. Mediante le due permutazioni illustrate nella figura 15.4,



Figura 15.4.

esse vengono ricondotte a due sole sequenze tra loro diverse ("123" e "121"). Un programma, che genera un unico elemento di ogni gruppo di 6 soluzioni simili, si può ottenere arrestando il processo non appena si deve modificare $S[2]$. Come soluzione del problema ampliato, si ottiene il programma

```

var S: array [1 .. N] of char;
m: integer; good: Boolean;
procedure extend;
begin m := m + 1; S[m] := '1' end;
procedure change;
begin {cf. (15.56)} end;
procedure check;
begin {cf. (15.64)} end;
procedure print;
var i: integer;
begin for i := 1 to N do write(S[i]);
      writeln
end;
begin m := 2; S[1] := '1'; S[2] := '2'; good := true;
repeat if good then
  if m = N then begin print; change end
  else extend
  else change;
  check
until m = 2
end.
    
```

(15.61)

La tabella 15.V riporta il numero K di soluzioni in funzione di N (è già stato considerato il fattore di riduzione 6).

TABELLA 15.V.

N	$K(N)$	N	$K(N)$	N	$K(N)$
3	2	9	18	15	103
4	3	10	24	16	133
5	5	11	34	17	174
6	7	12	44	18	232
7	10	13	57	19	305
8	13	14	76	20	398

15.5. Problemi

15.1. Il programma (15.17) (soluzione di un sistema di equazioni lineari) può essere semplificato, se si considera la variabile B come la $n+1$ -esima colonna della matrice A :

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \cdots & & & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{pmatrix} \quad (15.62)$$

Scrivere il programma corrispondente a tale semplificazione. (Si osservi che i due programmi descrivono lo stesso algoritmo).

15.2. Estendere il programma (15.17), aggiungendo una opportuna scelta del pivot:

versione 1: nel passo di eliminazione k -esimo si sceglie come pivot l'elemento $a_{kk}^{(k)}$ della k -esima riga, avente il massimo valore assoluto, cioè tale che:

$$|a_{kk}| \geq |a_{ik}| \quad \text{per ogni } i = 1 \dots n \quad (15.63)$$

Suggerimento: dopo aver determinato l'elemento pivot $a_{kk}^{(k)}$, scambiare le righe $a_k^{(k)}$ e $a_k^{(k)}$ e contemporaneamente le componenti $b_k^{(k)}$ e $b_k^{(k)}$.

versione 2: nel passo di eliminazione k -esimo si sceglie come pivot l'elemento $a_{hm}^{(k)}$, avente il massimo valore assoluto fra le componenti della k -esima riga e della k -esima colonna, cioè tale che:

$$|a_{hm}| \geq |a_{ij}| \quad \text{per ogni } i, j = k \dots n \quad (15.64)$$

Suggerimento: dopo aver determinato l'elemento pivot, scambiare le righe $a_h^{(k)}$ e $a_k^{(k)}$ (e $b_h^{(k)}$ e $b_k^{(k)}$) e le colonne $a_{xm}^{(k)}$ e $a_{xk}^{(k)}$. È però necessario memorizzare gli scambi fatti sulle colonne, poiché essi comportano una permutazione delle incognite x_m e x_k ; nella fase di risostituzione dev'essere eseguita la permutazione inversa. Se infine nessun elemento $a_{hm} > 0$, allora il sistema di equazioni è singolare e non ha soluzione; per questo caso occorre prevedere una istruzione di salto (v. (14.27)) opportuna.

15.3. Estendere il programma (15.17) per risolvere un sistema di equazioni lineari, in modo da aumentare la precisione dei risultati di un opportuno fattore di scala:

versione 1: nel k -esimo passo di eliminazione, tutte le componenti della riga $a_{ij}^{(k)}$ vengono moltiplicate per il medesimo fattore di scala $s_i^{(k)}$; si osservi che il valore delle soluzioni non viene influenzato dal fattore di scala; si scelga:

$$s_i^{(k)} = 1 / \sum_{j=k}^n a_{ij}^{(k)} \quad \text{for } i = k, \dots, n \quad (15.65)$$

versione 2: nel k -esimo passo di eliminazione vengono moltiplicati tutti gli elementi della colonna j -esima per il medesimo fattore di scala $s_j^{(k)}$; si scelga:

$$s_j^{(k)} = 1 / \sum_{i=k}^n a_{ij}^{(k)} \quad \text{for } j = k, \dots, n \quad (15.66)$$

e si ricordi che bisogna correggere di conseguenza i risultati x_j nella fase di risoluzione; suggerimento: per evitare che si sommino errori di arrotondamento contemporanei, in un'aritmetica in virgola mobile con base b è opportuno usare al posto di s_j e di s_i i fattori di scala $s'_j = N(s_j)$ e $s'_i = N(s_i)$, dove la funzione N è definita nel modo seguente:

$$N(x) = b^{\frac{\text{trunc}}{\text{trunc}}(\log_b(\text{abs}(x)))} \quad (15.67)$$

15.4. Costruire un programma per risolvere il sistema di equazioni

$$\begin{aligned} a_{11} * x_1 + a_{12} * x_2 &= b_1 \\ a_{k,k-1} * x_{k-1} + a_{kk} * x_k + a_{k,k+1} * x_{k+1} &= b_k \quad \text{for } k = 2, \dots, n-1 \\ a_{n,n-1} * x_{n-1} + a_{nn} * x_n &= b_n \end{aligned} \quad (15.68)$$

Rappresentando i coefficienti a_{ij} in una matrice, si ottiene la *matrice tri-diagonale* della figura 15.5.

$$\left(\begin{array}{cccccc} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & 0 \\ & a_{32} & a_{33} & a_{34} & & \\ & & \ddots & \ddots & \ddots & \\ 0 & & & & a_{n,n-1} & a_{nn} \end{array} \right)$$

Figura 15.5.

Suggerimento: nel k -esimo passo di eliminazione, sono da calcolare soltanto le componenti

$$a_{k+1,k+1}^{(k+1)} = a_{k+1,k+1}^{(k)} - (a_{k+1,k}^{(k)} * a_{k,k+1}^{(k)}) / a_{kk}^{(k)} \quad (15.69)$$

$$b_{k+1}^{(k+1)} = b_{k+1}^{(k)} - (a_{k+1,k}^{(k)} * b_k^{(k)}) / a_{kk}^{(k)}$$

mentre tutte le altre componenti rimangono invariate. Si rappresenti la matrice triagonale con l'*array*:

`var A: array [1..n, -1..1] of real` (15.70)

dove le componenti $a_{ij}^{(k)}$ siano date da $A[i, j-i]$, occupando soltanto lo spazio di memoria necessario per contenere gli elementi delle tre diagonali diverse da 0. Spiegare perché, in questo caso speciale, la scelta di un *pivot* sarebbe svantaggiosa.

15.5. Scrivere un programma per la soluzione di un sistema di equazioni del tipo:

$$\sum_{j=1}^i a_{ij} * x_j = b_i \quad \text{for } i = 1, \dots, n \quad (15.71)$$

Questo problema può essere trattato come caso particolare del problema della soluzione di un sistema di equazioni lineari con n incognite, dove i coefficienti formano una matrice $n \times n$ di forma triangolare (v. fig. 15.6).

$$A = \left(\begin{array}{cccccc} a_{11} & & & & & \\ a_{21} & a_{22} & & & & 0 \\ a_{31} & a_{32} & a_{33} & & & \\ \dots & & & & & \\ a_{n1} & \dots & & & & a_{nn} \end{array} \right)$$

Figura 15.6.

Questo caso si presenta assai spesso, trattando equazioni differenziali e sistemi di equazioni differenziali. Si osservi che, in questo caso, l'algoritmo di Gauss può essere semplificato in modo significativo e che è necessario memorizzare soltanto $(n/2)*(n+1)$ coefficienti. Conformemente a ciò, la matrice a_{ij} viene rappresentata da un *array*

`A: array [1..m] of real m = ½n(n + 1)` (15.72)

e i coefficienti $a_{ij}^{(k)}$ sono dati dalle componenti $A[i*(i-1) \text{ div } 2 + j]$. Si cerchi di applicare il metodo della costruzione sistematica per passi successivi.

15.6. Ampliare il programma (15.33), in modo che siano determinati i primi dieci numeri che si possono esprimere come somme di due coppie distinte di numeri elevati al cubo (invece del più piccolo solamente). Queste dieci coppie di somme siano inoltre linearmente indipendenti, cioè non deve esistere alcun n tale che:

$$a_i = n * a_j \quad \text{and} \quad b_i = n * b_j \quad i \neq j \quad (15.73)$$

15.7. Si consideri il seguente programma, soluzione del problema posto nel paragrafo [15.2]:

```

var i, j, m, n, x: integer; p: array [0 .. 13] of integer;
begin m := 0; p[0] := 0;
  while m < 13 do
    begin m := m + 1; p[m] := m * m * m; n := 0;
      while n < m do
        begin n := n + 1; x := p[m] + p[n]; i := m - 1;
          {x è il candidato successivo}
          while 2 * p[i] > x do
            begin j := i - 1;
              while p[i] + p[j] > x do j := j - 1;
              if p[i] + p[j] = x then go to 99 else i := i - 1
            end
          end
        end;
      99: write(x, m, n, i, j)
    end.
  
```

Ricostruire i passi di specificazione e le condizioni di verifica di questo programma. Anche se il programma fornisce il risultato corretto 1729, può darsi che il suo autore abbia utilizzato una condizione difficile da dimostrare, presunta valida (quale?). Il programma può allora essere considerato come una soluzione corretta ottenuta da un ragionamento sbagliato. Paragonare il tempo di calcolo impiegato da questo programma con quello impiegato dal programma (15.33), modificati per il caso delle potenze quarte ($x^4 = a^4 + b^4 = c^4 + d^4$).

15.8. Costruire un programma che calcola i primi dieci numeri che, elevati al cubo, siano eguali alla somma di tre numeri naturali elevati al cubo. Le somme devono essere, anche qui, linearmente indipendenti.

$$x_i^3 = a_i^3 + b_i^3 + c_i^3 \quad \text{for } i = 1, \dots, 10 \quad (15.75)$$

15.9. Determinare, analizzando (ciò significa senza l'aiuto del calcolatore) il programma (15.46), quale conseguenza ha la modifica di

if $\text{square} \leq x$ then (15.76)

in
 if $\text{square} < x$ then
 15.10. Modificando il programma (15.46) in accordo a (15.76), e cambiando contemporaneamente

while $n < \text{lim}$ do
 in
 while $n \leq \text{lim}$ do

il programma (15.46) risulta errato. Qual è l'invariante dimenticato, e perciò errato? Come si può correggere facilmente il programma cambiando le due modifiche? Studiare se, in tal modo, si può ottenere una versione più efficiente o più ampiamente utilizzabile dell'algoritmo.

15.11. La biimplicazione

$$x \neq a \Leftrightarrow x \text{ non è divisibile per } p$$

vale nel programma (15.46), solo se valgono le due premesse

$$x \leq a < x + p \quad \text{and} \quad a = n * p \quad (15.77)$$

Verificare, inserendo le condizioni di verifica rilevanti, che tali premesse sono invarianti nel segmento di programma (estratto da (15.46)) seguente, nel caso in cui sia $p > 2$ e intero.

```

x := 1; a := 0;
repeat x := x + 2;
  if a < x then a := a + p;
  { $x \leq a < x + p, a = n * p$ }
until P
  
```

(15.78)

15.12. Scrivere un programma che genera (in successione crescente) i primi 100 numeri dell'insieme \mathcal{M} , dove \mathcal{M} è definito nel modo seguente:

- 1) 1 appartiene ad \mathcal{M} ;
- 2) se x appartiene a \mathcal{M} , appartengono a \mathcal{M} anche $y = 2*x + 1$ e $z = 3*x + 1$;
- 3) appartengono a $\mathcal{M} = \{1, 3, 4, 7, 9, 10, 13, \dots\}$ solo i numeri ottenuti con le regole 1) e 2).

15.13. La figura 15.7 rappresenta un anello di 2^3 zeri e uni, dove compare esattamente una volta ognuna delle sottosequenze possibili di 3 simboli.

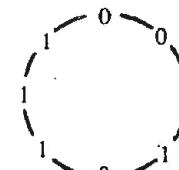


Figura 15.7.

Costruire un programma che generi un anello analogo contenente tutte le possibili sottosequenze di n cifre binarie, formato quindi da 2^n simboli. In tale esercizio è possibile applicare rigorosamente i principi dello sviluppo sistematico di un programma per passi successivi.

A1

Il linguaggio di programmazione PASCAL

A.1. Simboli base del linguaggio

<i>A...Z, a...z</i>	caratteri
<i>0 1 2 3 4 5 6 7 8 9</i>	cifre
<i>+ - * / div mod</i>	operatori aritmetici
<i>∨ ∧ ¬</i>	operatori logici
<i>= ≠ < ≤ ≥ > in</i>	operatori di relazione
<i>()</i>	parentesi
<i>[]</i>	parentesi di indice
<i>{ }</i>	parentesi di commento
begin end	parentesi di istruzioni
:=	operatore di assegnamento
/	operatore
, ;	simboli di interruzione
↑	puntatore (<i>pointer</i>)
if then else case of with	separatori di istruzioni
while do repeat until for to	
const type var procedure function	specificatori di classi di oggetti
array file record set	specificatori di classi di strutture
nil	puntatore zero
goto label	operatore di salto, dichiaratore di indirizzo

A.2. Grandezze standard

Costanti:	<i>false, true, eol</i>	(8.1) (8.3)
Tipi:	<i>Boolean, integer, char, real, text</i>	(8.1 ÷ 8.4, 10.4)
Variabili:	<i>input, output</i>	(10.4)
	<i>abs, sqr, odd</i> (valore assoluto, quadrato, dispari)	
	<i>succ, pred</i>	(8)
	<i>ord, chr</i>	(8.3)
	<i>trunc, eof, eoln</i>	(8.4, 10.3)
Funzioni:	<i>sin, cos, exp, ln, sqrt, arctan</i>	
Procedure:	<i>get, put, reset, rewrite</i>	(10.2-10.3)
	<i>read, write</i>	(10.4)

A.3. Operatori

A.3.1. Operatori di relazione (priorità inferiore)

$= \neq$ operandi qualsiasi, risultato Boolean
 $< \leq \geq >$ operandi scalari, risultato Boolean

A.3.2. Operatori additivi:

$+$ $-$ operandi numerici (8.2), (8.4)
 \vee operando booleano (OR) (8.1)

A.3.3. Operatori moltiplicativi:

$*$ operando numerico (8.2), (8.4)
 $/$ operando numerico, risultato *real* (8.4)
div divisione intera, risultato *integer* (8.2)
mod risultato della divisione intera (8.2)
 \wedge operando booleano (AND) (8.1)

A.3.4. Operatori monadici

\neg negazione; operando booleano (NOT) (8.1)

A.4. Rappresentazione dei programmi PASCAL nell'alfabeto ASCII ristretto.

A.4.1. Si usano solo i caratteri maiuscoli.

A.4.2. I simboli base forniti da parole inglesi (per esempio `begin`) sono rappresentati come tali e non possono essere usati come identificatori in un programma. Fra uno di tali simboli base e il successivo simbolo base, o identificatore, dev'essere inserito almeno uno spazio bianco.

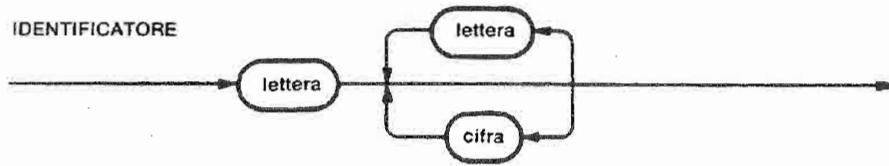
A.4.3. I simboli PASCAL non contenuti nell'alfabeto ASCII vengono tradotti nel modo indicato nella tabella A.I.

TABELLA A.I.

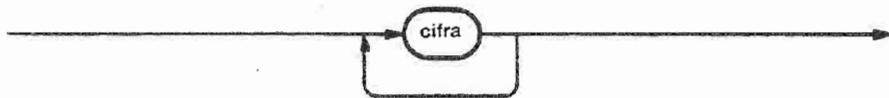
PASCAL Symbol	Corresponding ASCII Character(s)
<code>V</code>	<code>OR</code>
<code>&</code>	<code>AND</code>
<code>¬</code>	<code>NOT</code>
<code>≠</code>	<code><></code>
<code>≤</code>	<code><=</code>
<code>≥</code>	<code>>=</code>
<code>{</code>	<code>(*</code>
<code>}</code>	<code>*)</code>

A.5. SINTASSI

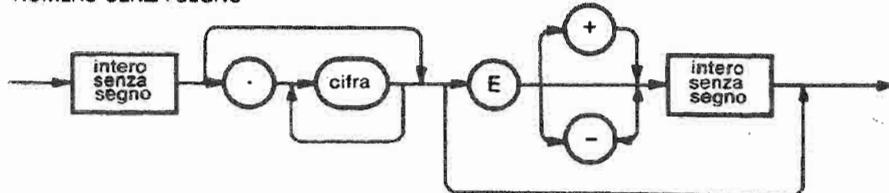
IDENTIFICATORE



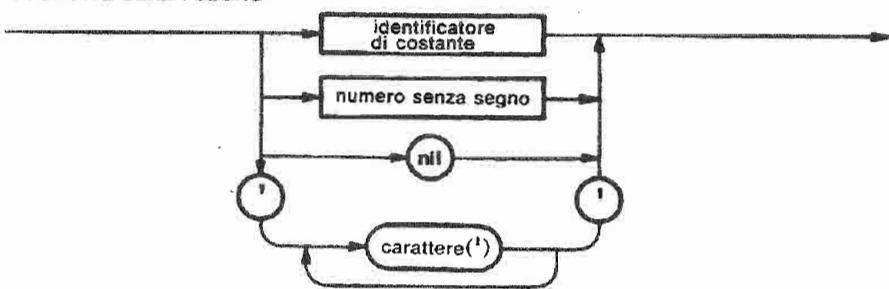
INTERO SENZA SEGNO



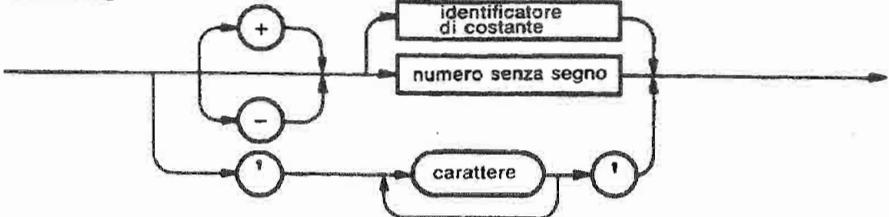
NUMERO SENZA SEGNO



COSTANTE SENZA SEGNO

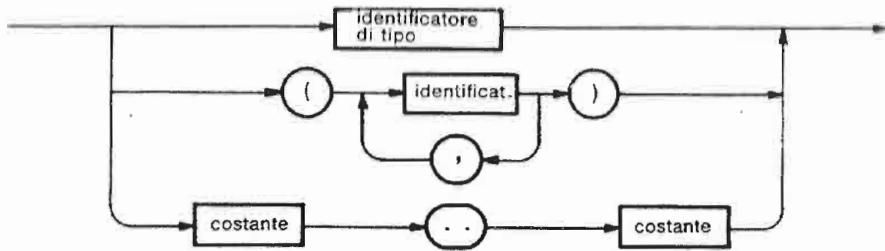


COSTANTE

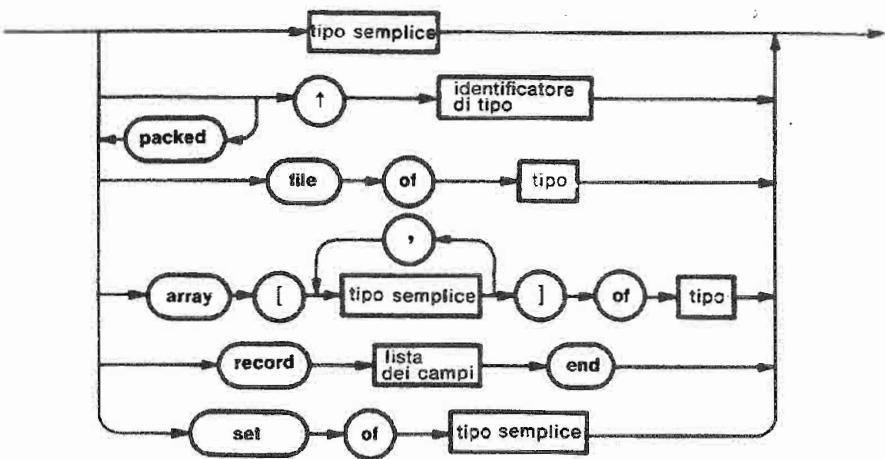


(¹) Il termine carattere sta per uno qualsiasi dei caratteri grafici elencati nell'appendice B. Negli schemi seguenti compaiono i termini identificatore (di costante), identificatore (di tipo), ecc.; si tratta di normali identificatori — cioè secondo lo schema sintattico IDENTIFICATORE — e l'attributo di costante, di tipo, ecc. indica in particolare che essi identificano una costante, un tipo, ecc.

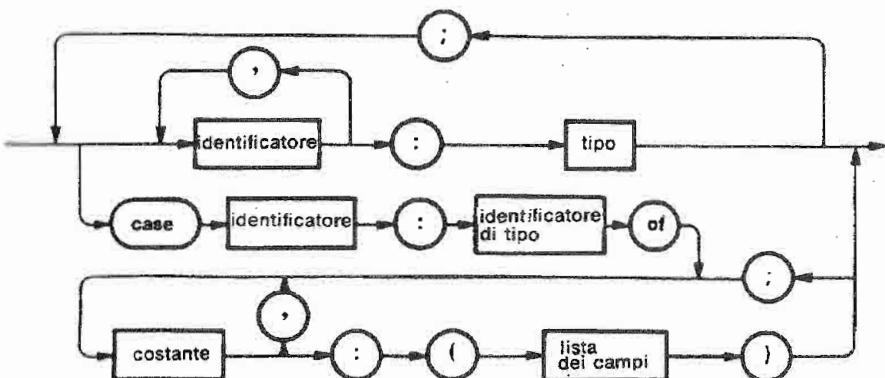
TIPO SEMPLICE



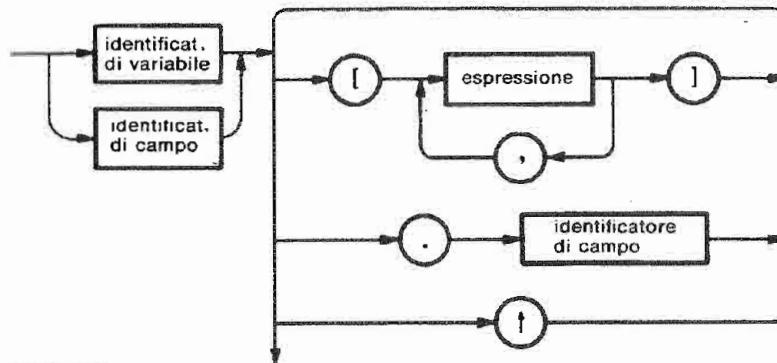
TIPO



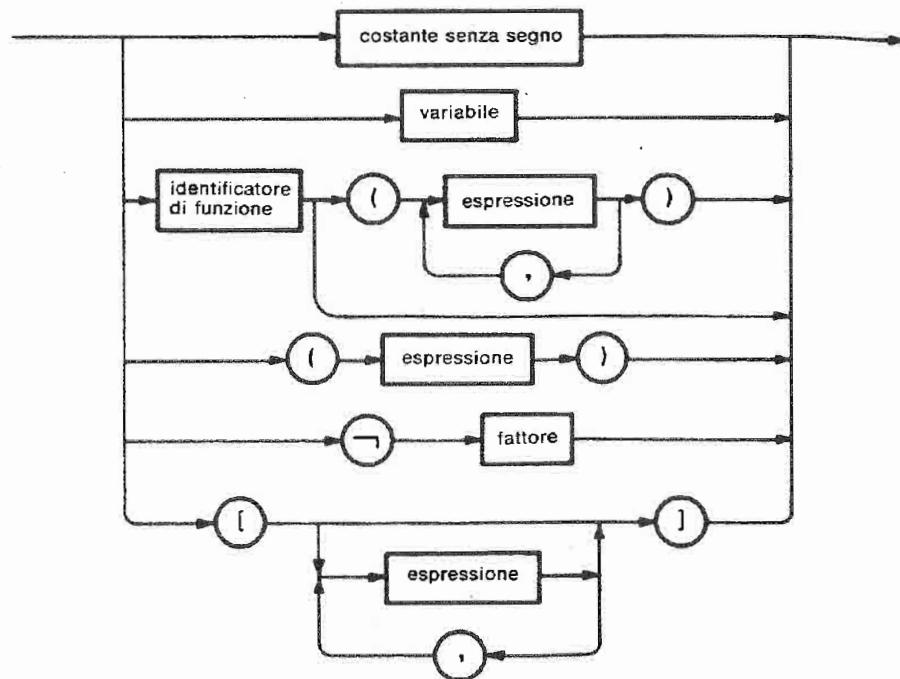
LISTA DEI CAMPI



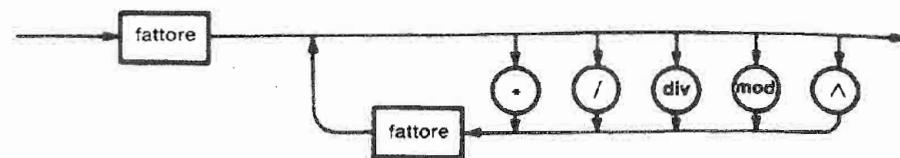
VARIABILE



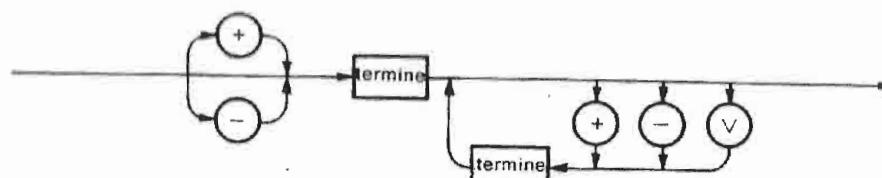
FATTORE



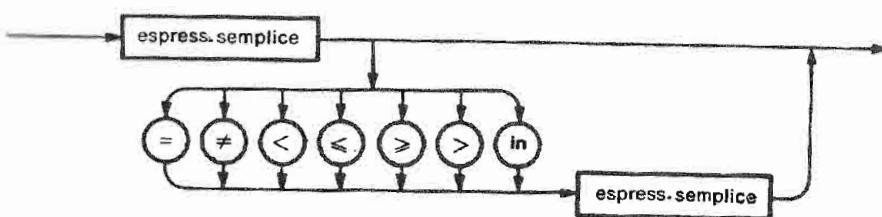
TERMINI



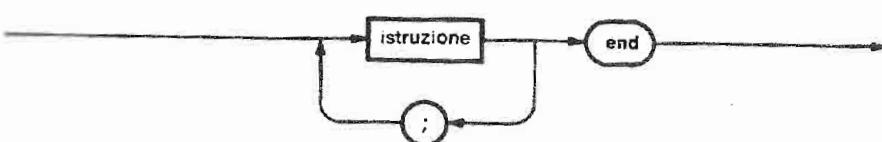
ESPRESSIONE SEMPLICE



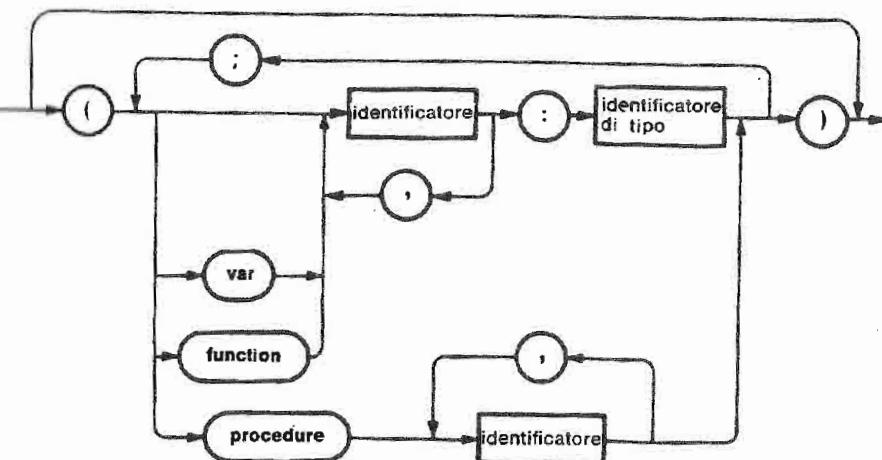
ESPRESSIONE



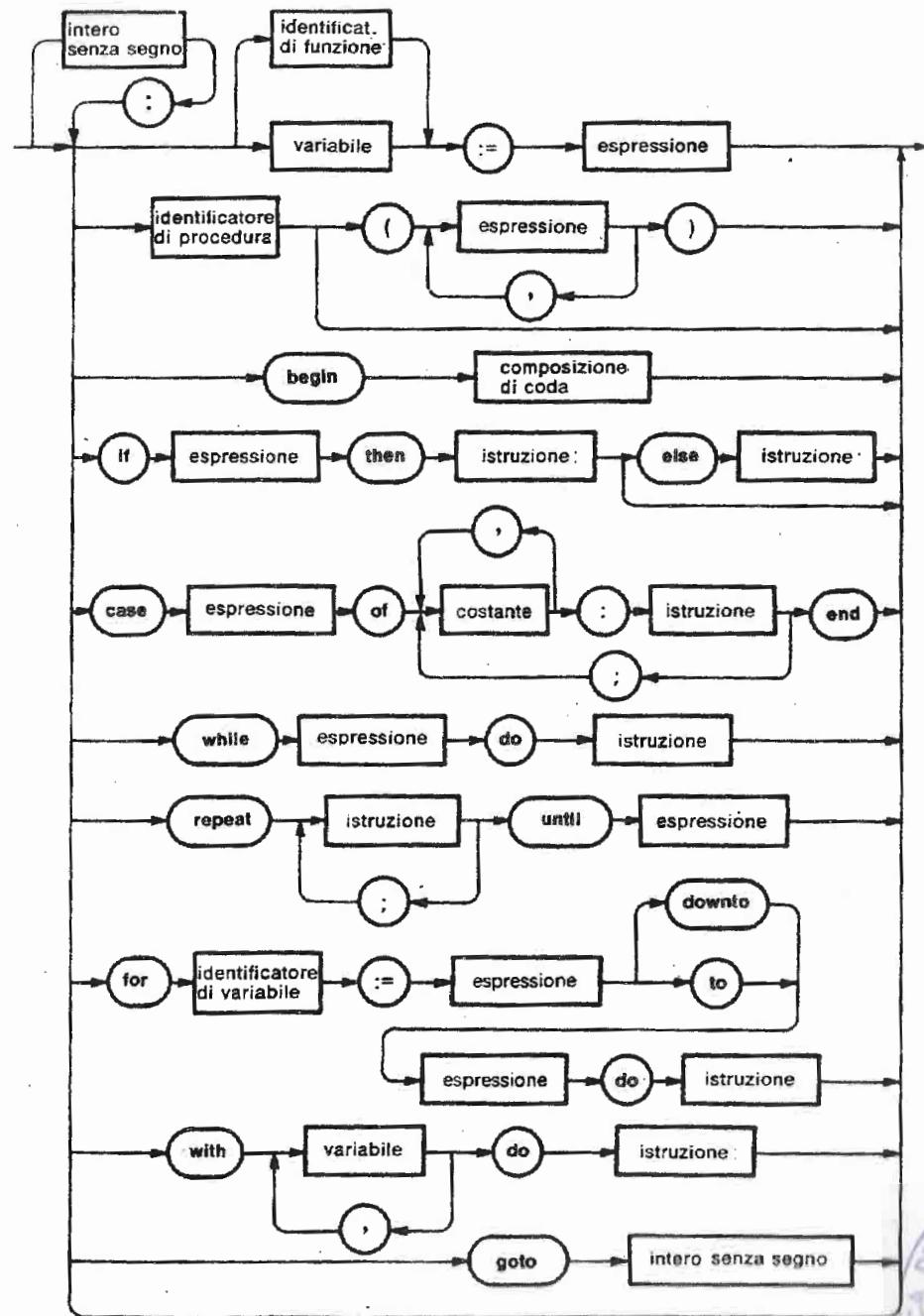
COMPOSIZIONE DI CODA



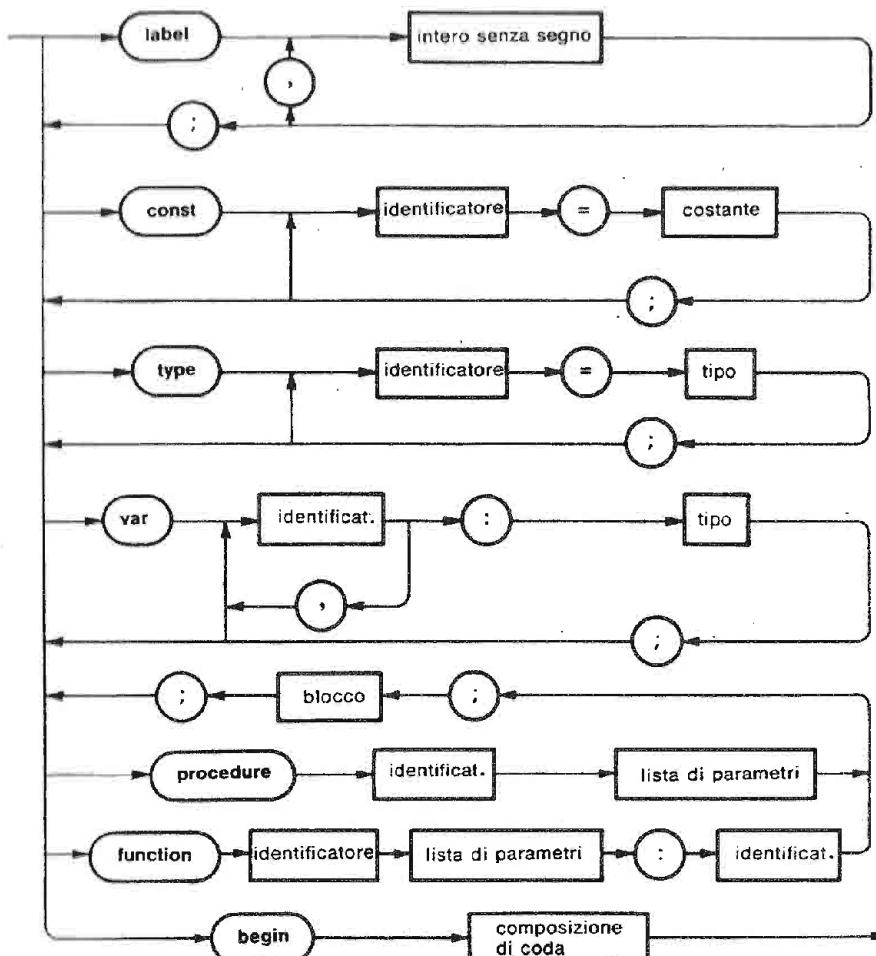
LISTA DEI PARAMETRI



ISTRUZIONE



BLOCCO



PROGRAMMA



A2

L'alfabeto ASCII

B.1. Rappresentazione su nastro perforato

b6	0	0	0	0	1	1	1	1
b5	0	0	1	1	0	0	1	1
b4	0	1	0	1	0	1	0	1
b3-b0								
0000	nul	dle		Ø	@	P	'	p
0001	soh	dcl	!	1	A	Q	a	q
0010	stx	dc2	"	2	B	R	b	r
0011	etx	dc3	#	3	C	S	c	s
0100	eot	dc4	\$	4	D	T	d	t
0101	enq	nak	%	5	E	U	e	u
0110	ack	syn	&	6	F	V	f	v
0111	bel	etb	'	7	G	W	g	w
1000	bs	can	(8	H	X	h	x
1001	ht	em)	9	I	Y	i	y
1010	lf	sub	*	:	J	Z	j	z
1011	vt	esc	+	;	K	[k	{
1100	ff	fs	,	<	L	\	l	-
1101	cr	qs	-	=	M]	m	}
1110	so	rs	.	>	N	^	n]
1111	si	us	/	?	O	_	o	del

caratteri di controllo caratteri grafici
 gruppo ristretto dei caratteri ASCII

! " # % & ' () * + , - . / Ø 1 2 3 4 5 6 7 8 9 ; : < = > ? @ 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 [\] ^ ~
 b6 b5 b4 b3 b2 b1 b0

B.2. Significato dei simboli di controllo per trascrizione di dati

LAYOUT CHARACTERS	ESCAPE CHARACTERS
bs backspace	so shift-out
ht horizontal tabulation	si shift-in
lf line feed	esc escape
vt vertical tabulation	
ff form feed	
cr carriage return	
	MEDIUM CONTROL CHARACTERS
IGNORE CHARACTERS	bel ring bell
	dcl-
nul null character	dc4 device control
can cancel	
sub substitute	em end of medium
del delete	
SEPARATOR CHARACTERS	
fs file separator	soh start of heading
gs group separator	stx start of text
rs record separator	etx end of text
us unit separator	eot end of transmission
	enq enquiry
	ack acknowledgement
	nak negative acknowledgement
	dle data link escape
	syn synchronous idle
	etb end of transmission block

Bibliografia

- [1] BAUER, F. L., GOOS, G.; *Informatik*, vol. I e II, Heidelberg Taschenbücher n. 80 e 91, Berlino, Heidelberg, New York 1970/71.
- [2] BAYER, G.; *Einführung in das Programmieren in ALGOL*, 2^a ed., Berlino 1971.
- [3] HEINRICH, W., STUCKY, W.; *Programmierung mit ALGOL 60*, Teubner Studienschriften n. 5, Stoccarda 1971.
- [4] KNUTH, D. E.; *The Art of Computer Programming*, vol. I, *Fundamental Algorithms*, vol. II *Seminumerical Algorithms*, Reading, Mass. 1968/69.
- [5] MÜLLER, D.; *Programmierung elektronischer Rechenanlagen*, BI-Hochschultaschenbücher n. 49, 3^a ed. Mannheim 1969.
- [6] NAUR, P. (a cura di); "Revised Report on the Algorithmic Language ALGOL 60" in *Comm. ACM*, 6 (1963), pp. 1-17; *Comp. J.*, 5 (1962/63) pp. 349-367; *Num. Math.*, 4 (1963), pp. 420-453.
- [7] NEUHOLD, E. J., LAWSON H. W.; *The PL/I Machine: an Introduction to Programming*, Reading, Mass. 1971.
- [8] RUTISHAUSER, H.; *Description of ALGOL 60*, Berlino, Heidelberg, New York 1967.
- [9] SAMMET, J.; *Programming Languages: History and Fundamentals*, Englewood Cliffs, N. Y. 1968.
- [10] SPIESS, RHEINGANS; *Einführung in das Programmieren in FORTRAN*. 3^a ed., Berlino 1973.
- [11] WEINBERG, G. M.; *PL/I Programming: A Manual of Style*, New York 1970.
- [12] WIRTH, N.; "The Programming Language PASCAL", *Acta Informatica*, 1 (1971), pp. 35-63.
- [13] HOARE, C. A. R., WIRTH, N.; "An Axiomatic Definition of the Programming Language PASCAL", *Acta Informatica*, 2, pp. 335-355 (1973).
- [14] SILVESTRI, A.; *Linguaggi di programmazione*, ISEDIL, in preparazione.
- [15] CRESPI REGHIZZI, S., DELLA VIGNA, P. L., GHEZZI, C.; *Linguaggi formali e compilatori*, ISEDIL, Milano, 1978.

A oltre vent'anni dalla sua pubblicazione, il testo di Niklaus Wirth rimane un classico della programmazione, intesa come «una disciplina autonoma, riguardante i metodi di formulazione e di costruzione degli algoritmi». Scopo del volume è l'insegnamento di un metodo rigoroso generale, formalizzato di programmazione, non di uno specifico linguaggio.

«La programmazione – afferma l'autore nella prefazione – è un'attività di costruzione e di sintesi, nella quale si impara molto dall'esperienza e dai propri errori»: per questo motivo il testo è corredata di problemi ed esercizi che consentono allo studente di mettere alla prova la propria conoscenza delle tecniche e dei metodi di programmazione strutturata.

Niklaus Wirth è nato a Winterthur, in Svizzera. Ha studiato al Politecnico di Zurigo, a Quebec e a Berkeley. Dal 1963 al 1967 è stato professore associato alla Stanford University, in California. Dal 1968 insegna Informatica al Politecnico di Zurigo.

€ 19,00