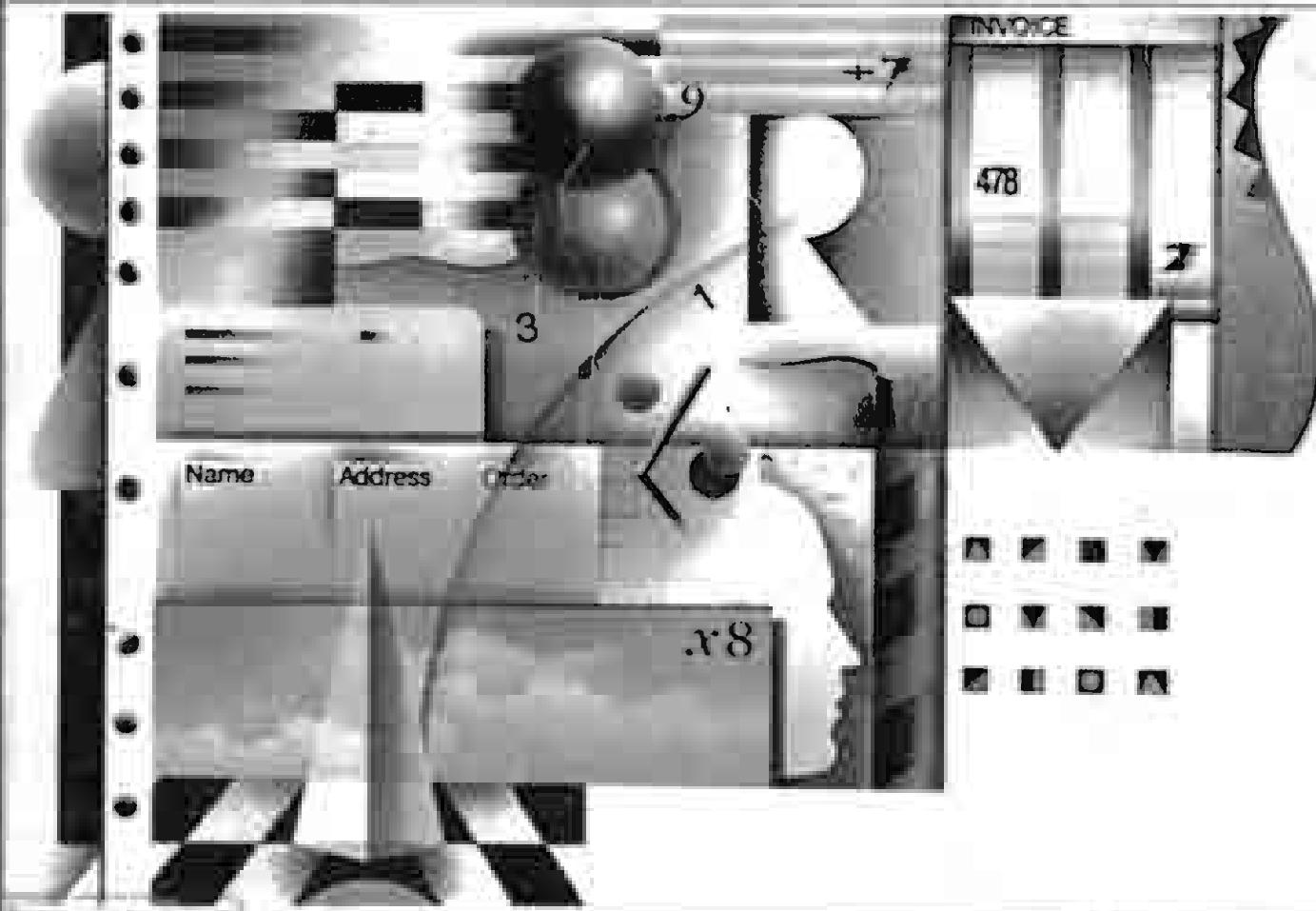


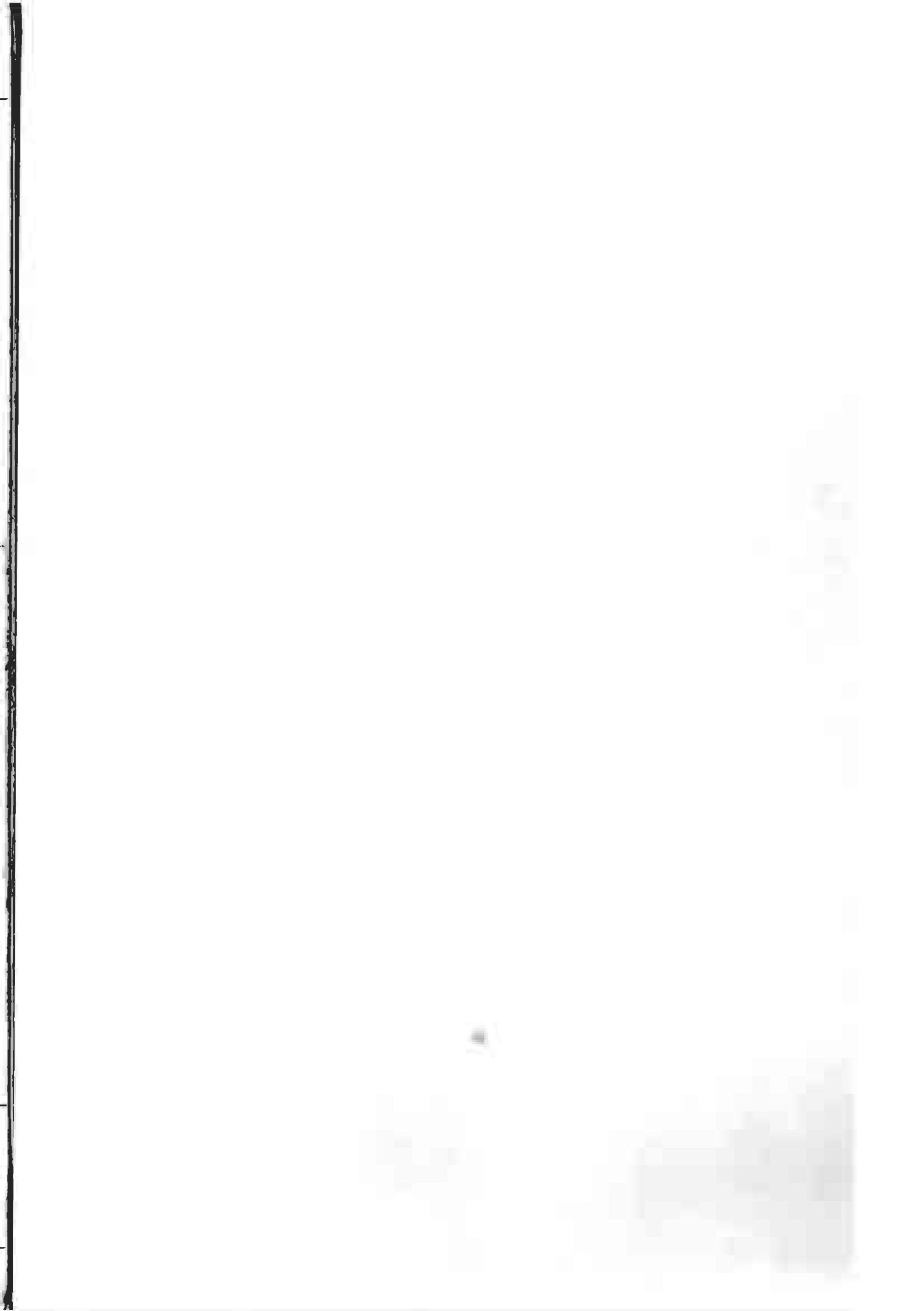
C. BATINI, L. CARLUCCI AIELLO, M. LENZERINI,
A. MARCHETTI SPACCAMELA, A. MIOLA

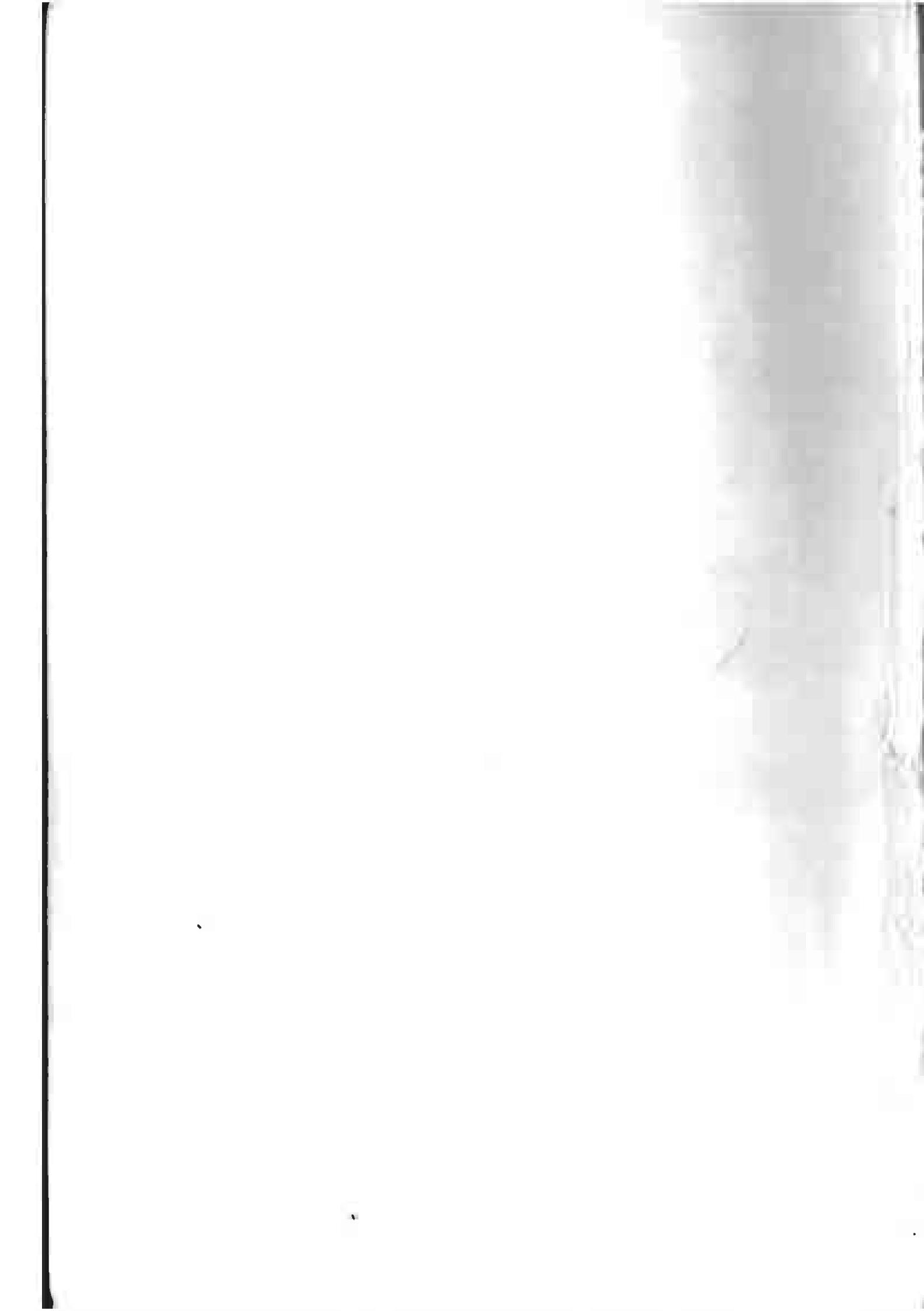
FONDAMENTI DI PROGRAMMAZIONE DEI CALCOLATORI ELETTRONICI



SCIENZE E TECNOLOGIE INFORMATICHE

FrancoAngeli





C. BATINI, L. CARLUCCI AIELLO, M. LENZERINI,
A. MARCHETTI SPACCAMELA, A. MIOLA

FONDAMENTI DI PROGRAMMAZIONE DEI CALCOLATORI ELETTRONICI

FrancoAngeli

Carlo Batini, laureato in ingegneria elettronica a Roma, è professore ordinario presso il Dipartimento di informatica e sistemistica dell'Università di Roma "La Sapienza", dove insegna Impianti di elaborazione alla Facoltà di ingegneria e dirige la Scuola a fini speciali di informatica. È coordinatore di progetti nazionali di ricerca e partecipa a progetti internazionali. Svolge attività di ricerca nel campo della progettazione di basi di dati, argomento in cui ha numerose pubblicazioni in ambito nazionale ed internazionale. Ha scritto diversi testi didattici nei campi della programmazione dei calcolatori elettronici, dei linguaggi programmativi e delle basi di dati relazionali.

Luigia Carlucci Aiello, laureata in matematica a Pisa, diplomata alla Scuola Normale Superiore, è professore ordinario presso il Dipartimento di informatica e sistemistica dell'Università di Roma "La Sapienza", dove insegna Intelligenza artificiale alla Facoltà di ingegneria. Svolge attività di ricerca in intelligenza artificiale, occupandosi di formalismi e metodi per la rappresentazione della conoscenza e di costruzione di sistemi intelligenti. Coordina e partecipa a diversi progetti di ricerca nazionali ed internazionali. Precedentemente ha svolto attività di ricerca e didattica presso le Università di Pisa, Stanford (California) e Ancona. Ha scritto alcuni testi didattici e numerosi articoli scientifici; è nel comitato di redazione di diverse riviste.

Maurizio Lenzerini, laureato in ingegneria elettronica a Roma, è professore ordinario presso il Dipartimento di informatica e sistemistica dell'Università di Roma "La Sapienza", dove insegna Fondamenti di Informatica nella Facoltà di ingegneria e Introduzione agli algoritmi e alla programmazione nella Scuola a fini speciali di informatica. Svolge attività di ricerca nell'area delle basi di dati e dell'intelligenza artificiale: in queste aree ha pubblicato diversi lavori in ambito nazionale ed internazionale, e partecipa a progetti di ricerca nazionali ed internazionali. Ha svolto attività di ricerca presso l'Università di Toronto. Ha scritto diversi testi didattici nei campi della programmazione dei calcolatori elettronici e delle basi di dati.

Alberto Marchetti Spaccamela, laureato in ingegneria elettronica a Roma, è professore ordinario presso il Dipartimento di informatica e sistemistica dell'Università di Roma "La Sapienza", dove insegna Fondamenti di informatica nella Facoltà di ingegneria. Precedentemente ha svolto attività di ricerca e didattica presso le Università della California (Berkeley, Usa) e L'Aquila. Svolge attività di ricerca nel campo del progetto di algoritmi e delle loro applicazioni alla progettazione di sistemi informatici; in queste aree ha pubblicato diversi lavori in ambito nazionale ed internazionale e partecipa a progetti di ricerca nazionali ed internazionali. Ha scritto diversi testi didattici nei campi del progetto di algoritmi e della programmazione dei calcolatori elettronici.

Alfonso Miola, laureato in matematica a Roma, è professore ordinario presso il Dipartimento di informatica e sistemistica dell'Università di Roma "La Sapienza", dove insegna Fondamenti di Informatica alla Facoltà di ingegneria e Linguaggi e metodi di programmazione nella Scuola a fini speciali di informatica. Ha svolto attività di ricerca presso il M.I.T. (Cambridge, Usa), l'Università di California (Berkeley, Usa) ed il Centro di ricerca Ibm (Yorktown, Usa). Svolge attività di ricerca nell'area della manipolazione algebrica e simbolica in cui ha diverse pubblicazioni in ambito nazionale ed internazionale. Ha scritto testi didattici ed ha curato l'edizione di testi scientifici nel campo della progettazione di linguaggi ed ambienti di programmazione per il calcolo simbolico.

7^a edizione: 1994
Ristampa invariata

Copyright © by FrancoAngeli s.r.l., Milano, Italy.
È vietata la riproduzione, anche parziale o ad uso interno o didattico, con qualsiasi mezzo effettuata, non autorizzata. Stampa Tipomonza, Viale Monza 126, Milano.

Indice

PREFAZIONE	pag.	11
<hr/>		
INTRODUZIONE: ALGORITMI E PROGRAMMI	pag.	13
1. Elaborazione automatica dell'informazione	»	13
2. Algoritmi e programmi	»	15
3. Una notazione grafica per esprimere algoritmi	»	19
4. Un linguaggio di programmazione	»	22
5. Il progetto di programmi	»	28
6. Struttura di un elaboratore	»	37
<hr/>		
PARTE I - LINGUAGGI DI PROGRAMMAZIONE		
<hr/>		
1. LINGUAGGI DI PROGRAMMAZIONE: SINTASSI E SEMANTICA	pag.	43
1. Sintassi	»	44
1.1. Linguaggi e grammatiche	»	44
1.2. Sintassi dei linguaggi di programmazione	»	51
2. Semantica	»	54
2.1. Metodo operazionale	»	55
2.2. Metodo denotazionale	»	58
<hr/>		
2. LINGUAGGI DI PROGRAMMAZIONE: DATI E CONTROLLO	pag.	63
1. Dati	»	63
1.1. Tipi di dato e rappresentazioni	»	63
1.2. Tipi di dato in Pascal	»	64

1.3. Compatibilità ed equivalenza tra tipi di dato	pag. 65
2. Controllo	» 68
2.1. Assegnazione	» 69
2.2. Strutture di controllo di base	» 72
2.3. Astrazioni funzionali: procedure e funzioni	» 72
2.4. Dichiarazioni e campo d'azione degli identificatori	» 73
2.5. Tecniche di legame dei parametri	» 76
2.6. Effetti collaterali in procedure e funzioni	» 80
2.7. Implementazione di procedure e funzioni	» 81
2.8. La ricorsione	» 86

PARTE II: STRUTTURE DI DATI

3. STRUTTURE DI DATI	pag. 91
1. Tipi astratti di dato	» 91
1.1. La specifica dei tipi astratti	» 93
1.2. La rappresentazione dei tipi astratti	» 96
2. I vettori e le matrici	» 103
2.1. Il tipo <i>matrice</i> nei linguaggi di programmazione	» 103
2.2. Rappresentazioni compatte di matrici	» 105
3. Le liste	» 112
3.1. Le liste semplici	» 112
3.2. La rappresentazione sequenziale	» 114
3.3. La rappresentazione collegata	» 119
3.4. Alcune varianti della rappresentazione collegata	» 133
3.5. Le liste composite	» 134
3.6. Recupero della memoria	» 138
4. Gli insiemi	» 139
4.1. Metodi di rappresentazione di insiemi	» 141
4.2. Confronto tra i metodi di rappresentazione di insiemi	» 144
5. Pile e code	» 145
5.1. Le pile	» 146
5.2. Le code	» 150
6. Gli alberi	» 154
6.1. Gli alberi binari	» 154
6.2. La rappresentazione sequenziale	» 157
6.3. La rappresentazione mediante lista	» 160
6.4. Gli alberi binari di ricerca	» 165

6.5. Gli alberi N-ari	pag. 167
7. I grafi	» 174
7.1. Rappresentazione di grafi	» 176
7.2. Il problema del percorso più breve in un grafo	» 185
8. Le tavole	» 189
8.1. Rappresentazioni sequenziali	» 191
8.2. Rappresentazioni collegate e rappresentazioni ad albero	» 194
8.3. Rappresentazione con funzioni di accesso	» 195

PARTE III: METODOLOGIE DI PROGRAMMAZIONE

• 4. METODOLOGIE DI PROGETTO DI PROGRAMMI	pag. 201
1. Le qualità di un programma	» 202
2. Principi di progetto	» 205
3. Metodologie di progetto	» 208
4. Criteri di modularizzazione	» 214
• 5. La programmazione strutturata	» 217
6. La documentazione	» 222
7. Una metodologia di progetto	» 232
7.1. Analisi dei requisiti	» 232
7.2. Raffinamento iterativo dell'algoritmo	» 232
7.3. Analisi dell'algoritmo	» 233
7.4. Ristrutturazione	» 235
• 5. ANALISI DI PROGRAMMI: LA CORRETTEZZA	pag. 236
1. Correttezza ed errori di un programma	» 236
1.1. Correttezza di un programma	» 236
1.2. Classificazione degli errori	» 239
2. Prove formali di correttezza	» 242
2.1. Correttezza di programmi senza procedure	» 242
2.2. Correttezza di programmi con procedure	» 251
2.3. Controllo di eventuali errori dei dati di ingresso	» 255
3. Test di un programma	» 259
3.1. Metodi basati sulle specifiche	» 261
3.2. Metodi basati sulla struttura del programma	» 262
4. Il test dei moduli	» 267
4.1. Test top-down	» 268
4.2. Test bottom-up	» 270

5. La correzione degli errori

pag. 271

6. ANALISI DI PROGRAMMI: LA COMPLESSITÀ	pag. 277
1. Efficienza dei programmi	» 277
1.1. Il modello di costo	» 278
1.2. Comportamento asintotico	» 282
2. La complessità di un programma e di un problema	» 283
2.1. Le notazioni O e Ω	» 283
2.2. Valutazione della complessità di un programma	» 286
2.3. Istruzione dominante	» 288
2.4. Delimitazioni alla complessità di un problema	» 289
3. La gestione di una tavola	» 292
3.1. Gestione sequenziale	» 293
3.2. Gestione sequenziale ordinata: la ricerca binaria	» 293
3.3. Alberi binari di ricerca	» 294
3.4. Tavole Hash	» 295
4. Il problema dell'ordinamento	» 296
4.1. Ordinamento per selezione (Selectionsort)	» 297
4.2. Ordinamento a bolle (Bubblesort)	» 298
4.3. Ordinamento per fusione (Mergesort)	» 301
4.4. Ordinamento veloce (Quicksort)	» 310
4.5. Delimitazione inferiore alla complessità	» 317
4.6. Binsort	» 320
5. Conclusioni	» 322
7. ELEMENTI DI CALCOLABILITÀ	pag. 325
1. Funzioni calcolabili e modelli di calcolo	» 325
2. Un modello di calcolo basato sul Pascal	» 327
2.1. Il linguaggio mini-Pascal	» 327
2.2. Funzioni calcolabili in mini-Pascal	» 328
3. Funzioni non calcolabili	» 334
3.1. Il problema della fermata	» 335
3.2. La tesi di Church	» 338
4. Cenni storici	» 339

PARTE IV: SISTEMA DI ELABORAZIONE

8. SISTEMA DI ELABORAZIONE: ARCHITETTURA	pag. 345
1. La struttura dell'elaboratore	» 345
1.1. La memoria centrale	» 346
1.2. L'unità centrale	» 348
1.3. Il funzionamento elementare dell'elaboratore	» 349
2. Il sistema di elaborazione	» 354
2.1. Le unità di memoria ausiliaria	» 354
2.2. Le unità di ingresso e uscita	» 357
2.3. Interfacciamento di unità periferiche	» 358
2.4. Collegamenti di elaboratori in rete	» 359
3. Rappresentazione dell'informazione	» 360
3.1. Sistemi di numerazione e algoritmi di conversione	» 361
3.2. L'aritmetica intera	» 366
3.3. L'aritmetica in virgola mobile	» 374
3.4. Rappresentazione dei caratteri	» 381
9. SISTEMA DI ELABORAZIONE: SOFTWARE DI BASE	pag. 382
1. Traduzione ed esecuzione di programmi	» 382
1.1. Compilatori ed interpreti	» 383
1.2. Correlazione e caricamento	» 388
2. Sistema operativo	» 390
2.1. Tipi di sistemi operativi	» 391
2.2. La struttura dei sistemi operativi	» 392
2.3. La gestione della memoria	» 395
2.4. Conseguenze della multiprogrammazione sull'architettura	» 397
3. Ambiente di programmazione	» 398
3.1. L'editore di testi	» 399
3.2. Il debugger	» 399
X 10. LINGUAGGI DI PROGRAMMAZIONE	pag. 401
1. I linguaggi imperativi	» 403
2. I linguaggi funzionali	» 409
3. I linguaggi dichiarativi basati sulla logica	» 413

APPENDICE	pag. 419
1. Insiemi, relazioni e funzioni	» 419
1.1. Insiemi	» 419
1.2. Relazioni e funzioni	» 422
1.3. Funzione inversa, composizione di funzioni	» 425
1.4. Cardinalità di un insieme	» 426
2. Elementi di algebra	» 427
2.1. Strutture algebriche	» 427
2.2. Omomorfismi	» 427
2.3. Operazioni su Z_n	» 428
3. Grafi e alberi	» 430
3.1. Relazioni e grafi	» 430
3.2. Alberi	» 436
4. Induzione matematica	» 440
4.1. Definizione induttiva di insiemi	» 441
5. Logica matematica	» 445
5.1. Il calcolo proposizionale	» 445
5.2. Il calcolo dei predicati	» 449

BIBLIOGRAFIA

pag. 455

Prefazione

Scopo di questo libro è presentare al lettore i concetti di base relativi alla programmazione dei calcolatori elettronici. La programmazione è una disciplina che si fonda su consolidate basi di teoria dei linguaggi e di metodologie di progetto ed analisi di soluzioni algoritmiche di problemi. Quello che ci proponiamo è fornire al lettore gli strumenti linguistici, le metodologie e le tecniche, in parte formali ed in parte pragmatiche, che soggiacciono alla programmazione, sia per gli aspetti relativi alla rappresentazione dei dati che al progetto e all'analisi di algoritmi.

Il libro si apre con un'ampia introduzione dove, in forma succinta, vengono presentate al lettore tutte le problematiche relative alla programmazione che verranno affrontate nei successivi capitoli. Questi sono organizzati in quattro parti.

Parte 1: Linguaggi di programmazione. Vengono presentate nozioni teoriche sui linguaggi di programmazione (sintassi e semantica) e tecniche di implementazione, con particolare riguardo ai linguaggi imperativi, cioè quei linguaggi in cui il costrutto linguistico fondamentale è l'assegnazione (capp. 1 e 2).

Parte 2: Strutture di dati. Vengono introdotte le problematiche connesse alla rappresentazione dei dati. Questo argomento è trattato nel cap. 3, dove sono esaminati i principali tipi di dato, partendo dalla loro definizione astratta e poi illustrando e confrontando varie tecniche per la loro rappresentazione.

Parte 3: Metodologie di programmazione. Il cap. 4 illustra i metodi per la costruzione di algoritmi risolutivi di problemi, con particolare enfasi per il

metodo cosiddetto top-down. Naturalmente, nel costruire un programma si esigono alcune proprietà: che esso risolva in maniera adeguata il problema affrontato, cioè che sia corretto, e che ci fornisca i risultati usando al meglio le risorse di calcolo di cui dispone, cioè che sia efficiente. Questi sono gli argomenti trattati nei capp. 5 e 6, rispettivamente. Il cap. 7 è dedicato ad una breve trattazione dei problemi connessi ai limiti teorici del calcolo.

Parte 4: Sistema di elaborazione. I capp. 8 e 9 sono dedicati al sistema di elaborazione con particolare riguardo all'architettura e al software di base, rispettivamente. Il cap. 10 contiene infine una presentazione dei principali linguaggi di programmazione e delle loro caratteristiche salienti.

Chiude il libro una *Appendice* contenente richiami di algebra e logica, dove sono state raccolte nozioni utili in programmazione.

Nella scelta riguardante il linguaggio di programmazione cui fare riferimento abbiamo optato per il Pascal, soprattutto per le caratteristiche che esso offre per la strutturazione dei programmi e dei dati. Tuttavia è bene sottolineare che questo non è un libro di introduzione al Pascal, né tantomeno un manuale. Il nostro tentativo è quello di introdurre il lettore ai concetti fondamentali della programmazione con particolare riferimento ai linguaggi imperativi, di cui il Pascal è l'esemplare più diffuso per la didattica. Molti dei concetti e delle nozioni che in questo testo sono presentati attraverso esempi in Pascal sono facilmente generalizzabili ad altri linguaggi di programmazione.

Questo libro nasce dalla esperienza didattica degli autori in un corso introduttivo alla programmazione tenuto presso la Facoltà di ingegneria. Suggeriamo di affiancarlo con un libro di introduzione al linguaggio Pascal, con un manuale relativo al compilatore Pascal usato nelle esercitazioni pratiche, e possibilmente con un eserciziario.

Nonostante le migliori intenzioni, il libro potrebbe contenere degli errori: saremo grati a quanti ce li faranno notare.

Un sentito ringraziamento va a quanti hanno letto varie versioni del manoscritto e hanno influenzato il nostro lavoro, in particolare gli studenti del corso di Programmazione dei calcolatori elettronici della Facoltà di ingegneria dell'Università di Roma "La Sapienza", i collaboratori didattici e i colleghi: Mirella Casini Schaerf, Paolo Ercoli e Maria Teresa Pazienza.

Gli Autori

Introduzione: algoritmi e programmi

In questa introduzione presentiamo la problematica generale della elaborazione automatica della informazione. Essa sarà condotta attraverso semplici esempi, che ci permettono di introdurre i principali aspetti connessi al progetto di programmi, che saranno poi sviluppati nei capitoli successivi. Il capitolo si presenta perciò come una introduzione a tutti gli argomenti sviluppati nell'intero testo. Poiché l'elaborazione della informazione fa parte della nostra vita quotidiana, il primo paragrafo introduce la problematica attraverso esempi tratti dalla nostra esperienza. Vengono poi definiti alcuni concetti di base, quali algoritmo, programma, linguaggio programmatico; vengono a questo punto introdotti due linguaggi programmativi utilizzati per progettare programmi di crescente complessità. Il capitolo è completato con un ultimo paragrafo in cui viene descritta la struttura di un elaboratore (o anche, nel seguito, calcolatore); anche questi aspetti verranno approfonditi successivamente.

1. ELABORAZIONE AUTOMATICA DELL'INFORMAZIONE

Esaminiamo anzitutto un esempio tratto da un problema quotidiano. Quando dobbiamo cercare un numero di telefono, le fonti di informazione a nostra disposizione sono varie: usualmente una rubrica personale e un elenco telefonico. Nella rubrica e nell'elenco gli stessi tipi di informazioni sono organizzate in modi diversi. In una rubrica cognomi e numeri di telefono sono suddivisi per gruppi di lettere (ad esempio A-C, D-F, ecc.): all'interno dello stesso gruppo, i cognomi compaiono in un ordine casuale. Nell'elenco telefonico le informazioni sono ordinate per cognome; tra le persone con cognomi uguali, sono ordinate per nome. A ogni pagina è associata una intestazione formata da

alcune lettere associate al primo cognome della pagina.

Il procedimento di ricerca di un numero di telefono è influenzato dal modo in cui le informazioni sono organizzate. Se cerchiamo un cognome in una rubrica, seguiamo questo procedimento: cerchiamo la pagina nella cui intestazione compare la lettera iniziale del cognome; una volta trovata, cerchiamo il cognome nella pagina fin quando non lo troviamo oppure terminiamo l'elenco dei cognomi. Se invece usiamo un elenco telefonico, sarebbe poco efficiente eseguire una ricerca sequenziale, e seguiamo in genere un'altra strategia:

1. apriamo l'elenco in un punto approssimativamente vicino a quello in cui dovrebbe trovarsi il cognome;
2. a seconda che il cognome cercato sia prima o dopo il punto in cui siamo, si procede in avanti o all'indietro, per gruppi di pagine, fin quando non arriviamo alla pagina di interesse;
3. a questo punto cominciamo una ricerca sequenziale, fino a trovare il cognome o a verificare che manca.

Il precedente esempio mette in evidenza vari aspetti importanti, che prefigurano gli argomenti centrali affrontati in questo testo. Li esaminiamo qui di seguito:

- quando dobbiamo risolvere problemi che richiedono la ricerca e/o manipolazione di informazione (effettuata o in modo manuale o con l'uso di strumenti automatici), è necessario individuare un *metodo risolutivo*, cioè un insieme di operazioni che, eseguite ordinatamente, ci permettono di ricavare la informazione cercata (risultati, o dati di uscita) a partire da quella a nostra disposizione (dati di ingresso);
- la struttura del metodo risolutivo è strettamente legata al modo in cui le informazioni a disposizione sono organizzate: nel caso dell'elenco, la strategia seguita è resa possibile dal fatto che i cognomi sono ordinati.

Se vogliamo automatizzare la soluzione del problema, dobbiamo affrontare un passo in più, che consiste nel *rappresentare* le informazioni ed il metodo risolutivo in un linguaggio eseguibile dallo strumento di calcolo. Perciò, progettare metodi risolutivi (nel seguito *algoritmi*) e descriverli in un linguaggio comprensibile da un elaboratore (*linguaggio programmatico*) per mezzo di sequenze di istruzioni del linguaggio (*programmi*) è una attività che richiede:

1. la conoscenza delle *tecniche* che permettono di automatizzare procedimenti risolutivi di problemi, cioè i linguaggi programmativi, e i tipi di rappresentazione delle informazioni in tali linguaggi;
2. la conoscenza delle *metodologie* che permettono di usare efficacemente tali tecniche.

Lo scopo principale di questo testo è, quindi, approfondire la conoscenza delle tecnologie e delle metodologie utilizzate per elaborare automaticamente informazione. Nella nostra indagine sui linguaggi e sulle metodologie emergerà che spesso è compito molto arduo individuare metodi risolutivi di problemi. Per questa ragione, e per una comprensibile economia di sforzi ed esigenza di riutilizzare quanto possibile precedenti risultati di progetto, studiamo nel seguito metodi che permettano più che la soluzione di singoli problemi, la soluzione di *classi di problemi*. L'analisi delle metodologie, dopo una prima introduzione indipendente dal problema, verrà affrontata per diversi tipi di rappresentazione dell'informazione: dopo aver introdotto i principali tipi di rappresentazione, mostreremo diversi problemi significativi e i corrispondenti algoritmi risolutivi.

Nel corso della nostra indagine, vedremo anche che esistono problemi che non sono "trattabili" con un elaboratore. Anzitutto, esistono problemi per i quali non è possibile trovare un metodo risolutivo che ci permetta di automatizzarne la soluzione. Accanto a questa prima classe, esiste un'altra classe di problemi che, pur ammettendo metodi risolutivi, non ne hanno nessuno che, in casi significativi, sia eseguibile in tempo ragionevole. Per ragioni diverse, entrambe le precedenti classi di problemi non possono essere "trattate" con strumenti automatici.

2. ALGORITMI E PROGRAMMI

I problemi che siamo interessati a risolvere con l'ausilio di un elaboratore possono essere di natura molto varia. Non diamo una definizione di problema: preferiamo aiutareci con alcuni esempi.

Problema 1: Dati due numeri, trovare il maggiore.

Problema 2: Dato un elenco di nomi e relativi numeri di telefono (rubrica o elenco telefonico), trovare il numero di telefono di una persona.

Problema 3: Data la struttura della rete stradale di una città e le informazioni sui flussi di veicoli in ciascuna strada e in ogni momento di una giornata, trovare il percorso che permette di andare in tempo minimo dalla propria casa al posto di lavoro.

Problema 4: Scrivere tutti i valori di n per cui la equazione $X^n + Y^n = Z^n$ ha soluzioni X, Y, Z intere.

Problema 5: Decidere, per ogni x e per ogni funzione $f(x)$, se $f(x)$ è, o meno, una funzione costante.

I precedenti problemi hanno la caratteristica comune di trasformare un insieme di informazioni (*di ingresso* al problema) in nuove informazioni (*risultato* del problema). Ad esempio, nel problema 1 le informazioni in ingresso sono due numeri, e la informazione prodotta è il maggiore tra di essi; nel problema 2, le informazioni in ingresso sono una rubrica e un cognome, la informazione in uscita è un numero di telefono (o il messaggio ‘non l’ho trovato’). Riguardo al modo in cui sono formulati i precedenti problemi, si può osservare che:

- la descrizione del problema non fornisce in genere un metodo per calcolare il risultato dalle informazioni di partenza. Ad esempio, nella descrizione del problema 3 non c’è traccia della strategia con cui trovare il percorso più breve;
- la descrizione del problema è talvolta ambigua o imprecisa, nel senso che a essa possono essere associate molteplici interpretazioni. Ad esempio, che risposta occorre dare nel caso del problema 2 quando si cerca in un elenco il numero di telefono di Paolo Rossi e quindi, probabilmente, esiste più di una persona con quel nome e cognome nell’elenco?
- per alcuni problemi allo stato attuale delle conoscenze non è stato trovato un metodo risolutivo. È questo il caso del problema 4, formulato la prima volta dal matematico francese Fermat, per cui non è ancora stata individuata una soluzione;
- esistono problemi per i quali è stato dimostrato che non è possibile in generale definire un algoritmo risolutivo: questo è il caso del problema 5.

Relativamente agli ultimi due punti sollevati, nel seguito concentreremo l’attenzione su problemi che ragionevolmente ammettono un metodo risolutivo, e che sono i più interessanti da un punto di vista pratico. Nel cap. 7 ritneremo sul problema dell’esistenza o meno di metodi risolutivi per problemi, analizzando classi di problemi che non ne ammettono.

Ogni elaboratore può considerarsi come una macchina in grado di eseguire *azioni elementari* su oggetti detti *dati*. L’esecuzione delle azioni è richiesta all’elaboratore tramite comandi, detti usualmente *istruzioni*. Ad esempio, l’azione consistente nell’effettuare una somma tra due dati va richiesta per mezzo di una istruzione, detta appunto istruzione di somma. Affinché il calcolatore possa essere in grado di eseguire le azioni, è necessario che le istruzioni siano descritte in un dato *linguaggio programmatico*. Per delegare ad

un calcolatore la esecuzione di un calcolo, si deve perciò:

• individuare (almeno) un procedimento risolutivo del problema, cioè un insieme di regole che, eseguite ordinatamente, permettono di calcolare i risultati del problema a partire dalle informazioni a disposizione; chiamiamo nel seguito *algoritmo* un tale procedimento. Un algoritmo non deve necessariamente essere espresso per mezzo di istruzioni di un linguaggio programmatico; qualunque sia il linguaggio usato, perché un insieme di istruzioni possa considerarsi un algoritmo deve rispettare alcune proprietà, che sono:

1. non ambiguità - Le istruzioni devono essere univocamente interpretabili dall'esecutore dell'algoritmo (nel seguito l'esecutore sarà l'elaboratore, ma il problema si pone anche per esecutori costituiti da esseri umani);
 2. eseguibilità - L'esecutore deve essere in grado, con le risorse a disposizione, di eseguire ogni istruzione, ed in tempo finito;
 3. finitezza - L'esecuzione dell'algoritmo deve terminare in un tempo finito per ogni insieme di valori di ingresso;
- individuare una *rappresentazione* dell'algoritmo, delle informazioni a disposizione e di quelle utilizzate dall'algoritmo per mezzo di un linguaggio programmatico, composto di regole comprensibili sia per noi che per l'elaboratore. Chiamiamo *dati di ingresso* le rappresentazioni (fornite all'elaboratore) delle informazioni a disposizione, *programma* la rappresentazione dell'algoritmo nel linguaggio (un programma sarà costituito da un insieme di istruzioni), *dati di uscita* le rappresentazioni fornite dal calcolatore al termine della esecuzione del programma.

Dunque, il processo tipico per la risoluzione di un problema può essere riassunto come segue:

$$\text{problema} \rightarrow \text{algoritmo} \rightarrow \text{programma}$$

Consideriamo i due metodi risolutivi per la ricerca di un numero in una rubrica ed in un elenco. Se siamo noi l'esecutore dell'algoritmo, il metodo risolutivo descritto nel caso della rubrica può essere considerato un algoritmo, perchè:

- ogni passo è interpretabile in modo univoco;
- siamo in grado di eseguire le attività descritte (se, come diamo per scontato, conosciamo l'alfabeto, l'ordinamento tra lettere, i numeri, ecc.);
è intuitivo che la esecuzione richiederà sempre un tempo finito, poiché ogni singola operazione avvicina la soluzione finale.

Nel caso dell'elenco, invece, non è un algoritmo: infatti, viene meno la condizione di non ambiguità per la frase "in un punto approssimativamente

vicino”, e per la specifica “gruppi di pagine”.

Il processo con cui, a partire da un problema, ne individuiamo un algoritmo risolutivo, può essere molto complesso: una parte rilevante di questo testo è dedicata a fornire metodi per aiutare il progettista in tale attività. Un metodo che si dimostra molto potente nel progetto è quello di operare per *livelli di astrazione*, partendo da una versione iniziale molto generale per raffinare via via la descrizione dell’algoritmo in tutti i dettagli.

→ Mostriamo tale procedimento nel caso del calcolo del massimo comun divisore di due numeri n ed m . Nelle figg. 1 e 2 abbiamo riportato due algoritmi, espressi per mezzo di frasi in linguaggio italiano.

- Calcola l’insieme I dei divisori di m
- Calcola l’insieme J dei divisori di n
- Calcola l’insieme dei divisori comuni $k = I \cap J$
- Calcola il numero massimo nel precedente insieme
- Il massimo comun divisore è il numero trovato

Fig. 1 - Un primo algoritmo per il massimo comun divisore

L’algoritmo di fig. 2 (algoritmo di Euclide) si basa sulla seguente proprietà:

$$\text{mcd } (m, n) = \begin{cases} m \text{ o } n \text{ se } m = n \\ \text{mcd}(m-n, n) \text{ se } m > n \\ \text{mcd } (m, n-m) \text{ se } m < n \end{cases}$$

- Finché m è diverso da n esegui le seguenti azioni:
 - Se $m > n$ allora sostituisci a m il valore $m - n$
altrimenti sostituisci a n il valore $n - m$
- Il massimo comun divisore è uno qualunque dei due numeri

Fig. 2 - Secondo algoritmo per il massimo comun divisore

Entrambi gli algoritmi esprimono un metodo risolutivo del problema in termini di rappresentazioni di dati e di elaborazioni su questi dati. Nel primo

algoritmo si fa riferimento ad insiemi di dati di varia natura (l'insieme dei divisori di m, ecc.) e ad operazioni di selezione di un insieme, ricerca del massimo, ecc. Nel secondo algoritmo, i dati cui si fa riferimento sono i due valori, e le operazioni descritte sono di confronto e di aggiornamento dei due valori. Nel costruire i due algoritmi abbiamo seguito la stessa strategia; il problema è stato frammentato in attività più elementari, espresse, peraltro, in modo ancora astratto e indipendente da un particolare linguaggio programmatico.

Se l'elaboratore che utilizziamo per il calcolo non dispone, ad esempio, di una istruzione di ricerca di un massimo in un insieme, occorrerà procedere ad una ulteriore frammentazione della istruzione in termini di istruzioni più elementari, fin quando non riusciamo ad utilizzare solo istruzioni direttamente eseguibili dall'elaboratore. Ad esempio, un possibile raffinamento del calcolo del massimo è il seguente:

1. scegli un elemento come massimo provvisorio;
2. per ogni elemento i dell'insieme:
 - se i è maggiore del massimo provvisorio, eleggi i a nuovo massimo provvisorio.

Dalle precedenti osservazioni emerge chiaramente il ruolo fondamentale che i linguaggi programmatici hanno nella progettazione di programmi. Quanto più il linguaggio programmatico è espressivo, tanto più saremo aiutati nelle scelte di progetto. Le caratteristiche dei linguaggi programmatici saranno studiate nei capp. 1 e 2, mentre le caratteristiche dei metodi di progetto saranno studiate nei capp. 4, 5, e 6.

Nei paragrafi seguenti introduciamo una notazione grafica per esprimere algoritmi ed un semplice linguaggio di programmazione, col duplice scopo di mostrare un insieme di istruzioni tipiche di molti linguaggi programmatici e procedere al progetto di (semplici) algoritmi e programmi; avremo in questo modo la possibilità di introdurre le principali caratteristiche dei metodi di progetto descritti nel seguito.

3. UNA NOTAZIONE GRAFICA PER ESPRIMERE ALGORITMI

Il primo linguaggio che introduciamo, chiamato linguaggio dei "grafi di flusso", è in effetti una notazione grafica per descrivere le istruzioni e l'ordine di esecuzione tra di esse; esso non può propriamente definirsi un linguaggio programmatico, ma sarà utilizzato talvolta nel testo.

Stabiliamo per semplicità che la comunicazione (all'algoritmo) dei dati di

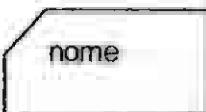
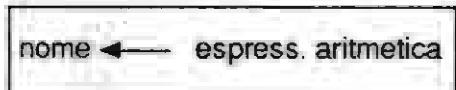
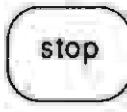
Simbolo	Significato
	Istruzione di lettura 1. Leggere da ingresso un nuovo dato e assegnare il suo valore alla variabile denominata "nome"
	Istruzione di scrittura 1. Scrivere in uscita il valore della variabile denominata "nome"
	Istruzione di assegnazione 1. Calcolare il valore della espressione aritmetica 2. Assegnare il risultato alla variabile denominata "nome"
	Istruzione di test 1. Calcolare il valore della espressione booleana 2. Se la risposta è vera, seguire il ramo sì, altrimenti seguire il ramo no
	Istruzione di stop 1. Fermarsi

Fig. 3 - Simboli del linguaggio dei grafi di flusso e loro significato

ingresso e quella dei dati di uscita (dall'algoritmo) avvenga tramite fogli di carta: sia in ingresso che in uscita i dati vengono scritti uno per ogni foglio. Stabiliamo anche, per semplicità, che l'algoritmo possa operare solo su dati costituiti da numeri interi. Assumiamo infine che accanto a dati con valore costante (ad esempio i numeri +5, -234), sia possibile fare riferimento a nomi di variabili. Quando formuliamo algoritmi risolutivi, le informazioni oggetto del calcolo sono spesso riferite tramite un nome, piuttosto che attraverso il loro specifico valore. Ad esempio, nella descrizione dell'algoritmo per la ricerca del cognome abbiamo usato i termini *elenco*, *cognome*, ad indicare che facciamo riferimento ad un generico elenco e cognome. Questo accade anche nei linguaggi programmativi: chiamiamo d'ora in poi *variabili* i nomi simbolici usati nell'algoritmo per denotare i dati. Accanto al nome, la variabile avrà associato un valore, che è appunto il valore del dato che la variabile rappresenta, e che può in generale variare nel corso della esecuzione dell'algoritmo. Per rappresentare materialmente tale valore si potrà utilizzare un foglio di carta, come per i dati di ingresso e di uscita.

Riportiamo nella fig. 3 i simboli usati nella notazione dei grafi di flusso per

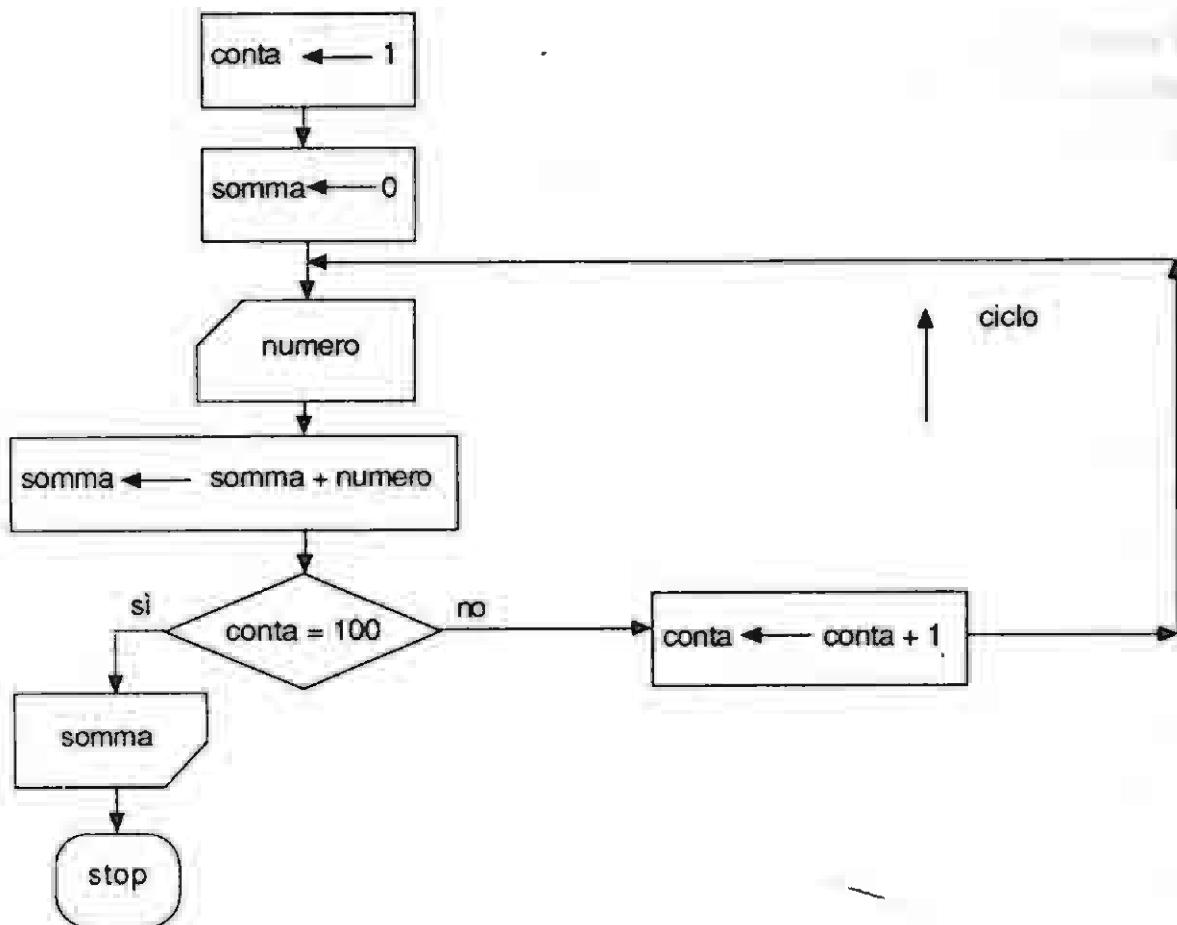


Fig. 4 - Un primo esempio di algoritmo

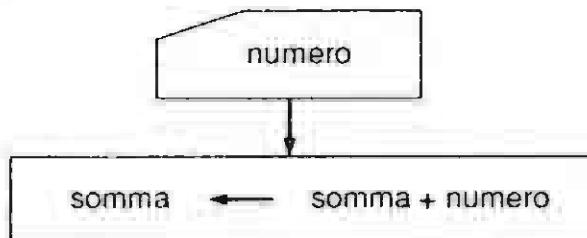
esprimere le istruzioni, e il significato di tali simboli, cioè l'insieme delle azioni svolte quando nel corso della esecuzione dell'algoritmo si incontra il simbolo corrispondente.

Le espressioni aritmetiche sono costituite da insiemi di operatori (somma, sottrazione, moltiplicazione e divisione) e da operandi, cioè costanti o variabili aventi valori interi. Nel calcolo di una espressione, alle variabili vanno sostituiti i rispettivi valori. Le espressioni booleane sono semplici espressioni relazionali, che mettono a confronto costanti o variabili (numeriche) per mezzo degli operatori minore, maggiore, uguale. Un esempio di algoritmo scritto nella notazione dei grafi di flusso è mostrato nella fig. 4.

Il nucleo centrale è costituito da un ciclo di istruzioni, messo in evidenza nella figura. Le prime due istruzioni eseguite assegnano rispettivamente i valori 1 e 0 alle variabili `conta` e `somma`. Successivamente, inizia l'esecuzione iterata delle istruzioni nel ciclo. L'iterazione termina quando il valore della espressione

`conta = 100`

è vero; questo avverrà alla centesima esecuzione della istruzione di test, poiché ad ogni esecuzione del ciclo il valore della variabile conta è incrementato di 1. Le due istruzioni:



vengono perciò eseguite 100 volte; ad ogni esecuzione si legge un nuovo valore nella variabile numero, che viene successivamente sommato alla variabile somma; vengono perciò letti 100 valori, sommati uno per volta alla variabile somma. Poichè all'inizio somma ha valore 0, l'esecuzione dell'algoritmo consiste nella lettura e somma di 100 numeri, e la stampa finale del risultato.

L'algoritmo può vedersi come una applicazione del principio di induzione (vedere Appendice). Supponiamo infatti di saper fare la somma di due numeri n_1 e n_2 , ed indichiamo tale operazione con $n_1 + n_2$. La somma di n numeri può allora essere espressa come segue:

- se i numeri sono due
la somma è $n_1 + n_2$
- se i numeri sono $n > 2$
la somma è la somma dei primi $n-1$ numeri più l'ultimo numero.

Non ci interessa proseguire ulteriormente nella utilizzazione dei grafi di flusso, che in algoritmi non elementari manifesta seri limiti di potenza espresiva.

4. UN LINGUAGGIO DI PROGRAMMAZIONE

In questo paragrafo introduciamo il linguaggio Pascal, presentandone un primo frammento. Come nel precedente paragrafo, assumiamo che i dati su cui i programmi operano siano numeri interi. Si veda in fig. 5 un esempio di programma scritto in Pascal: il programma legge 100 numeri e ne stampa la somma; esso è equivalente al programma di fig. 4.

Come si vede dall'esempio, il linguaggio Pascal risponde a regole per la composizione delle istruzioni più precise e formali di quelle descritte nella notazione dei grafi di flusso. Descriviamo ora alcune caratteristiche sintattiche e semantiche del linguaggio. Viene detta *sintassi* di un linguaggio l'insieme di regole che i programmi del linguaggio devono rispettare. Anche nel linguaggio

program esempio;

var somma, numero, conta: integer;

begin

somma := 0;

conta := 1;

while conta <= 100

do begin

readln (numero);

somma := somma + numero;

conta := conta + 1

end;

writeln (somma)

end.

intestazione
parte dichiarazione
variabili

Parte
istruzioni

Fig. 5 - Esempio di programma

naturale esistono tali regole, per cui siamo in grado di dire che la frase

il si è cane morso la coda

è sintatticamente scorretta nella lingua italiana, mentre la frase

il cane si è morso la coda

è sintatticamente corretta.

Viene detta *semantica* di un linguaggio programmatico il significato che si attribuisce alle istruzioni del linguaggio, espresso o in termini dell'insieme delle azioni che vengono eseguite dall'elaboratore per eseguire le istruzioni del linguaggio ovvero in termini di funzione calcolata. Nel cap. 1 torneremo ad analizzare più formalmente i concetti di sintassi e semantica di un linguaggio, e introdurremo diversi formalismi per la loro descrizione. In questo paragrafo, per la descrizione della sintassi e semantica adotteremo prevalentemente il linguaggio naturale.

Il linguaggio Pascal utilizza un vocabolario costituito dall'alfabeto inglese, dalle cifre, da un insieme di simboli speciali e da un insieme di parole chiave, cioè di parole che hanno un particolare ruolo nel linguaggio. Ogni programma

utilizza un insieme di identificatori che associano nomi univoci alle variabili utilizzate dal programma. I nomi di identificatori sono formati da stringhe (cioè sequenze) di caratteri alfanumerici (lettere o cifre) che devono iniziare con una lettera. Nel programma di fig. 5 compaiono gli identificatori *somma*, *numero*, *conta*, che rappresentano i nomi delle variabili.

La struttura di un programma (vedi ancora fig. 5) consiste in una intestazione, in cui si assegna un nome al programma, seguita da due parti:

1. una parte dichiarazioni, che ha lo scopo di dichiarare tutte le variabili utilizzate nel programma;
2. una parte istruzioni, che descrive l'algoritmo risolutivo utilizzato, mediante istruzioni del linguaggio.

Lo scopo della parte dichiarazione delle variabili è di definire tutte le variabili che il programma utilizza, con il loro tipo, cioè l'insieme dei valori che esse possono assumere e le operazioni che su di esse possono essere eseguite: questa è una novità rispetto alla notazione dei grafi di flusso, in cui si assume implicitamente che le variabili operino su numeri interi. Nel linguaggio Pascal i tipi dei dati devono essere dichiarati esplicitamente. I linguaggi programmatici mettono a disposizione del progettista vari meccanismi linguistici per la rappresentazione dei dati: come abbiamo accennato nei precedenti paragrafi, la scelta delle rappresentazioni per i dati coinvolti dal problema è parte importante del progetto dell'algoritmo. Torneremo ad analizzare questo aspetto nel cap. 3: in questo capitolo, come già detto, usiamo solo variabili intere.

Vediamo ora la sintassi della parte dichiarazione delle variabili: in essa compare la parola chiave **var**, seguita da tutti gli identificatori delle variabili che il programma utilizza. Viene infine dichiarato il tipo delle variabili (in questo caso il tipo intero), mediante la parola chiave **integer**.

La parte istruzioni è la parte del programma in cui viene descritto l'algoritmo. Anche in questo caso il linguaggio Pascal presenta una importante novità rispetto alla notazione dei grafi di flusso, che riguarda le modalità con cui è espresso nel linguaggio l'ordine di esecuzione delle istruzioni (detto anche *controllo*). Nella notazione dei grafi di flusso la sola istruzione condizionale permetteva di esprimere una ramificazione del controllo: nel linguaggio Pascal l'ordine di esecuzione è espresso per mezzo di un insieme di istruzioni, dette appunto istruzioni di controllo.

Dal punto di vista della sintassi, la parte istruzioni di un programma Pascal è costituita da una sequenza di istruzioni separate da punto e virgola e racchiusa tra le parole chiave **begin** e **end**; un punto specifica la fine del programma.

Le istruzioni che introdurremo nel seguito appartengono ai seguenti tipi:

1. istruzioni di lettura e scrittura, il cui scopo è di far comunicare dati tra il programma e l'ambiente esterno;
2. istruzione di assegnazione, il cui scopo è di modificare i valori delle variabili con nuovi valori risultato di calcoli;
3. istruzioni di controllo. Esse sono:
 - la istruzione composta, che permette di costruire a partire da un insieme di istruzioni una nuova istruzione;
 - la istruzione condizionale, che permette di ramificare il controllo del programma a seconda del valore di una espressione booleana;
 - la istruzione di iterazione, che dà luogo alla esecuzione ripetuta di una istruzione (eventualmente composta) fin tanto che una data espressione booleana rimane vera.

All'interno di un programma è anche possibile scrivere frasi di commento con cui si può descrivere in linguaggio naturale il ruolo svolto dalle varie istruzioni di un programma. Le frasi di commento devono essere racchiuse tra una coppia di parentesi graffe. Descriviamo ora le istruzioni.

Istruzioni di lettura e scrittura

L'istruzione per la lettura dei dati ha il seguente formato:

readln (<identificatore>)

Corrisponde alla istruzione di lettura del linguaggio dei grafi di flusso, ed ha lo stesso significato.

In questa introduzione, per descrivere in modo preciso la struttura delle istruzioni sarà adottata la convenzione di racchiudere tra parentesi acute nomi (eventualmente composti) che indicano un generico elemento di una classe cui appartengono. Così, con

<espressione aritmetica>

si intenderà appunto una qualunque espressione aritmetica. I simboli che non sono racchiusi tra parentesi descrivono esattamente il simbolo da scrivere.

La istruzione di scrittura ha il seguente formato:

writeln (<identificatore>)

ed è analoga alla istruzione di scrittura del linguaggio dei grafi di flusso.

Istruzione di assegnazione

Il formato dell'istruzione di assegnazione è:

<identificatore> := <espressione aritmetica>

Il significato dell'istruzione di assegnazione è il seguente:

1. viene calcolato il valore della espressione aritmetica che segue il simbolo `:=` (detto simbolo di assegnazione);
2. tale valore viene assegnato alla variabile il cui identificatore si trova prima del simbolo di assegnazione.

Istruzione composta

L'istruzione composta ha la seguente forma:

begin
<lista di istruzioni>
end

dove *<lista di istruzioni>* è una sequenza di istruzioni separate da punto e virgola. L'istruzione composta permette di trattare un insieme di istruzioni come una nuova istruzione. L'esecuzione dell'istruzione comporta l'esecuzione delle istruzioni che la compongono nell'ordine in cui sono scritte.

Istruzione condizionale

Il formato dell'istruzione è:

if *<espressione booleana>*
 then *<istruzione>*
 else *<istruzione>*

Il significato della istruzione condizionale è il seguente:

1. viene calcolato il valore di verità (cioè vero o falso) della espressione booleana che segue la parola `if`;
2. se tale valore di verità è il valore vero, viene eseguita la istruzione compresa fra la parola chiave `then` e la parola chiave `else` e quindi si passa alla esecuzione della istruzione successiva all'istruzione condizionale; non viene quindi eseguita l'istruzione che immediatamente segue la parola

chiave **else**. Se invece il valore di verità è il valore falso, allora viene eseguita l'istruzione che segue la parola chiave **else** e quindi si passa ad eseguire l'istruzione successiva all'istruzione condizionale. In questo caso non viene eseguita l'istruzione che segue la parola chiave **then**.

Si noti che dopo le parole chiave **then** ed **else** deve necessariamente comparire una sola istruzione. Questo significa che se si ha necessità di usare più istruzioni, è necessario utilizzare l'istruzione composta.

Della istruzione condizionale esiste un'variante semplificata il cui formato è:

```
if <espressione booleana>
    then <istruzione>
```

In questo caso viene dapprima calcolato il valore dell'espressione booleana: se tale valore di verità è vero allora viene eseguita l'istruzione che segue la parola chiave **then**, altrimenti tale istruzione non viene eseguita. In entrambi i casi si passa ad eseguire l'istruzione immediatamente successiva all'istruzione condizionale.

Istruzione di iterazione

Il formato della istruzione è il seguente:

```
while <espressione booleana>
    do <istruzione>
```

L'istruzione che segue la parola chiave **do** può essere una qualunque istruzione, ed è detta corpo del ciclo. Il significato dell'istruzione è il seguente:

1. viene valutata l'espressione booleana che compare dopo la parola chiave **while**;
2. se l'espressione booleana assume il valore vero viene eseguita l'istruzione che compare dopo la parola chiave **do** e quindi si torna ai passo 1; se, invece, l'espressione assume il valore falso l'esecuzione dell'istruzione termina e si passa ad eseguire l'istruzione successiva.

Prima di concludere il paragrafo mostriamo un semplice esempio di programma. Si consideri il problema:

Leggere due numeri e stampare il maggiore

Un programma che risolve il problema è mostrato in fig. 6.

Le due istruzioni di lettura permettono di leggere i valori interi e assegnarli alle variabili *primo* e *secondo*. Successivamente, la istruzione condizionale

confronta i due numeri, e a seconda dell'esito del confronto, stampa l'uno o l'altro.

```
program maggiore;  
{Legge due numeri e stampa il maggiore}  
var primo, secondo: integer;  
  
begin  
  readln(primo);  
  readln(secondo);  
  if primo > secondo then writeln(primo)  
    else writeln(secondo)  
end.
```

Fig. 6 - Programma per la scelta del maggiore tra due numeri

5. IL PROGETTO DI PROGRAMMI

Esaminiamo ora alcuni esempi di programmi, di complessità crescente. Gli esempi hanno lo scopo di mettere in evidenza diversi aspetti del progetto di programmi, che saranno oggetto di un successivo approfondimento.

Problema 1. Leggere 100 numeri, calcolare la somma e stamparla.

Il problema è già stato esaminato nel precedente paragrafo. Il programma è mostrato nuovamente in fig. 7 con un insieme di commenti che hanno lo scopo di chiarire la logica dell'algoritmo.

Alla somma parziale dei numeri (*somma*) viene assegnato il valore iniziale 0, ed al contatore dei numeri da leggere (*conta*) viene assegnato il valore iniziale 1. Vengono poi letti 100 numeri, ciascuno memorizzato volta a volta nella variabile *numero*, e successivamente sommato alla variabile *somma*. Alla fine viene stampata la somma.

Quando si progetta un programma, è fondamentale provare che il programma rispetta correttamente le specifiche del problema. Accade spesso che le versioni iniziali del programma manifestino errori di varia natura, e l'attività necessaria per eliminarli può essere una parte rilevante dell'intero progetto. Mostriamo a titolo di esempio varie possibili versioni scorrette del programma precedente, analizzandone il comportamento. Il programma di fig. 8.a è errato, poiché la variabile *somma* viene azzerata ad ogni esecuzione del ciclo. L'esecuzione del

```
program sommanumeri;
{Legge 100 interi e li somma}
{La variabile numero memorizza il generico numero letto}
{La variabile somma rappresenta la somma parziale dei numeri letti}
{La variabile conta è l'indice del ciclo di lettura e somma}
var numero, somma, conta: integer;

begin
{Assegna i valori iniziali alle variabili somma e conta}
somma := 0;
conta := 1;
{Legge e somma i 100 numeri}
while conta <= 100
do begin
    readln (numero);
    somma := somma + numero;
    conta := conta + 1;
end;
{Stampa il risultato}
writeln (somma)
end.
```

Fig. 7 - Programma per il problema 1

programma fornisce come risultato l'ultimo numero letto. Il programma di fig. 8.b non termina mai, perché ci si è scordati di incrementare la variabile *conta*, e dunque non verrà mai verificata la condizione di uscita dal ciclo. Infine il programma di fig. 8.c legge e somma solo 50 numeri, perché la variabile *conta* è scorrettamente incrementata di 2.

I metodi per provare la correttezza di un programma si ispirano a due diverse strategie. I primi eseguono il programma su dati di test, con lo scopo di verificare se il suo comportamento è quello richiesto dalle specifiche del problema. La scelta dei dati di test è parte essenziale del metodo. Altri metodi dimostrano la correttezza per mezzo di una prova formale eseguita a partire da una analisi statica del testo del programma, senza eseguirlo. I metodi per provare la correttezza di un programma sono descritti nel cap. 5.

Prima versione

```
begin
  conta:= 1;
  while conta <= 100
    do begin
      somma := 0;
      readln (numero);
      somma := somma + numero;
      conta := conta + 1;
    end;
    writeln (somma)
  end.
```

(a)

Seconda versione

```
begin
  somma := 0;
  conta:= 1;
  while conta <= 100
    do begin
      readln (numero);
      somma := somma + numero;
    end;
    writeln (somma)
  end.
```

(b)

Terza versione

```
begin
  somma := 0;
  conta:= 1;
  while conta <= 100
    do begin
      readln (numero);
      somma := somma + numero;
      conta := conta + 2;
    end;
    writeln (somma)
  end.
```

(c)

Fig. 8 - Versioni scorrette del programma per il problema 1

{Dato un intero positivo n, stampa la somma dei primi n numeri}
 {La variabile n memorizza il numero dei valori da sommare}
 {La variabile somma rappresenta la somma parziale degli N numeri}
 {La variabile conta è l'indice del ciclo di lettura e somma}
program somma_primi_numeri;
var n, somma, conta: integer;

begin
readln (n);
 somma := 0;
 conta := 1;
while conta <= n
do begin
 somma := somma + conta;
 conta := conta + 1;
end;
writeln (somma)
end.

Fig. 9 - Programma per il problema 2

Problema 2. Dato un numero positivo n, calcolare e stampare la somma dei primi n numeri interi.

In questo caso l'unico dato di ingresso è il valore di n, poichè, noto n, si conoscono tutti i dati su cui operare, costituiti appunto dalla sequenza di valori 1,..., n. Una prima soluzione al problema consiste nel modificare l'algoritmo risolutivo del problema 1, generando i numeri da sommare direttamente all'interno dell'algoritmo, senza leggerli dall'esterno. I numeri da generare coincidono con i valori assunti via via dalla variabile *conta*. In base alle precedenti osservazioni possiamo costruire l'algoritmo di fig. 9.

Per il precedente problema esiste un secondo algoritmo, che ottiene il risultato desiderato con un numero di operazioni molto inferiore. Basterà applicare la formula che esprime in forma chiusa la somma dei primi n numeri interi:

$$\sum_{i=1}^n = n(n+1)/2$$

Vogliamo confrontare i due algoritmi dal punto di vista della loro efficienza. Intendiamo per efficienza la capacità dell'algoritmo di risolvere il problema

```

program minima_eta;
{Legge n numeri interi, che rappresentano le età di un gruppo di persone e
stampa la età minima}

{La variabile n rappresenta il numero di persone}
{La variabile minimo memorizza il minimo provvisorio}
{La variabile numero memorizza il generico numero della sequenza}
{La variabile i conta il numero di valori della sequenza letti}

var n, minimo, numero, i: integer;
begin

{Legge dapprima il numero di persone}
readln (n);
{Il primo numero letto è scelto come minimo provvisorio}
readln (numero);
minimo := numero;
{Vengono letti i restanti numeri; ciascuno di essi è confrontato con il
minimo provvisorio. Se necessario, questo viene aggiornato}
i := 1;
while i <= n - 1
do begin
    readln (numero);
    if numero < minimo
        then
            {Si è trovato un nuovo minimo}
            minimo := numero;
            {Si passa al numero successivo}
            i := i + 1
        end;
    {Stampa del risultato}
    writeln (minimo)
end.

```

Fig. 10 - Programma per il problema 3

con limitato uso di risorse di calcolo. Come possiamo misurare l'efficienza di un programma? Per il momento possiamo misurarla in termini del numero di istruzioni eseguite; non ci interessa tanto contare le istruzioni in assoluto, quanto in rapporto al numero di dati da sommare (cioè in funzione di n): questo ci fornirà una misura più interessante, in quanto appunto messa in rapporto con la quantità di dati da elaborare.

Contando le istruzioni eseguite, è facile vedere che nel caso del primo programma vengono eseguite $2n + 3$ istruzioni, mentre nel secondo caso ne vengono eseguite 3. Dunque, nel primo programma il numero di istruzioni cresce con lo stesso ordine di grandezza del numero dei dati da sommare, mentre nel secondo è costante, e indipendente da tale numero. Possiamo perciò affermare che il secondo programma è più efficiente del primo.

Nell'esempio precedente abbiamo semplificato molto la analisi di efficienza; ad esempio, abbiamo dato lo stesso peso ad istruzioni che possono richiedere tempi di esecuzione anche molto diversi. Nel cap. 6 renderemo più formale la trattazione, e vedremo quali fattori possono influenzare l'analisi di efficienza, e come possiamo tenerne conto nel progetto di un programma.

Problema 3. Note le età di un gruppo di persone, trovare l'età del più giovane.

Poiché le età possono essere rappresentate per mezzo di valori interi, il problema consiste nel leggere un insieme di valori interi e trovare il minimo. Possiamo adottare un metodo induttivo di soluzione. Supponiamo in questo caso di saper calcolare il minimo di due numeri, e indichiamo tale operazione con minimo (n_1, n_2). Il minimo di n numeri può allora esprimersi come:

se i numeri sono due

il minimo è minimo (n_1, n_2)

altrimenti il minimo è

minimo (primo numero, minimo dei numeri dal secondo all'ultimo)

Applicando anche in questo caso la istruzione di iterazione, otteniamo il programma di fig. 10.

Nel programma di fig. 10 abbiamo utilizzato commenti per esprimere il ruolo delle varie istruzioni nella strategia risolutiva. Senza i commenti il programma risulterebbe meno comprensibile. La capacità di comprendere in ogni momento dalla lettura del programma le soluzioni adottate è un requisito di grande importanza, tanto maggiore quanto nel tempo il programma sarà sottoposto a modifiche ed aggiornamenti. Le metodologie per una buona documentazione saranno oggetto di studio nel cap. 4

Problema 4. Scrivere un programma che legge tre numeri, che rappresentano le lunghezze dei lati di un triangolo, e stampa il messaggio:

equilatero
isoscele
scaleno

a seconda che il triangolo sia appunto equilatero, isoscele oppure scaleno.

Il progetto di un algoritmo risolutivo per questo problema si presenta non immediato. Procediamo al progetto per raffinamenti successivi, cercando di frammentarlo in sottoproblemi più semplici, adottando il metodo già introdotto alla fine del par. 2. Utilizziamo il linguaggio naturale per esprimere inizialmente l'algoritmo. Una prima versione di algoritmo risolutivo compare in fig. 11.a.

Partendo da questa prima versione dell'algoritmo possiamo già fare delle scelte per quanto riguarda le variabili richieste nel programma: abbiamo la necessità di assegnare le tre lunghezze dei lati a tre variabili, per poterle poi confrontare nel corso del programma; chiamiamo le tre variabili *primo*, *secondo*, *terzo*. Per trasformare la versione di fig. 11.a in un programma Pascal dobbiamo a questo punto raffinare la frase che esprime il confronto in termini di attività più elementari. Possiamo iniziare con i primi due lati; a seconda dell'esito del confronto, possiamo poi procedere con il primo ed il terzo lato, e così via finché non siamo in grado di prendere una decisione. Otteniamo così il programma di fig. 11.b.

Lo stesso problema può essere risolto con altri algoritmi. Un nuovo algoritmo è descritto ad un primo livello di dettaglio in fig. 12.a. In questo caso la strategia consiste nell'effettuare inizialmente una statistica su quante sono le coppie di lati uguali: a seconda del risultato si stamperà l'opportuno messaggio. Possiamo ora raffinare l'algoritmo in una nuova versione. Per quanto riguarda le variabili, in questo caso abbiamo bisogno di una nuova variabile, *uguali*, che indica il numero di coppie di lati uguali. Riguardo all'algoritmo, un aspetto che non è ancora espresso nella precedente versione riguarda l'ordine con cui confrontare le coppie di lati. Possiamo decidere un ordine a caso, dando luogo così alla versione finale di fig. 12.b.

Questi esempi ci confermano che nel progetto di un programma possiamo procedere per successivi raffinamenti, dapprima diamo una descrizione dell'algoritmo espressa in termini di attività complesse, non immediatamente descrivibili per mezzo di istruzioni del linguaggio programmatico, e, successivamente, frammentiamo tali attività in termini di passi più elementari, fin quando non siamo in grado di esprimere l'algoritmo mediante istruzioni del

-
- Leggi i tre lati
 - Confronta i lati a coppie, fin quando non hai raccolto un numero di informazioni sufficienti a prendere la decisione
 - Stampa il risultato

(a)

```
program triangolo;  
{Legge tre numeri, che rappresentano le lunghezze dei lati di  
un triangolo. Stampa il messaggio 'equilatero', 'isoscele', o 'scaleno'  
a seconda che i tre numeri rappresentino le lunghezze di un  
triangolo equilatero, isoscele o scaleno}  
{primo, secondo e terzo rappresentano le lunghezze dei lati}  
var primo, secondo, terzo: integer;  
  
begin  
  readln (primo); readln (secondo); readln (terzo);  
  
  if primo = secondo  
  then  
    if secondo = terzo  
    then  
      writeln ('equilatero')  
    else  
      writeln ('isoscele')  
  else  
    if secondo = terzo  
    then  
      writeln ('isoscele')  
    else  
      if primo = terzo  
      then  
        writeln ('isoscele')  
      else  
        writeln ('scaleno')  
end.
```

(b)

Fig. 11 - Primo programma per il problema 4

-
- Leggi i tre lati
 - Confronta i lati a coppie, contando quante coppie di lati sono uguali
 - Se le coppie uguali sono due, il triangolo è equilatero
 - altrimenti se sono una il triangolo è isoscele
altrimenti il triangolo è scaleno.

(a)

program triangolo;

{Legge tre numeri, che rappresentano le lunghezze dei lati di un triangolo. Stampa il messaggio 'equilatero', 'isoscele' o 'scaleno' a seconda che i tre numeri rappresentino le lunghezze di un triangolo equilatero, isoscele o scaleno }

{primo, secondo e terzo rappresentano le lunghezze dei lati}

{uguali rappresenta il numero di coppie di lati uguali}

var primo, secondo, terzo, uguali: integer;

begin

readln (primo); readln (secondo); readln (terzo);

uguali := 0;

if primo = secondo **then** uguali := uguali + 1;

if primo = terzo **then** uguali := uguali + 1;

if secondo = terzo **then** uguali := uguali + 1;

if uguali = 0

then writeln ('scaleno')

else if uguali = 1

then writeln ('isoscele')

else writeln ('equilatero')

end.

(b)

Fig 12 - Secondo programma per il problema 4

linguaggio. Procedendo per raffinamenti, abbiamo visto che nei diversi livelli sono stati esplicitati successivi dettagli riguardo alle variabili utilizzate dall'algoritmo e la sua struttura. Nel cap. 4 studieremo un insieme di metodologie che aiutano il progettista nel processo di scelta dei tipi di dati e della struttura dell'algoritmo.

Riassumendo quanto finora esposto, i precedenti esempi di programmi hanno mostrato che individuare algoritmi ed esprimerli in un linguaggio di programmazione richiede in genere una sofisticata attività di progetto. Aspetti fondamentali di tale attività sono:

1. una precisa conoscenza della sintassi e della semantica del linguaggio di programmazione a disposizione;
2. un insieme di strategie per individuare uno (o più) algoritmi risolutivi del problema; aspetti essenziali del progetto sono la scelta delle strutture con cui rappresentare i dati coinvolti dal problema e la scelta della struttura delle istruzioni;
3. un insieme di strategie per garantire che il programma individuato rispetti un certo insieme di qualità, quali la correttezza, la efficienza, la comprensibilità.

Come si è detto, la definizione precisa di tali aspetti è lo scopo principale degli argomenti trattati nei prossimi capitoli.

6. STRUTTURA DI UN ELABORATORE

Un elaboratore è organizzato in un insieme di risorse tra loro coordinate. In questo paragrafo introduciamo brevemente lo schema funzionale di un elaboratore, che verrà ripreso in modo più approfondito nei capp. 8 e 9. La fig. 13 sintetizza la struttura di un elaboratore indicandone le diverse funzioni.

Per eseguire automaticamente un procedimento di calcolo, occorre comunicare all'elaboratore tramite gli organi di ingresso (o dispositivi di *input*) l'algoritmo risolutivo, espresso tramite un programma, e i dati su cui il programma opera. L'elaboratore deve a sua volta essere in grado di comunicare, tramite gli organi di uscita (o dispositivi di *output*), i risultati dei calcoli. Esempi di organi di ingresso sono la tastiera, e il lettore ottico. Esempi di organi di uscita sono la stampante, ed il video. Accanto agli organi di ingresso e di uscita, un elaboratore dispone di altri tre tipi di risorse:

- la memoria, in cui sono rappresentate le istruzioni del programma, i dati di ingresso, i risultati parziali del calcolo;
- un processore, la unità dove si svolgono i calcoli, suddiviso a sua volta in:
 - a. una unità logico-aritmetica, in cui vengono effettuati i calcoli logici e aritmetici connessi alla esecuzione del programma;
 - b. un organo di governo, che ha lo scopo di interpretare le istruzioni del programma e attivare le risorse coinvolte dalla esecuzione delle istruzioni.

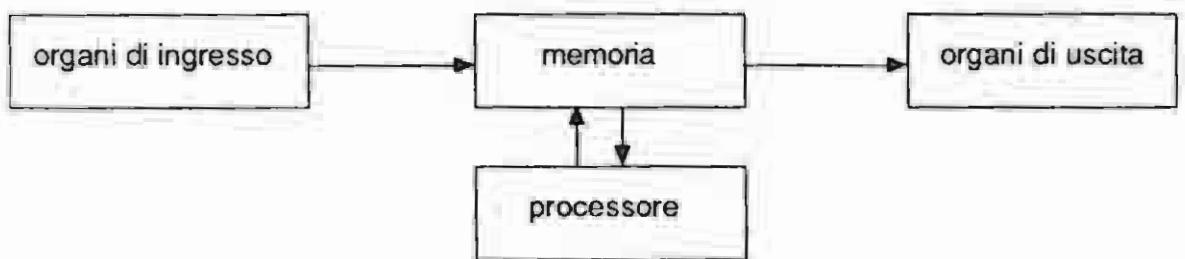


Fig. 13 - Schema funzionale di un elaboratore

Istruzioni e dati sono rappresentati nella memoria e negli altri organi per mezzo di codici binari, cioè simboli appartenenti ad un alfabeto di due soli simboli (nel seguito 0 e 1). La memoria è organizzata in un insieme di *celle*, ognuna delle quali può rappresentare un insieme di simboli binari; ogni cella è univocamente identificata da un indirizzo.

Mediante i circuiti dell'organo di governo, un elaboratore è in grado di interpretare istruzioni che realizzano azioni in genere molto semplici: l'insieme di tali istruzioni è detto linguaggio macchina dell'elaboratore. Esempi di istruzioni in un ipotetico linguaggio macchina sono le seguenti:

1. "Leggi un dato dall'organo di ingresso e memorizzalo nella cella di indirizzo 1000";
2. "Somma il contenuto della cella di indirizzo 1000 al contenuto della cella di indirizzo 1001 e metti il risultato nella cella di indirizzo 1010";
3. "Vai ad eseguire la istruzione che si trova nella cella di indirizzo 1110";
4. "Stampa il contenuto della cella di indirizzo 1010".

Come si può capire da questi esempi, scrivere programmi in linguaggio macchina è complicato, poiché ogni istruzione svolge una operazione molto elementare e descritta con codici vicini alla macchina e lontani dal modo tipicamente simbolico con cui l'essere umano è abituato a organizzare i suoi ragionamenti. Nasce allora l'idea di progettare nuovi linguaggi più potenti e più vicini al linguaggio con cui ci si esprime normalmente: questi linguaggi sono comunemente detti *linguaggi ad alto livello*: un esempio di linguaggio ad alto livello è quello introdotto nel par. 4. Le istruzioni dei linguaggi ad alto livello non possono essere direttamente interpretate dai circuiti degli organi di governo: occorre perciò che i programmi scritti nel linguaggio ad alto livello siano tradotti nelle equivalenti versioni in linguaggio macchina. A tale scopo sono utilizzati programmi detti traduttori, che possono essere di due tipi, *compilatori* ed *interpreti*.

I compilatori effettuano la traduzione dell'intero programma in linguaggio macchina. Per fare ciò il compilatore deve conoscere la sintassi e la semantica

del linguaggio ad alto livello e del linguaggio macchina. Il compilatore effettua il processo di traduzione controllando inizialmente la correttezza sintattica del programma e producendo successivamente la corrispondente versione in linguaggio macchina.

Perciò, tutte le volte che intendiamo eseguire un programma scritto in un linguaggio ad alto livello, se abbiamo a disposizione un compilatore occorrerà procedere in due fasi (vedi fig. 14).

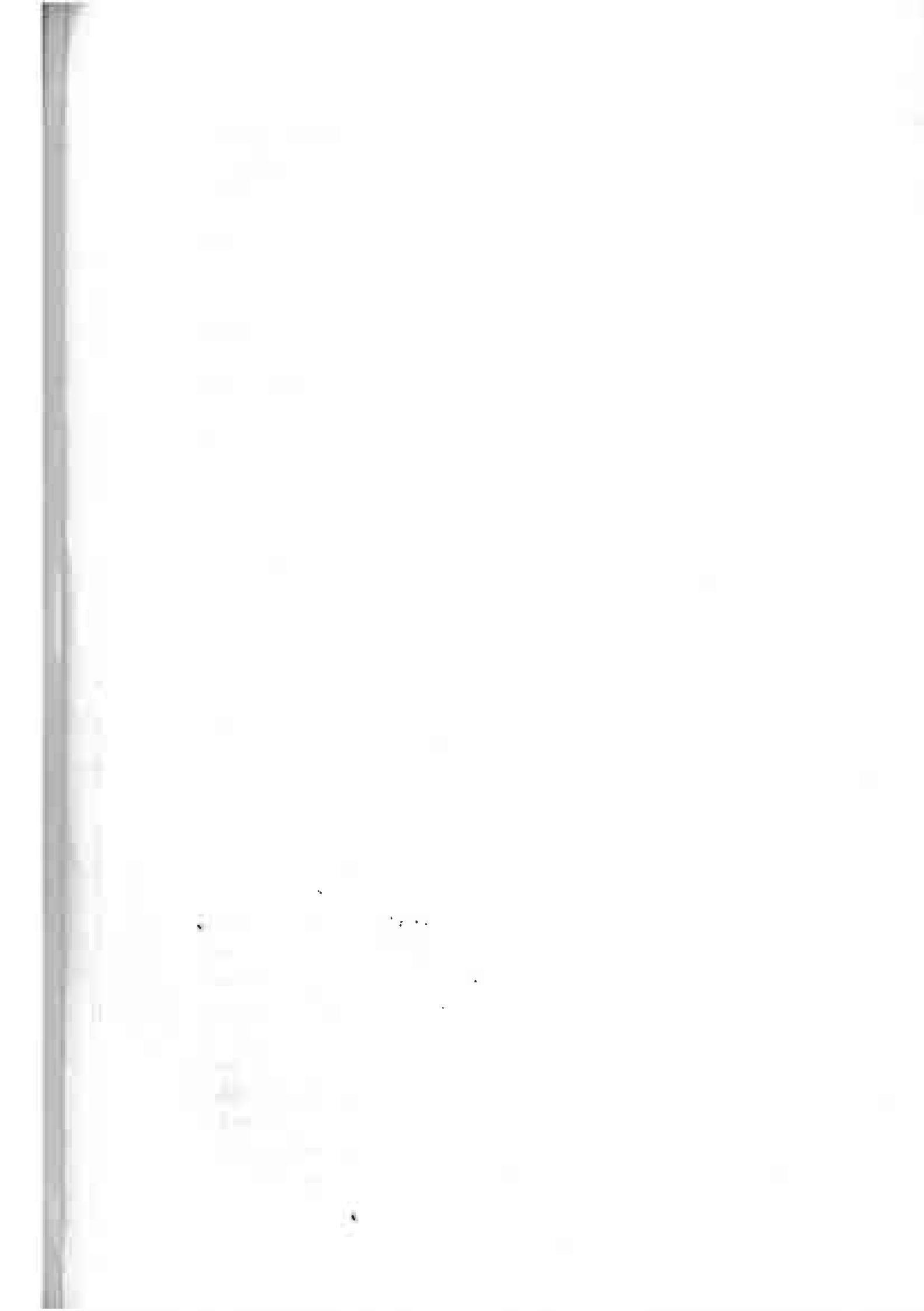
- dapprima il programma scritto dall'utente è tradotto nel corrispondente programma in linguaggio macchina dal programma traduttore;
- successivamente il programma in linguaggio macchina viene eseguito sui dati per cui è richiesta la elaborazione.

Un secondo approccio al processo di esecuzione di programmi è quello che prevede l'utilizzo di programmi interpreti, che procedono alla traduzione del programma istruzione per istruzione: ogni nuova istruzione viene interpretata ed eseguita.

La realizzazione di linguaggi sempre più ricchi di meccanismi linguistici estende le possibilità di utilizzo degli elaboratori per la soluzione automatica dei problemi: un elaboratore dotato di un compilatore per un linguaggio ad alto livello è in grado di eseguire istruzioni più potenti (e utilizzare rappresentazioni di dati più ricche) di un elaboratore che possa solo eseguire istruzioni in linguaggio macchina. Un secondo meccanismo a disposizione del programmatore nei linguaggi ad alto livello è la possibilità di estendere il linguaggio dal suo interno con nuove istruzioni: vedremo nel cap. 2 come questa possibilità sia data dall'uso delle procedure e funzioni.



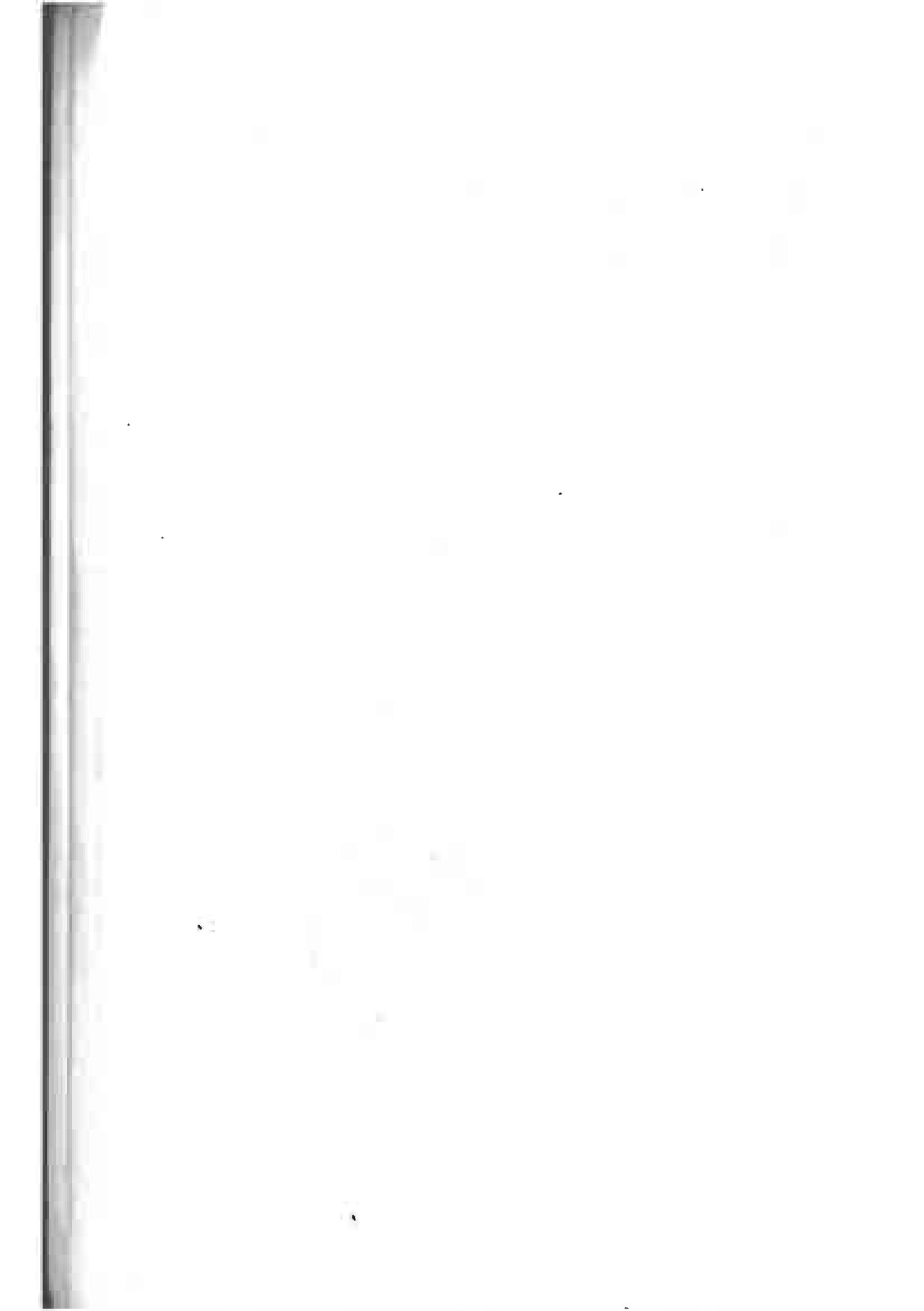
Fig. 14 - Fasi del processo di traduzione di un programma



Parte I

Linguaggi di programmazione

- 1. Linguaggi di programmazione: sintassi e semantica**
- 2. Linguaggi di programmazione: dati e controllo**



1.

Linguaggi di programmazione: sintassi e semantica

Come già detto nell'introduzione, i compiti che un elaboratore deve espletare vengono specificati attraverso programmi: questi rappresentano algoritmi che l'elaboratore esegue su dati forniti in ingresso, producendo risultati in uscita. I programmi vengono espressi in linguaggi artificiali, detti linguaggi di programmazione o linguaggi programmativi.

Allo scopo di poter definire l'insieme dei programmi legali di un linguaggio di programmazione ed il loro significato è necessario fornirne una descrizione precisa e non ambigua. Si parla quindi di definizione dei linguaggi di programmazione.

Nello studio delle lingue naturali si distinguono due componenti fondamentali: sintassi e semantica. La sintassi si occupa della forma delle frasi, cioè delle regole che vanno rispettate nella loro costruzione; la semantica si occupa del significato delle frasi. Analogamente, in un linguaggio di programmazione, la definizione della sintassi riguarda le regole che vanno rispettate per costruire tutti e soli i programmi legali nel linguaggio; la definizione della semantica invece riguarda le regole che governano l'attribuzione di un significato ai programmi del linguaggio. Si parla infine di implementazione di un linguaggio per intendere la definizione di una macchina che sia in grado di eseguire i programmi di quel linguaggio; più spesso per implementazione di un linguaggio si intende semplicemente la realizzazione di un traduttore che ne renda i programmi eseguibili su un dato elaboratore.

Nel resto del capitolo vengono presentate le nozioni fondamentali relative alla definizione sintattica dei linguaggi di programmazione (par. 1) ed i metodi principali per definirne la semantica (par. 2). Alcuni aspetti legati alla implementazione dei linguaggi di programmazione vengono invece affrontati nei capp. 2 e 9.

1. SINTASSI

In questo paragrafo ci occupiamo della definizione della sintassi di un linguaggio. Parliamo dapprima di linguaggi e grammatiche in generale, e poi approfondiremo gli aspetti relativi alla sintassi dei linguaggi di programmazione.

1.1. Linguaggi e grammatiche

Un linguaggio di programmazione è un linguaggio artificiale; iniziamo perciò questo paragrafo introducendo in maniera rigorosa la nozione di linguaggio artificiale e descrivendo gli strumenti necessari alla sua definizione.

Definizione. Dato un insieme finito non vuoto V , si definisce *universo linguistico* su V (e si indica con V^*) l'insieme delle sequenze finite di lunghezza arbitraria di elementi di V .

L'insieme V viene di solito chiamato *alfabeto*, oppure *vocabolario* o *lessico*. Gli elementi di V sono chiamati simboli terminali. Si noti che talvolta i simboli di V possono essere più complessi di una singola lettera dell'alfabeto della lingua italiana; per esempio 'if', 'then', 'begin', ecc. sono simboli dell'alfabeto del Pascal.

Gli elementi di V^* vengono detti *stringhe*, o *frasi* costruite su V .

Definizione. Un *linguaggio* L sull'alfabeto V è un sottoinsieme di V^* .

Sebbene V sia finito, V^* non lo è; esso è numerabile, ed i sottoinsiemi di V^* sono in quantità non numerabile (vedere il par. 1.4 dell'appendice). Nel considerare i linguaggi di programmazione, non siamo interessati a tutti i sottoinsiemi di V^* , ma solo a quelli che sono descrivibili in maniera finita. Questa 'descrizione può essere per esempio fornita attraverso una grammatica, nel modo che verrà precisato dalle seguenti definizioni.

Definizione. Una *grammatica* o *sintassi* G è definita da:

- V , un alfabeto di *simboli terminali*;
- N , un alfabeto di *simboli non terminali* (detti anche *categorie sintattiche*);
- $S \in N$, detto *assioma*, o *simbolo iniziale*, o anche *simbolo distinto*, tale che $V \cap N = \emptyset$.
- P , un insieme finito di regole sintattiche (o *produzioni*) del tipo $X \rightarrow \alpha$, dove $X \in N$ ed $\alpha \in (N \cup V)^*$;

Le produzioni sono talora scritte nella forma $X ::= \alpha$ invece che $X \rightarrow \alpha$.

Se in una grammatica esistono più regole aventi la stessa parte sinistra, ad esempio $X \rightarrow \alpha_1, X \rightarrow \alpha_2, \dots, X \rightarrow \alpha_n$, esse sono raggruppate, usando la convenzione notazionale:

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

e in tal caso si dice che $\alpha_1, \alpha_2, \dots, \alpha_n$ sono parti destre alternative per X .

Gli alfabeti terminale e non terminale sono disgiunti. Per distinguere i simboli di questi due alfabeti spesso si usa una delle due seguenti convenzioni: nella prima i simboli non terminali sono distinti dai terminali perché racchiusi tra parentesi angolate (es. `<frase>`, `<cifra>`); nella seconda i non terminali sono scritti in corsivo ed i terminali tra apici.

Definizione. Data una grammatica G e due stringhe $\beta, \gamma \in (N \cup V)^*$, si dice che γ deriva direttamente da β in G e si scrive $\beta \rightarrow \gamma$ se le stringhe si possono decomporre come $\beta = \eta A \delta$, $\gamma = \eta \alpha \delta$ con $A \in N$; $\alpha, \eta, \delta \in (N \cup V)^*$, ed esiste la produzione $A \rightarrow \alpha \in P$.

Si noti che le stringhe α , η , e δ nella definizione precedente possono anche essere vuote.

In modo semplice si può definire una catena di derivazioni dirette:

$$\beta_0 \rightarrow \beta_1 \rightarrow \beta_2 \dots \rightarrow \beta_n, \text{ o anche } \beta_0 \xrightarrow{n} \beta_n$$

Definizione. Data una grammatica G e due stringhe $\beta, \gamma \in (N \cup V)^*$, si dice che γ deriva da β e si scrive $\beta \rightarrow^* \gamma$, se esiste un $n \geq 0$ tale che $\beta \xrightarrow{n} \beta_n$ e $\beta_n = \gamma$.

Definizione. Data una grammatica G , dicesi *linguaggio generato* da G , e si indica con L_G l'insieme delle frasi di V^* derivabili a partire dall'assioma S .

Definizione. Un *linguaggio di programmazione* L su un alfabeto V è un sottoinsieme di V^* per cui esiste una grammatica G , tale che $L = L_G$.

Le stringhe o frasi di un linguaggio di programmazione vengono dette *programmi* (di tale linguaggio).

Si noti che, con la precedente definizione abbiamo dato una caratterizzazione di un linguaggio di programmazione solo in termini della sua sintassi, cioè delle regole che soprintendono alla costruzione dei suoi programmi. Perché la definizione di un linguaggio di programmazione sia completa dobbiamo anche

dare ai programmi una caratterizzazione semantica. Di questo parleremo nel par. 2.

Il formalismo appena introdotto per descrivere la grammatica di un linguaggio di programmazione è un metalinguaggio formale che prende il nome di BNF (Backus-Naur-Form, forma di Backus e Naur, dai nomi dei due studiosi che per primi l'hanno introdotta negli anni '50).

Un metalinguaggio è un linguaggio usato per parlare di un altro linguaggio. Per esempio, se diciamo "l'articolo determinativo in inglese è 'the'", od anche "il pronomine personale di terza persona singolare è 'he', oppure 'she' oppure 'it'", stiamo usando l'italiano come metalinguaggio per descrivere l'inglese.

Innanzitutto osserviamo che nel metalinguaggio BNF le produzioni si scrivono di solito col simbolo ::= . Il formalismo BNF viene spesso usato non nella forma originale, ma utilizzando alcune estensioni che permettono una scrittura più concisa delle grammatiche; si parla in questi casi di EBNF (Extended BNF). Illustriamo qui di seguito le estensioni più comuni.

Se nella parte destra di una produzione un simbolo (o sequenza di simboli, o alternativa di simboli) è racchiuso tra parentesi quadre, questo significa che esso è opzionale, che cioè può comparire zero oppure una volta, per esempio:

$$X \rightarrow [\alpha] \beta \text{ equivale a } X \rightarrow \beta \mid \alpha \beta.$$

Se invece esso è racchiuso tra parentesi graffe, con un numero intero ad apice, questo significa zero, una o più occorrenze del simbolo stesso, fino ad un massimo di n; per esempio:

$$X \rightarrow \{\alpha\}^n \beta$$

significa che da X si può derivare:

$$\beta, \quad \alpha\beta, \quad \alpha\alpha\beta, \quad \alpha\alpha\alpha\beta, \quad \dots$$

con un massimo di n occorrenze di α.

Se, per esempio, n=3, questo equivale ad avere nella grammatica le produzioni:

$$X \rightarrow \beta \mid \alpha\beta \mid \alpha\alpha\beta \mid \alpha\alpha\alpha\beta.$$

Se invece un simbolo α è racchiuso tra parentesi graffe (senza apice), questo significa zero, una o più (in numero finito, ma arbitrario) occorrenze del simbolo stesso; per esempio:

$$X \rightarrow \{\alpha\}\beta$$

significa che da X si può derivare:

$\beta, \alpha\beta, \alpha\alpha\beta, \alpha\alpha\alpha\beta, \dots$

Questo equivale ad avere nella grammatica la produzione:

$$X \rightarrow \beta + \alpha X$$

Si noti che essa è ricorsiva, cioè la categoria sintattica X è definita in termini di se stessa.

Per illustrare la nozione di grammatica e di linguaggio generato presentiamo qui di seguito alcuni esempi.

Iniziamo con uno tratto dalla lingua italiana. Sia data la grammatica:

$V = \{ \text{il, lo, gatto, topo, sasso, mangia, beve} \}$

$N = \{ \langle \text{frase} \rangle, \langle \text{soggetto} \rangle, \langle \text{verbo} \rangle, \langle \text{complemento-oggetto} \rangle, \langle \text{articolo} \rangle, \langle \text{nome} \rangle \}$

$S = \langle \text{frase} \rangle$

P consiste di:

$\langle \text{frase} \rangle ::= \langle \text{soggetto} \rangle \langle \text{verbo} \rangle \langle \text{complemento-oggetto} \rangle$

$\langle \text{soggetto} \rangle ::= \langle \text{articolo} \rangle \langle \text{nome} \rangle$

$\langle \text{articolo} \rangle ::= \text{il} \mid \text{lo}$

$\langle \text{nome} \rangle ::= \text{gatto} \mid \text{topo} \mid \text{sasso}$

$\langle \text{verbo} \rangle ::= \text{mangia} \mid \text{beve}$

$\langle \text{complemento-oggetto} \rangle ::= \langle \text{articolo} \rangle \langle \text{nome} \rangle.$

L'alfabeto V è costituito da simboli terminali che sono alcune parole della lingua italiana. L'insieme dei non terminali contiene alcune categorie sintattiche della lingua italiana. Il linguaggio generato dalla grammatica precedente è il sottoinsieme delle frasi della lingua italiana ottenibili a partire dal simbolo iniziale $\langle \text{frase} \rangle$ mediante le regole elencate, per esempio: 'il gatto mangia il topo', 'il topo beve il sasso', ecc.

Il processo di derivazione di una frase mediante un grammatica può essere convenientemente illustrato mediante un albero, detto albero di derivazione sintattica, o più semplicemente *albero sintattico* (si veda il par. 3.2. dell'appendice per la definizione di albero). Piuttosto che definire formalmente la nozione di albero sintattico, la introduciamo attraverso l'esempio in fig. 1, che mostra la derivazione della frase 'il gatto mangia il topo'.

Presentiamo ora un altro esempio: una grammatica che genera il linguaggio dei numeri interi di una o due cifre senza segno:

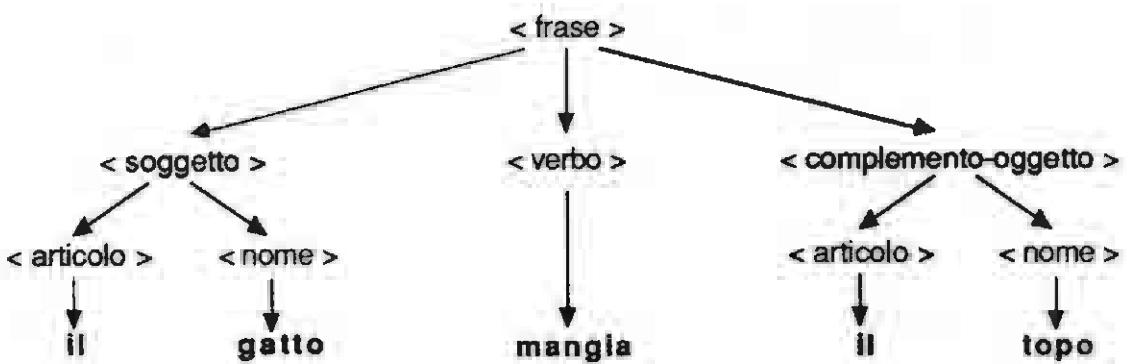


Fig. 1 - Esempio di albero sintattico

$$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$N = \{\text{<intero-senza-segno>}, \text{<cifra-non-nulla>}, \text{<cifra>}\}$$

$$S = \text{<intero-senza-segno>}$$

P consiste di:

$$\text{<intero-senza-segno>} ::= \{\text{<cifra-non-nulla>} | \text{<cifra>}\}$$

$$\text{<cifra>} ::= \text{<cifra-non-nulla>} | 0$$

$$\text{<cifra-non-nulla>} ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.$$

Abbiamo distinto la categoria sintattica **<cifra>** e **<cifra-non-nulla>** per non includere nel linguaggio numeri in cui la cifra delle decine sia zero. Per cui: 0, 1 e 59 sono numeri del linguaggio appena descritto, mentre 07 e 05 non lo sono.

La fig. 2 presenta l'albero sintattico per il numero 59.



Fig. 2 - Albero sintattico per la generazione del numero 59

Presentiamo ora una grammatica G_1 per descrivere i numeri interi di lunghezza qualsiasi, con o senza segno:

$$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \cup \{+, -\}$$

$$N = \{\text{<intero>}, \text{<intero-senza-segno>}, \text{<numero>}, \text{<cifra-non-nulla>}, \text{<cifra>}\}$$

$$S = \text{<intero>}$$

P consiste di:

```

<intero> ::= [+ | -] <intero-senza-segno>
<intero-senza-segno> ::= <cifra> | <cifra-non-nulla><numero>
<numero> ::= <cifra> | <cifra><numero>
<cifra> ::= <cifra-non-nulla> | 0
<cifra-non-nulla> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

```

Si noti che la terza regola è ricorsiva, cioè essa è definita seguendo lo schema induttivo presentato nel par. 4.1 dell'appendice; essa serve a generare numeri come sequenze di cifre di lunghezza arbitraria (ma finita). Abbiamo usato il solito artificio di introdurre la categoria sintattica <cifra-non-nulla> per non permettere numeri la cui cifra più a sinistra è zero.

Una grammatica G_2 per lo stesso linguaggio può essere scritta in EBNF nel seguente modo:

$$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \cup \{+, -\}$$

$$N = \{\langle \text{intero} \rangle, \langle \text{intero-senza-segno} \rangle, \langle \text{cifra-non-nulla} \rangle, \langle \text{cifra} \rangle\}$$

$$S = \langle \text{intero} \rangle$$

P consiste di:

```

<intero> ::= [+ | -] <intero-senza-segno>
<intero-senza-segno> ::= 0 | <cifra-non-nulla> | <cifra>
<cifra> ::= <cifra-non-nulla> | 0
<cifra-non-nulla> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

```

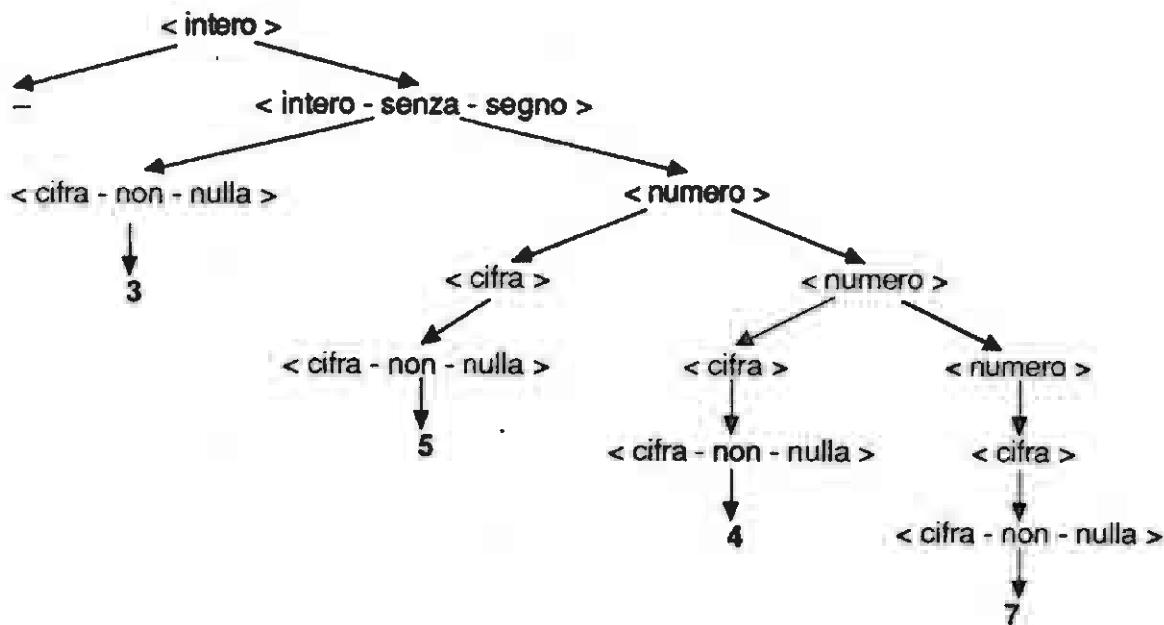
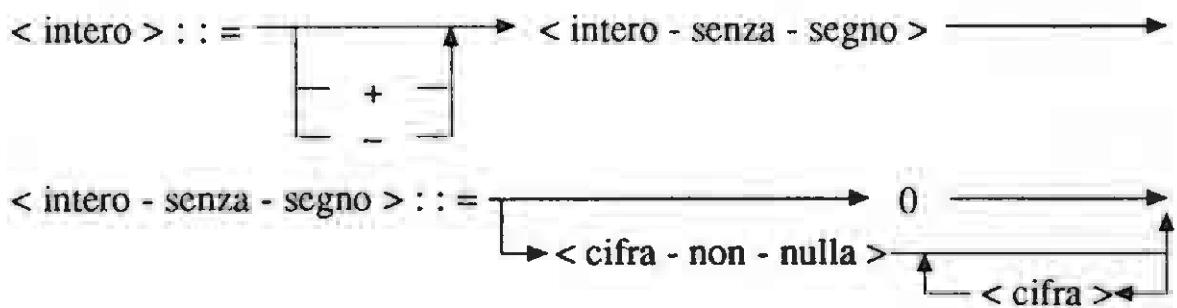


Fig. 3 - Albero sintattico per la generazione del numero -3547

Nella fig. 3 mostriamo l'albero sintattico per il numero -3547 generato da G_1 .

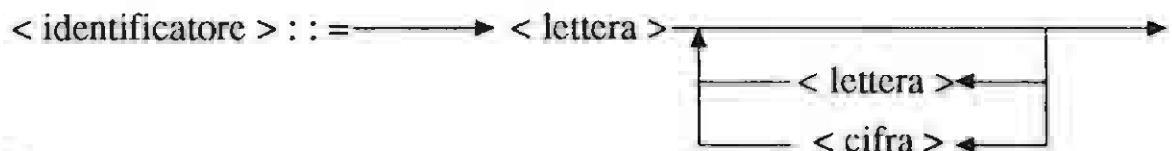
Talvolta nella definizione dei linguaggi la sintassi non viene descritta solo mediante regole espresse in EBNF, ma anche mediante i cosiddetti diagrammi sintattici. Questi sono dei grafi (per la definizione di grafo si veda il par. 3 dell'appendice) i cui nodi sono etichettati con i simboli terminali o non terminali del linguaggio collegati da archi orientati. La presenza di un arco da un nodo n_i ad un nodo n_j significa che il simbolo n_i è seguito dal simbolo n_j nella parte destra della regola rappresentata in forma di diagramma. La presenza di più di un arco in uscita (o in ingresso) da un nodo significa un'alternativa. Per comodità grafica possono essere introdotti dei nodi fintizi per rappresentare le diramazioni. Per esempio:



è la rappresentazione diagrammatica delle prime due regole della grammatica G_2 presentata sopra; mentre la regola

<identificatore> ::= <lettera> {<lettera> | <cifra>}

che definisce gli identificatori in un linguaggio di programmazione tipo Pascal può essere rappresentata in forma diagrammatica come segue:



dove <lettera> è l'insieme dei caratteri dell'alfabeto inglese e <cifra> è l'insieme delle dieci cifre decimali. Gli identificatori sono quindi definiti come una sequenza di una o più lettere o cifre che inizia con una lettera.

1.2. Sintassi dei linguaggi di programmazione

La definizione della sintassi di un linguaggio di programmazione viene data definendo il lessico, cioè l'insieme dei simboli terminali del linguaggio, un alfabeto di simboli non terminali, tra cui ne viene scelto uno come iniziale (di solito `<programma>`) ed un insieme di regole in EBNF.

Il lessico è costituito da un alfabeto finito di parole chiave (cioè sequenze di caratteri riservate, nel caso del Pascal esse sono `begin`, `end`, `if`, `then`, `case`, ecc.) di simboli speciali, corrispondenti ad operatori e simboli di interruzione, e da un alfabeto di caratteri e cifre, mediante i quali vengono costruiti gli identificatori e le costanti.

Senza addentrarci in considerazioni che sono di pertinenza della teoria dei linguaggi facciamo notare che:

- una grammatica definisce un unico linguaggio (finito o infinito), la cui esistenza ed unicità si possono dimostrare anche in presenza di regole ricorsive (infatti una grammatica è un metodo per definire induttivamente un insieme che realizza lo schema presentato nel par. 4.1 dell'appendice);
- un linguaggio può essere generato da più di una grammatica (vedere, per esempio, le due differenti grammatiche precedentemente introdotte per i numeri interi);
- alcune grammatiche sono ambigue, cioè esistono elementi del linguaggio che possono essere generati mediante diverse derivazioni (in altre parole, ad alcuni elementi del linguaggio possono essere associati diversi alberi sintattici). Nella definizione dei linguaggi di programmazione questa situazione va evitata, in quanto creerebbe delle ambiguità sul significato da dare ai programmi.

Come esempio di grammatica ambigua si consideri la seguente che, a partire dal simbolo iniziale `<expr>`, definisce il linguaggio delle espressioni costituite da tre simboli di variabile `x`, `y` e `z`, dai simboli di operatore `+` e `*` e dalle parentesi aperta e chiusa:

$$\text{<expr>} ::= \text{x} \mid \text{y} \mid \text{z} \mid (\text{<expr>}) \mid \text{<expr>} + \text{<expr>} \mid \text{<expr>} * \text{<expr>}$$

Nella fig. 4 mostriamo come all'espressione `x+y+z` possano essere associati due alberi sintattici.

Il fatto che l'espressione `x+y+z` abbia due alberi sintattici si rivela un inconveniente nel momento in cui a tale espressione si tenti di dare un significato come espressione aritmetica. Infatti, il significato delle frasi (in questo caso l'ordine di esecuzione degli operatori) viene determinato in



Fig. 4 - Due alberi sintattici per $x+y*z$

funzione dell'albero di derivazione. Nel caso del primo albero sintattico l'espressione viene valutata eseguendo prima $y*z$ e poi sommando x , nel secondo viene prima sommato x a y , poi il risultato viene moltiplicato per z ; i due valori così ottenuti sono, in generale, diversi.

Una grammatica ambigua può in alcuni casi essere riscritta in un'altra forma in modo che non sia più ambigua. Il lettore interessato ad una soluzione per il problema dell'ambiguità nel caso delle espressioni aritmetiche può vedere la grammatica del Pascal, dove non solo è stato risolto il problema dell'ambiguità, ma è stata data la precedenza agli operatori + e *, seguendo le convenzioni dell'algebra elementare.

Per quanto nella definizione delle grammatiche di linguaggi di programmazione si cerchi di evitare ambiguità, talvolta queste non sono eliminabili. In questo caso il problema dell'assegnazione di un unico significato alle frasi si risolve stabilendo una convenzione su come va interpretato il costrutto ambiguo. A titolo di esempio, un caso di regola ambigua in Pascal si ha nella definizione dell'istruzione if-then-else: essa ha la forma:

- $\langle \text{istruzione-if} \rangle ::= \text{if } \langle \text{espressione} \rangle \text{ then } \langle \text{istruzione} \rangle$
 $\quad [\text{else} \langle \text{istruzione} \rangle]$

Poiché $\langle \text{istruzione} \rangle$ può anche essere una $\langle \text{istruzione-if} \rangle$, la regola appena introdotta può generare un (frammento di) programma del tipo:

if ... then ... if ... then ... else ...

in cui non è evidente se la parte **else** fa riferimento al primo od al secondo **if**. In Pascal questa ambiguità viene risolta decidendo che, per convenzione, l'**else** fa riferimento all'ultimo **if** presente nella frase, in questo caso il secondo.

Nella definizione della sintassi dei linguaggi di programmazione ci sono degli aspetti che sfuggono alla definizione fornita mediante una grammatica in forma EBNF. Se si pensa infatti che la generazione di una frase del linguaggio è rappresentata mediante un albero, si vede come la grammatica non esprime nessun vincolo tra diverse parti della frase, in quanto non c'è nessuna correlazione tra i diversi sottoalberi di un albero sintattico. Spesso invece nei linguaggi di programmazione sono definiti dei vincoli tra le diverse parti di un programma. Per esempio, nel definire il Pascal si debbono anche enunciare regole del tipo: "un identificatore non può essere usato prima che sia stato dichiarato" oppure "in una attivazione di procedura i parametri attuali debbono corrispondere in numero e tipo ai parametri formali della corrispondente dichiarazione", oppure ancora "in un blocco di programma non ci possono essere due dichiarazioni dello stesso identificatore", ecc. Questi aspetti, che si chiamano di sintassi contestuale, vengono di solito espressi a parole, con delle frasi aggiunte a commento delle regole espresse in EBNF.

La verifica della correttezza sintattica di un programma è un compito che viene eseguito dall'elaboratore stesso: più precisamente, per un dato linguaggio di programmazione viene scritto un programma detto analizzatore sintattico che esegue la verifica basandosi sulle regole che costituiscono la grammatica del linguaggio in oggetto. In pratica quello che un analizzatore sintattico fa è, data una sequenza di simboli, tentare di costruire un albero sintattico per essa. Per far ciò il programma si basa sulla definizione del lessico, cioè del vocabolario dei simboli terminali del linguaggio, sulle regole grammaticali espresse in EBNF, sulle convenzioni notazionali e sui vincoli contestuali. Quindi il ruolo di un analizzatore sintattico è quello di verificare la correttezza sintattica di un programma, controllando i seguenti aspetti:

- che il lessico usato nel programma sia corretto, cioè tutti i simboli (parole chiave, identificatori, ecc.) siano legali nel linguaggio. Ad esempio, un programma Pascal viene rifiutato se in esso si ha una occorrenza di una parola 'than', o qualunque altra stringa di caratteri, dove ci sarebbe voluto un 'then', oppure se in esso compare il simbolo 3L65, che non è un identificatore legale e neanche un numero;
- che le regole grammaticali siano rispettate. Ad esempio, un programma Pascal viene rifiutato se esso contiene frammenti del tipo:

A := B := C+1;

oppure:

if a := b then s := 0 else s := 1;

- che i vincoli imposti dalle restrizioni di tipo contestuale siano rispettati. Ad esempio, un programma Pascal viene rifiutato se esso contiene frammenti del tipo:

```
type t = ... ;
var t : ... ;
```

oppure:

```
var bool: integer;
...
if bool then ...
else...;
```

Le verifiche suddette vengono svolte da parti dell'analizzatore sintattico che prendono il nome rispettivamente di: analizzatore lessicale, analizzatore grammaticale e analizzatore di sintassi contestuale o di semantica statica. L'analizzatore sintattico svolge le tre verifiche in un programma in tre fasi che logicamente si susseguono, ma che in pratica possono anche essere svolte simultaneamente. Tipicamente gli analizzatori sintattici per il Pascal vengono detti 'ad una passata', in quanto le tre verifiche suddette vengono effettuate simultaneamente durante un'unica scansione del programma.

In linea di principio, durante la scansione di un programma, l'analisi sintattica dovrebbe proseguire in caso di successo e bloccarsi non appena si incontri un errore. Di fatto gli analizzatori sono costruiti in modo che, dopo un errore, si tenta un recupero e l'analisi viene portata avanti fino alla fine del testo del programma. Questa strategia, pur se utile perché permette di analizzare tutto un programma anche nel caso che esso contenga più di un errore, non è completamente affidabile in quanto l'unico errore sulla cui diagnosi non si hanno dubbi è il primo incontrato: il resto del testo del programma può essere analizzato in maniera scorretta dall'analizzatore a causa di un errato recupero.

2. SEMANTICA

Spesso nel definire un linguaggio di programmazione ci si limita a definire in modo formale solo la sintassi, mentre la definizione della semantica viene presentata informalmente usando il linguaggio naturale. Questo ha lo scopo di illustrare il significato intuitivo dei costrutti, senza oberare il lettore con formalismi che tendono a diventare complicati. Lo svantaggio di questo

approccio pragmatico è comunque non trascurabile, se si pensa alle ambiguità e imprecisioni che sono intrinseche nelle descrizioni fornite in linguaggio naturale.

Molti sono i vantaggi di una definizione formale della semantica dei linguaggi di programmazione. Qui di seguito accenniamo brevemente ai più evidenti:

1. ne beneficia il programmatore, che può comprendere con esattezza il significato dei costrutti che il linguaggio gli fornisce, ed è messo in grado, almeno in linea di principio, di costruire prove formali di correttezza dei programmi che scrive;
2. ne beneficia l'implementatore, cioè colui che scrive un traduttore per un particolare elaboratore, in quanto trova nella definizione formale della semantica una guida precisa per la costruzione di un traduttore corretto;
3. ultimo, ma non meno importante, ne beneficia il progettista di linguaggi, che ha a disposizione degli strumenti formali di progetto. Infatti questi, basandosi su una descrizione formale della semantica, può dimostrare proprietà dei costrutti del linguaggio.

In questo paragrafo accenneremo al problema della definizione formale della semantica fornendo alcuni esempi basati sul semplice linguaggio di programmazione presentato nell'introduzione. Nel resto del libro adotteremo l'approccio pragmatico e non definiremo formalmente la semantica dei costrutti linguistici trattati, rimandando questo a testi specializzati.

Mentre esiste di fatto un solo metodo per definire la sintassi dei linguaggi di programmazione, per quanto riguarda la semantica ne esistono almeno due, che si adattano a diversi scopi: metodo operazionale e metodo denotazionale.

2.1. Metodo operazionale

Definire una semantica operazionale per un linguaggio di programmazione significa definire una macchina astratta (modello astratto di calcolatore) e definire come la esecuzione delle varie istruzioni del linguaggio viene condotta su tale macchina. La configurazione istantanea della macchina astratta viene descritta mediante uno stato; la semantica di ciascun costrutto del linguaggio consiste nella descrizione di una transizione di stato, cioè come la macchina passa da uno stato ad uno successivo per effetto della esecuzione del costrutto stesso. Il significato di un programma è definito dal comportamento della macchina astratta per effetto della esecuzione del programma, cioè dalla

sequenza di stati che la macchina astratta assume. Ovviamente tale sequenza di stati sarà finita se la esecuzione del programma termina, infinita altrimenti.

Presentiamo, a titolo di esempio, la semantica operazionale per il linguaggio descritto nell'introduzione. Lo stato s della macchina astratta M consiste di tre componenti:

$$s = \langle is, os, mem \rangle$$

dove is è una sequenza di elementi di input (essa descrive lo stato dell'input, cioè l'insieme di dati che devono essere immessi in ingresso), os è una sequenza di elementi di output (essa descrive lo stato dell'output, cioè l'insieme di dati già forniti in uscita), mem è una sequenza finita di coppie $\langle nome, valore \rangle$ che descrive la configurazione della memoria. Una transizione di stato per M è descritta come:

$$s \rightarrow s'$$

cioè:

$$\langle is, os, mem \rangle \rightarrow \langle is', os', mem' \rangle$$

dove almeno uno degli elementi della terna di destra è diverso dal corrispondente elemento di sinistra.

Supponiamo di disporre delle seguenti operazioni:

$primo(is)$, che dà come risultato il primo elemento della sequenza di input, e dà un errore se essa è vuota;

$resto(is)$, che dà come risultato la sequenza di input cui è stato tolto il primo elemento, e dà un errore se essa è vuota;

$app(v, os)$, che aggiunge l'elemento v alla sequenza di output os ;

$mod(mem, \langle n, v \rangle)$, che modifica mem aggiungendovi la coppia $\langle n, v \rangle$, e rimpiazzando la coppia che ha n come primo elemento, se già presente;

$val(n, mem)$, che restituisce l'elemento v se la coppia $\langle n, v \rangle$ è presente in mem , dà un errore se in mem non è presente nessuna coppia che abbia n come primo membro.

La semantica dell'istruzione di lettura

$$readln(n)$$

in uno stato $s = \langle is, os, mem \rangle$, cioè l'effetto della esecuzione della istruzione di lettura a partire da s , può essere descritta come la seguente transizione di M:

$$Sem(readln(n)) =$$

$$\langle is, os, mem \rangle \rightarrow \langle resto(is), os, mod(mem, \langle n, primo(is) \rangle) \rangle$$

La semantica dell'istruzione di scrittura

writeln(n)

in uno stato $s = \langle is, os, mem \rangle$ può essere descritta come la seguente transizione di M :

$$\begin{aligned} Sem(writeln(n)) = \\ \langle is, os, mem \rangle \rightarrow \langle is, app(val(n,mem), os), mem \rangle \end{aligned}$$

Consideriamo ora la semantica dell'istruzione di assegnazione

n := exp

in uno stato $s = \langle is, os, mem \rangle$. Supponiamo che v sia il valore risultante dalla valutazione in $\langle is, os, mem \rangle$ dell'espressione exp . La semantica di $n := exp$ in s può essere così descritta:

$$\begin{aligned} Sem(n := exp) = \\ \langle is, os, mem \rangle \rightarrow \langle is, os, mod(mem, \langle n, v \rangle) \rangle \end{aligned}$$

Si noti che, nel linguaggio in esempio, la valutazione di una espressione (aritmetica o booleana) non modifica lo stato, cosa non sempre vera nei linguaggi di programmazione reali (si veda in proposito il par. 2.6 del cap. 2).

La semantica della sequenza di istruzioni

i₁ ; i₂

in uno stato s consiste in due transizioni per M: la transizione in s' determinata dall'esecuzione di i_1 , a partire dallo stato s e dalla transizione determinata dall'esecuzione di i_2 in s' .

La semantica della istruzione condizionale

if *bool* **then** *i₁* **else** *i₂*

in uno stato s consiste in una transizione per M: la transizione determinata da i_1 , a partire dallo stato s , oppure la transizione determinata da i_2 a partire dallo stato s , a seconda che il valore di *bool*, valutato in s , sia vero o falso, rispettivamente.

La semantica della istruzione di iterazione

while *bool* **do** *i*

in uno stato s consiste in una sequenza di transizioni per M: le transizioni determinate dalla esecuzione ripetuta delle istruzioni che costituiscono il corpo

i dell'iterazione, fintantoché la valutazione dell'espressione booleana *bool* risulta vera.

A conclusione dell'esempio facciamo notare che, con questo approccio alla semantica dei linguaggi di programmazione, il significato di un programma P su un input *is* consiste nella sequenza (finita, se il programma termina, altrimenti infinita) di transizioni determinata sulla macchina M dalla esecuzione delle varie istruzioni che compongono P, a partire da uno stato iniziale $s_0 = \langle is, \emptyset, \emptyset \rangle$, dove \emptyset denota l'output e la memoria vuoti.

2.2. Metodo denotazionale

Dare una semantica denotazionale (detta anche semantica matematica) per un linguaggio di programmazione significa fornire un metodo rigoroso per associare ad ogni programma del linguaggio l'oggetto matematico da esso denotato. Tale oggetto matematico è quindi il significato del programma. Esistono due metodi per la definizione della semantica denotazionale, uno funzionale, l'altro relazionale o assiomatico.

Un metodo denotazionale può essere basato su un calcolo funzionale, in cui ad ogni programma viene associata la funzione da esso calcolata. Con questo approccio la semantica dei programmi di un linguaggio di programmazione L è descritta come:

$$F = \{f_p : I_p \rightarrow O_p\}$$

in cui F è un insieme di funzioni dall'insieme I_p dei valori di input per i programmi di L, all'insieme O_p dei valori di output dei programmi di L. Le funzioni f_p possono essere totali o parziali, cioè, si può avere che $f_p(i)$ non è definito. La semantica di un programma P di L, che sull'input *i* termina e produce l'output *o*, sarà la funzione f_p di F tale che $f_p(i) = o$; se P su *i* non termina, $f_p(i)$ sarà indefinita. In altre parole, detta $S : L \rightarrow F$ la funzione che associa ad un programma P la sua semantica, cioè $S(P) = f_p$, si avrà che, per un qualunque input *i*:

$$S(P)(i) = S(P, i) = f_p(i)$$

Presentiamo qui di seguito la semantica denotazionale funzionale del linguaggio già visto in precedenza. La memoria viene descritta come una funzione da un insieme finito di nomi ad un insieme finito di valori. Sia $\sigma : N \rightarrow V$ tale funzione; la semantica delle istruzioni va definita relativamente ad un σ .

Definiamo $\sigma[v/n]$ come la funzione $\sigma': N \rightarrow V$ tale che $\sigma'(m) = \sigma(m)$ per ogni $m \neq n$, e $\sigma'(m) = v$ per $m = n$. La semantica di un programma P che ha una variabile di input n ed una variabile di output z , ed al quale viene fornito un input i , sarà definita come:

$$S(P, i) = \text{output}(z, \text{sem}(P, \text{input}(n, i)))$$

dove, chiamando Σ l'insieme dei σ , e C l'insieme delle istruzioni (o comandi) del linguaggio, tranne quelle di lettura e scrittura, avremo che :

$$\text{output} : N \times \Sigma \rightarrow V$$

cioè output è una funzione che, dato un nome di variabile ed una memoria, fornisce un valore (si noti che output sarà indefinito se uno dei suoi argomenti è indefinito);

$$\text{sem} : C \times \Sigma \rightarrow \Sigma$$

sem è una funzione che, data una istruzione ed una memoria fornisce una nuova memoria se la sua esecuzione termina, è indefinita altrimenti; ed infine:

$$\text{input} : N \times V \rightarrow \Sigma$$

input è una funzione che, da coppie <nomi, valori> crea una memoria.

Definiamo:

$$\begin{aligned} \text{input}(n, i) &= \sigma \text{ tale che } \sigma(n) = i \\ \text{output}(z, \sigma) &= \sigma(z) \end{aligned}$$

Sia val la funzione che valuta le espressioni del linguaggio, definiamo sem per ciascuna istruzione del linguaggio:

$$\begin{aligned} \text{sem}(n := \text{exp}, \sigma) &= \sigma[\text{val}(\text{exp}, \sigma)/n] \\ \text{sem}(i_1; i_2, \sigma) &= \text{sem}(i_2, \text{sem}(i_1, \sigma)) \\ \text{sem}(\text{if } \text{bool} \text{ then } i_1 \text{ else } i_2, \sigma) &= \\ &\quad \text{sem}(i_1, \sigma) \text{ se } \text{val}(\text{bool}, \sigma) \text{ è vero,} \\ &\quad \text{sem}(i_2, \sigma) \text{ se } \text{val}(\text{bool}, \sigma) \text{ è falso.} \\ \text{sem}(\text{while } \text{bool} \text{ do } i, \sigma) &= \\ &\quad \text{sem}(\text{while } \text{bool} \text{ do } \text{sem}(i, \sigma)) \text{ se } \text{val}(\text{bool}, \sigma) \text{ è vero,} \\ &\quad \sigma \text{ se } \text{val}(\text{bool}, \sigma) \text{ è falso.} \end{aligned}$$

Si noti come la semantica dell'istruzione di iterazione è definita da un'equazione funzionale ricorsiva. Questa è l'unica istruzione del linguaggio in esempio la cui esecuzione può non terminare, quindi l'unica la cui semantica

può fornire un σ indefinito.

La semantica di un linguaggio di programmazione definita col metodo denotazionale funzionale risulta quindi essere un insieme di equazioni funzionali. La semantica di un programma è la funzione risultato dell'applicazione di tali equazioni al programma stesso.

Come abbiamo già detto, un metodo denotazionale può essere basato su un calcolo relazionale in cui ad ogni programma viene associata la relazione da esso calcolata. In questo caso la semantica dei programmi di un linguaggio di programmazione L è descritta come:

$$R = \{R_p(I_p, O_p)\}$$

in cui R è un insieme di relazioni R_p tra l'insieme I_p dei valori di input per i programmi di L e l'insieme O_p dei valori di output dei programmi di L. La semantica di un programma P di L, che sull'input i produce l'output o , sarà la relazione R_p di R tale che $R_p(i, o)$. In altre parole, se un programma P per un input i termina e fornisce un output o , si avrà che vale la relazione $R_p(i, o)$, o ancora $\langle i, o \rangle \in R_p$, altrimenti la relazione è indefinita. Il calcolo relazionale spesso usato è il calcolo dei predici (vedere il par. 5 dell'appendice). In questo caso la relazione che lega i dati di ingresso ai dati di uscita viene espressa mediante due predici: se il primo (precondizione) è vero sui dati di ingresso e se il programma termina su quei dati, allora il secondo (postcondizione) è vero sui dati di uscita.

Nel caso del metodo denotazionale relazionale definire la semantica di un linguaggio consiste nel definire come le precondizioni e le postcondizioni siano correlate per ciascuna istruzione del linguaggio, dove le precondizioni descrivono la situazione della memoria prima dell'esecuzione dell'istruzione, le postcondizioni descrivono la nuova situazione dopo l'esecuzione di tale istruzione. Questo viene ottenuto fornendo assiomi (onde il nome di metodo assiomatico) o regole di inferenza per ciascuna istruzione. Nel caso del linguaggio in esempio avremo:

$$\begin{aligned} & \{ P \} \text{readln}(n) \{ P[i/n] \} \\ & \{ P \} \text{writeln}(n) \{ P \} \\ & \{ P \} n := \text{exp} \{ P[V(\text{exp})/n] \} \end{aligned}$$

In questi assiomi i è il valore fornito in input e V è la funzione che valuta le espressioni aritmetiche. I predici che indicano la precondizione e la postcondizione sono stati racchiusi tra parentesi graffe e scritti rispettivamente a sinistra e a destra della istruzione; la precondizione è stata indicata con P, la

postcondizione è sempre P, eventualmente modificato con una sostituzione. $P[x/y]$ sta ad indicare P in cui x ha rimpiazzato tutte le occorrenze di y. Si noti che la notazione è la stessa di $\sigma[v/n]$ visto sopra, anche se le due operazioni sono diverse.

Passiamo ora alle regole di inferenza che descrivono la semantica della sequenza, del condizionale e dell'iterazione:

$$\{P_1\} i_1 \{P_2\} \quad \{P_2\} i_2 \{P_3\}$$

$$\{P_1\} i_1 ; i_2 \{P_3\}$$

$$\{P_1 \wedge B(bool) = true\} i_1 \{P_2\} \quad \{P_1 \wedge B(bool) = false\} i_2 \{P_2\}$$

$$\{P_1\} \text{ if } bool \text{ then } i_1 \text{ else } i_2 \{P_2\}$$

$$\{P \wedge B(bool) = true\} i \{P\}$$

$$\{P\} \text{ while } bool \text{ do } i \{P \wedge B(bool) = false\}$$

dove B è la funzione che valuta le espressioni booleane. Le prime due regole di inferenza dovrebbero essere evidenti; la terza ci dice che, se l'esecuzione del corpo dell'istruzione **while** non modifica la verità di P, fin quando la espressione booleana che controlla l'istruzione **while** resta vera, allora l'esecuzione di tutta l'istruzione **while** non modifica la verità di P. In altre parole, se P è vero prima di eseguire l'istruzione **while** e questa esecuzione termina, P sarà ancora vero dopo, ed a tal punto *bool* sarà diventata falsa. Nel caso dell'istruzione **while**, il predicato P è anche detto l'invariante del ciclo.

Dato un programma *Prog*, la sua semantica assiomatica è la coppia di prediciati P e Q per i quali vale l'espressione $\{P\} Prog \{Q\}$.

Il metodo assiomatico per la definizione della semantica di un linguaggio di programmazione consente di sviluppare un metodo formale per dimostrare la correttezza dei programmi di quel linguaggio. Infatti stabilire che il programma *Prog* è corretto rispetto alle condizioni P e Q significa dimostrare, usando gli assiomi e le regole di inferenza che definiscono la semantica del linguaggio, che l'espressione $\{P\} Prog \{Q\}$ è soddisfatta. Questo metodo verrà ripreso nel cap. 5.

Anche se forse la presentazione precedente è stata troppo schematica per poterlo apprezzare, facciamo notare che il metodo denotazionale permette una

descrizione degli effetti di un programma ad un maggiore livello di astrazione rispetto al metodo operazionale. Quindi, mentre una descrizione della semantica denotazionale di un linguaggio si presta di più a fornire una guida al programmatore ed alla costruzione delle prove formali di correttezza dei programmi in esso scritti, la descrizione operazionale della semantica si presta a fornire una guida all'implementatore del linguaggio.

Nel definire la semantica di un linguaggio di programmazione, spesso, proprio per non rinunciare ai vantaggi di nessuno dei due metodi, si fornisce una semantica operazionale ed una denotazionale e se ne dimostra (matematicamente) l'equivalenza.

2.

Linguaggi di programmazione: dati e controllo

Il potere espressivo di un linguaggio di programmazione è caratterizzato da due aspetti:

1. i tipi di dato che nel linguaggio si possono esprimere e manipolare;
2. le operazioni esprimibili direttamente o per composizione.

In questo capitolo esaminiamo i due aspetti facendo particolare riferimento al linguaggio Pascal. Nel par. 1 parliamo dei tipi di dato nei linguaggi di programmazione, mentre nel par. 2 trattiamo il problema del controllo, cioè la specifica delle operazioni che l'elaboratore deve eseguire. Viene soprattutto illustrato il problema delle astrazioni funzionali, cioè l'introduzione di nuove operazioni ottenute fornendo al programmatore la possibilità di definire procedure (astrazione della nozione di istruzione) e funzioni (astrazione della nozione di operatore in una espressione). Nel seguito, quando non si vorrà distinguere tra procedure e funzioni, si parlerà genericamente di "unità di programma".

1. DATI

Una delle caratteristiche salienti di un linguaggio di programmazione è costituita dai tipi di dato che esso permette di rappresentare o direttamente (tipi primitivi) o mediante delle definizioni (tipi di dato definiti da utente).

1.1. Tipi di dato e rappresentazioni

Un tipo di dato T (o tipo di dato astratto, o anche tipo di dato matematico) è definito come un dominio di valori D, alcune funzioni f_1, f_2, \dots, f_n e predici P_i ,

P_2, \dots, P_r su questo dominio ed alcune costanti c_1, c_2, \dots, c_k .

$$T = \langle D, f_1, f_2, \dots, f_n, P_1, P_2, \dots, P_r, c_1, c_2, \dots, c_k \rangle$$

Le funzioni e i predicati sono chiamati genericamente operazioni. Per esempio possiamo definire un tipo di dato per i numeri naturali come:

$$\text{NAT} = \langle N, +, *, =, >, <, 0, 1 \rangle$$

dove le operazioni $+, *, =, <, >$ hanno il noto significato.

Definire un tipo di dato per i numeri naturali significa quindi specificare non solo il dominio N , ma anche le operazioni che trasformano naturali in naturali, i predicati che su essi si applicano ed alcune costanti significative.

Perché un linguaggio possa esprimere delle manipolazioni su oggetti di un tipo di dato esso deve permettere di rappresentarli. Una rappresentazione per un tipo di dato è una descrizione del tipo di dato in termini delle strutture linguistiche fornite dal linguaggio.

Gli aspetti relativi ai tipi di dato e alla loro rappresentazione saranno trattati nel cap. 3. Qui ci si concentrerà su alcuni problemi legati ai tipi di dato del linguaggio Pascal.

1.2. Tipi di dato in Pascal

Il Pascal fornisce all'utente alcuni tipi di dato primitivi elementari. Essi sono il tipo degli interi, il tipo dei reali, il tipo dei caratteri e quello dei booleani. Questi tipi sono detti primitivi in quanto forniti dal linguaggio (*integer*, *real*, *char* e *boolean*, rispettivamente), elementari in quanto il loro dominio è costituito da elementi atomici, cioè non strutturati e non decomponibili.

Si rimanda ad un testo sul Pascal per una definizione dei domini e delle operazioni fornite per i dati rappresentabili mediante questi tipi primitivi. Qui facciamo solo notare che il Pascal, come altri linguaggi di programmazione, permette che operazioni indicate con lo stesso simbolo possano essere usate nell'ambito di tipi di dato diversi: per esempio $+$ può esser usato per la somma tra interi e tra reali, $>$ per confronto di interi, reali, caratteri e booleani. Questa possibilità prende il nome di sovraccarico degli operatori (*overloading*) ed ha come conseguenza che se un simbolo è usato come operatore nell'ambito di diversi tipi di dato, la sua semantica è diversa a seconda del tipo degli argomenti cui è applicato. Per esempio: il simbolo $+$ su argomenti interi significa la somma tra interi, a risultato intero; su argomenti reali significa la somma tra reali,

a risultato reale.

Accanto ai tipi primitivi il Pascal fornisce all'utente la primitiva linguistica **type** che permette di definire, dando loro un nome o in modo anonimo, tipi di dato di sua scelta. Questi possono essere ancora tipi elementari o strutturati. Riguardo ai primi, essi si suddividono in tipi enumerati, definiti fornendo la enumerazione di tutte le costanti che compaiono nel dominio, e tipi intervallo, definiti fornendo gli estremi di un intervallo degli interi, dei caratteri o di un tipo enumerato. I tipi strutturati in Pascal sono definiti mediante l'uso delle primitive linguistiche **array**, **record** e mediante puntatori.

Mentre per una trattazione dettagliata dell'uso delle primitive linguistiche per la rappresentazione di tipi di dato strutturati si rimanda al cap. 3, qui facciamo notare che le primitive linguistiche del Pascal non sono in generale sufficienti a fornire all'utente la possibilità di definire rappresentazioni per tipi astratti di dato. Infatti esse permettono al più di definire il dominio di un tipo di dato, ma non di definire contestualmente gli operatori ad esso relativi. In alcuni casi non è neanche possibile rappresentare il dominio del tipo. Si consideri ad esempio l'anello degli interi modulo due, che ha come dominio l'insieme dei numeri pari. Non esiste in Pascal un modo di rappresentare un tipo il cui dominio è l'insieme dei numeri pari.

Per quanto riguarda gli operatori relativi ai tipi di dato definiti da utente in Pascal, essi possono essere definiti attraverso funzioni o procedure, ossia come astrazioni sul controllo (vedere par. 2.3) e per essi non è possibile nessun sovraccarico. L'unica forma di sovraccarico in Pascal si ha in quanto i tipi di dato elementari definiti da utente importano gli operatori dal tipo di dato di cui sono "sottotipi", ossia dal tipo di dato del cui dominio si sta definendo un sottinsieme (per esempio un tipo intervallo degli interi importa gli operatori dagli interi).

1.3. Compatibilità ed equivalenza tra tipi di dato

Spesso nei linguaggi di programmazione si parla di compatibilità tra tipi di dato, intendendo con questo la possibilità di combinare in una espressione o in una assegnazione variabili che siano state dichiarate di tipo diverso oppure la possibilità di legare parametri attuali e formali in procedure e funzioni. Per esempio, si possono costruire espressioni aritmetiche che coinvolgono variabili intere e reali, mentre generalmente non è consentito costruire espressioni in cui si somma una variabile intera ad una booleana. Illustriamo qui brevemente

te il meccanismo che regola tali compatibilità.

Un tipo di dato T_1 è compatibile con un tipo T_2 se D_1 , dominio di T_1 , è contenuto in D_2 , dominio di T_2 .

Per esempio, in Pascal i tipi intervallo sono compatibili con il tipo degli interi o con il tipo enumerato di cui sono intervallo; i tipi intervallo degli interi ed il tipo degli interi sono compatibili con il tipo dei reali.

Se due tipi T_1 e T_2 sono compatibili e op è un'operatore definito per T_2 esso può avere come argomenti sia variabili di tipo T_1 che variabili di tipo T_2 ; naturalmente il risultato sarà un valore di tipo T_2 . In altre parole, se op è un operatore definito per il tipo T_2 , cioè se:

$$op : D_2 \times D_2 \rightarrow D_2$$

allora op potrà anche essere usato come segue:

$$op : D_1 \times D_2 \rightarrow D_2$$

$$op : D_2 \times D_1 \rightarrow D_2$$

Con questo si intende che variabili di tipo T_1 sono automaticamente considerate di tipo T_2 , e su esse op agisce come se fossero di tipo T_2 . Questa operazione è detta forzatura (*coercion*). Se entrambi gli argomenti sono di tipo T_1 si distinguono due casi: l'operatore è definito solo per T_2 ; l'operatore è definito sia per T_1 che per T_2 (in questo caso l'operatore è sovraccarico). Nel primo caso si avrà:

$$op : D_1 \times D_1 \rightarrow D_2$$

nel secondo:

$$op : D_1 \times D_1 \rightarrow D_1$$

Quindi, in Pascal, se I è stata dichiarata variabile intera e R variabile reale, essendo l'operatore “/” non sovraccarico (in quanto la divisione sugli interi è definita da **div**), avremo che:

R/R dà risultato reale

I/R dà risultato reale

R/I dà risultato reale

I/I dà risultato reale

mentre $I \text{ div } I$ dà risultato intero. Inoltre, essendo “+” un operatore sovraccarico, avremo che:

R+R dà risultato reale
I+R dà risultato reale
R+I dà risultato reale
I+I dà risultato intero

Se infine B è una variabile dichiarata di tipo booleano, nessuna operazione è possibile tra I e B oppure tra R e B.

Come già detto, il Pascal permette al programmatore di definire tipi di dato; questo fa sorgere il problema di decidere quando due oggetti hanno lo stesso tipo, cioè quando due tipi, creati con definizioni diverse, possono essere considerati lo stesso.

Purtroppo la definizione del Pascal non fornisce una soluzione a questo problema; essa viene rimandata alla scelta dell'implementatore. In alcune implementazioni del Pascal i tipi sono distinti in base alla loro struttura, per cui due tipi di uguale struttura, anche se di nome diverso o anonimi, sono considerati uguali. In questo caso si parla di *equivalenza strutturale* di tipi. Per esempio, nel contesto:

```
type t = record a : real; b : char end;  
var x : array [boolean] of t;  
y : array[boolean] of record a : real; b : char end;
```

sia x che y hanno lo stesso tipo.

Nella maggior parte delle implementazioni Pascal viene seguito un approccio diverso: ogni costrutto che definisca un tipo in un programma e che non sia un nome di tipo, è considerato caratterizzare un tipo diverso da qualunque altro nel programma, anche se i due tipi hanno la stessa struttura. In questo approccio le variabili x e y dell'esempio precedente hanno tipo diverso, così pure sono diversi tra loro i seguenti tipi:

```
type coordinata = record primo, secondo : real end;  
type complesso = record primo, secondo : real end;
```

per cui se z è una variabile dichiarata di tipo coordinata e p di tipo complesso, le assegnazioni $p:=z$ e $z:=p$ sarebbero errate. Viceversa se le dichiarazioni di tipo fossero:

```
type coordinata= record primo, secondo : real end;  
type complesso= coordinata;
```

z e p, dichiarate come sopra, avrebbero lo stesso tipo, quindi sia $p:=z$ che $z:=p$ sarebbero ammissibili.

Questa equivalenza è stata chiamata *equivalenza per nome*, o meglio equivalenza *per occorrenza*. Con questo approccio infatti le variabili *x* e *y* qui di seguito dichiarate:

```
var x, y : record primo, secondo : real end;
```

hanno lo stesso tipo, mentre se la dichiarazione fosse:

```
var x : record primo, secondo : real end;  
var y : record primo, secondo : real end;
```

x e *y* avrebbero tipo diverso.

La motivazione per adottare l'equivalenza per nome è legata al fatto di permettere al programmatore di fare distinzioni tra tipi più fini della semplice differenza strutturale. Per cui, tornando all'esempio precedente dei tipi *coordinata* e *complesso*, il fatto che il programmatore li abbia creati in due distinte dichiarazioni, significa che egli li vuole tenere distinti. Tuttavia, questa caratteristica del Pascal è estremamente debole, infatti, nello stesso caso, le operazioni di aggiornamento selettivo delle componenti del record:

```
z.primo := p.primo  
z.secondo := p.secondo
```

sono ammissibili, in quanto tutte le variabili che compaiono nelle assegnazioni appena mostrate hanno lo stesso tipo: *real*.

2. CONTROLLO

Nei linguaggi di programmazione le trasformazioni da effettuare sui dati vengono espresse mediante una serie di operazioni. L'ordine di esecuzione di tali operazioni è specificato dal programmatore mediante le primitive di controllo. Il resto di questo capitolo tratterà le primitive o strutture per esprimere il controllo presenti nei linguaggi di programmazione del tipo del Pascal, suddividendole in strutture di base e primitive per la definizione di astrazioni funzionali; verrà poi discusso come queste interagiscono con il trattamento delle variabili. Infine verrà presentata e discussa una particolare tecnica di programmazione: la ricorsione.

I linguaggi del tipo del Pascal vengono di solito detti imperativi in quanto un programma consiste in una serie di ordini o comandi impartiti all'elaboratore. Questi ordini vengono espressi mediante istruzioni. Tra le istruzioni discutia-

mo separatamente: l'istruzione di assegnazione e le istruzioni che riguardano il flusso del controllo nella esecuzione del programma.

2.1. Assegnazione

L'assegnazione è l'istruzione fondamentale su cui si basano i linguaggi imperativi. In alcuni, per esempio il Fortran o il Basic, è indicata come:

$$<\text{variabile}> = <\text{espressione}>$$

Nel linguaggio dei grafi di flusso presentato nell'introduzione essa è rappresentata come:

$$<\text{variabile}> \leftarrow <\text{espressione}>$$

Nel Pascal essa è indicata come:

$$<\text{variabile}> := <\text{espressione}>$$

Si noti come l'uso del simbolo “=” in questo contesto può essere fuorviante, in quanto non si tratta di un predicato di uguaglianza.

Prima di illustrare la semantica della assegnazione, ricordiamo che in un linguaggio come il Pascal, tutti gli identificatori usati in un programma devono essere dichiarati prima dell'uso. Alle variabili, al momento della dichiarazione viene associato un tipo. Discutiamo la semantica delle dichiarazioni nel caso più semplice, quello degli identificatori di variabile elementare, facendo riferimento allo schema funzionale di elaboratore fornito nell'introduzione.

Durante l'esecuzione di un programma, sia il programma che i dati da esso manipolati sono immagazzinati nella memoria. La memoria può essere immaginata organizzata linearmente in locazioni contigue: ciascuna locazione ha un contenuto ed è accessibile attraverso un indirizzo, come esemplificato in fig. 1.

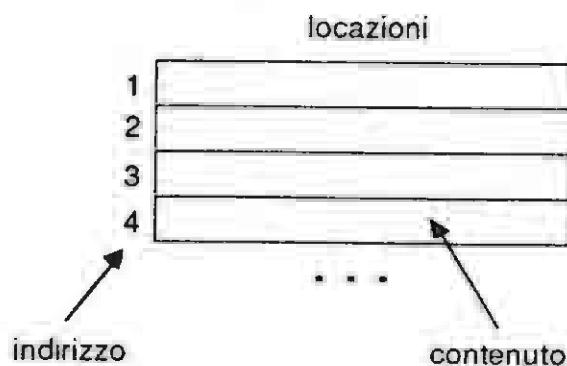
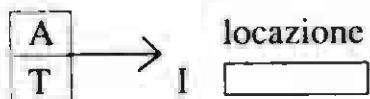


Fig. 1 - Esempio di memoria

Qualora in un programma (o in una unità di programma) sia stata dichiarata una variabile di nome A e di tipo T, l'effetto della dichiarazione è quello di creare un'associazione permanente (cioè immutabile durante tutta l'esecuzione del programma, o dell'unità di programma) tra l'identificatore A, il tipo T ed un indirizzo I relativo ad una locazione di memoria. Rappresentiamo questa associazione con il diagramma:



Mentre l'associazione tra l'identificatore, il tipo e l'indirizzo resta fissa, il contenuto della locazione può cambiare durante l'esecuzione del programma. Il cambiamento può avvenire in vari modi, in particolare attraverso un'istruzione di assegnazione.

Nel contesto della dichiarazione appena vista, la semantica di una istruzione di assegnazione del tipo

$A := e$

è data dall'esecuzione in sequenza delle seguenti operazioni:

- valuta l'espressione e;
- deposita tale valore, cancellando il precedente, nella locazione di memoria il cui indirizzo è associato ad A, ammesso che tale valore appartenga al dominio del tipo di dato della variabile A, altrimenti produci un errore.

Per cui l'istruzione di assegnazione esegue un aggiornamento del contenuto di una locazione di memoria. Dopo l'esecuzione dell'istruzione di assegnazione $A := e$ avremo:



Si noti quindi che in un'istruzione di assegnazione del tipo:

$A := B + C$

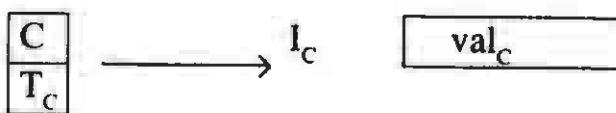
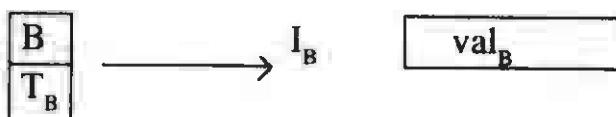
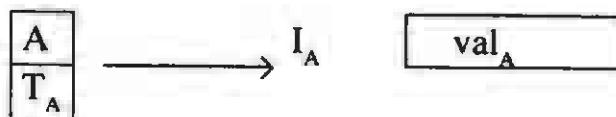
il ruolo delle variabili che compaiono a sinistra e a destra del segno “ $:$ =” è molto diverso. L'effetto di questa istruzione può essere così descritto:

- preleva il valore contenuto della locazione di memoria il cui indirizzo è associato a B;
- preleva il valore contenuto della locazione di memoria il cui indirizzo è associato a C;
- somma tali valori;

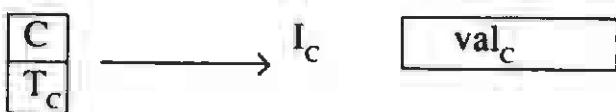
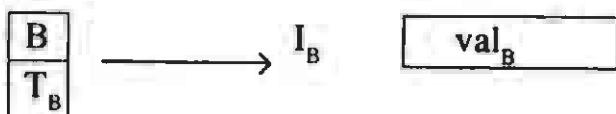
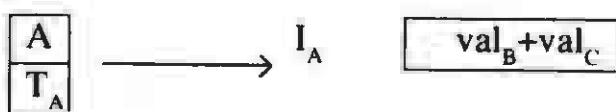
- memorizza il valore così ottenuto nella locazione di memoria il cui indirizzo è associato ad A, previo controllo di compatibilità sul tipo.

La situazione, prima e dopo la valutazione dell'istruzione appena illustrata è riportata graficamente qui di seguito:

Prima:



Dopo:



Questa semantica rende quindi perfettamente significativa un'istruzione quale:

somma := somma + numero

come nell'esempio visto nell'introduzione, in quanto le due occorrenze dell'identificatore *somma* hanno un ruolo completamente diverso: *somma* a sinistra sta ad indicare un indirizzo di memoria in cui memorizzare, *somma* a destra sta ad indicare un indirizzo di memoria da cui prelevare un valore.

Si noti come la semantica di una istruzione di lettura sia del tutto analoga a quella di un'assegnazione: l'effetto della lettura da input di una variabile A è quello di 'assegnare' ad A il valore che si trova sull'organo di ingresso. Per cui

`read(A)` va interpretato come `A := i`, dove `i` è il valore che si trova sull'unità di ingresso.

2.2. Strutture di controllo di base

Il controllo dell'esecuzione di un programma in linguaggi di tipo imperativo viene espresso mediante istruzioni. Le istruzioni di un linguaggio ne caratterizzano la potenza espressiva: il Pascal è un linguaggio completo, nel senso che esso permette di esprimere tutte le funzioni calcolabili (vedere cap. 7) e che si presta particolarmente bene per la costruzione di programmi strutturati.

Ricordiamo infatti che il Pascal offre come strutture di controllo di base quelle relative alla sequenzializzazione (`begin...;...end`), all'iterazione definita (`for`) ed indefinita (`while` e `repeat`) ed alla selezione condizionale (`if...then...`; `if...then...else...`; `case...of...`). A queste si aggiunge una istruzione di alterazione del flusso del controllo (`goto...`).

Per la definizione della sintassi e della semantica informale di queste primitive si rimanda ad un testo introduttivo al Pascal, per il loro uso nella costruzione di programmi secondo la metodologia della programmazione strutturata si rimanda al cap. 4. Qui di seguito concentriamo invece l'attenzione sulle astrazioni funzionali che il Pascal permette e quindi sulle nozioni di procedura e funzione, discutendo in particolare il trattamento delle variabili, ed alcuni aspetti legati alla loro implementazione.

2.3. Astrazioni funzionali: procedure e funzioni

Tutti i linguaggi di programmazione ad alto livello permettono l'uso di astrazioni funzionali. Esse consistono nel creare delle unità di programma (in alcuni contesti le unità di programma vengono chiamate sottoprogrammi) dando un nome ad un gruppo di istruzioni e stabilendo le modalità di comunicazione tra l'unità di programma creata ed il resto del programma in cui essa si inserisce. Il programmatore userà il nome dell'unità da lui creata ogni volta desideri che nel programma venga eseguito il blocco di istruzioni che costituiscono il corpo di detta unità. Si parla in questo caso di richiesta di attivazione (o chiamata) dell'unità di programma.

All'atto dell'attivazione di un'unità di programma viene sospesa l'esecuzione del programma (o dell'unità) che contiene la richiesta di attivazione: il

controllo passa all'unità attivata e, all'atto del completamento della sua esecuzione, l'attivazione termina ed il controllo ritorna al programma (o unità) chiamante. Le modalità che governano questo meccanismo verranno meglio discusse nel par. 2.7. In questa sezione invece facciamo notare come il Pascal permetta di definire due tipi di astrazione funzionale:

- **astrazione mediante procedura (procedure)**: è un'astrazione della nozione di istruzione. Una procedura è di fatto un'istruzione complessa e può essere attivata dal programmatore ovunque può essere usata un'istruzione;
- **astrazione mediante funzione (function)**: è un'astrazione della nozione di operatore (funzione o predicato) su un tipo di dato (primitivo o definito da utente). Una funzione può essere attivata durante la valutazione di una qualsiasi espressione.

Le procedure e le funzioni sono del tutto analoghe per quel che riguarda la loro forma sintattica e anche per quanto riguarda le regole che governano la loro valutazione. La principale differenza sta nel loro uso: le procedure hanno lo scopo di modificare il contenuto di locazioni di memoria; lo scopo di una funzione invece è quello di fornire un valore, che contribuisce al valore dell'espressione entro cui la funzione viene attivata.

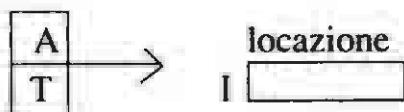
Alcuni degli operatori primitivi in Pascal sono sovraccarichi (vedere par. 1.3), per esempio l'operatore +, che è definito per diversi tipi di dato. Il Pascal, attraverso la definizione di funzioni, permette all'utente di definire nuovi operatori. Si noti però che il Pascal non permette sovraccarico per gli operatori definiti dall'utente come **function**, cioè una funzione non può essere definita all'interno della stessa unità di programma per diversi tipi di argomenti.

2.4. Dichiarazioni e campo d'azione degli identificatori

Ricordiamo che in Pascal, come in molti altri linguaggi di programmazione, tutti gli identificatori usati in un programma debbono essere dichiarati esplicitamente prima del loro uso. Per identificatori si intendono i nomi che il programmatore usa per indicare le costanti, le variabili, i tipi e le unità di programma, cioè le procedure e funzioni, da lui definiti. Per quanto riguarda la sintassi delle dichiarazioni e le regole che le governano, quali quelle relative alla posizione delle dichiarazioni nel programma ed all'ordine in cui esse debbono comparire, si rimanda ad un testo sul Pascal.

Il significato delle dichiarazioni è quello di associare, durante l'attivazione dell'unità di programma, varie informazioni agli identificatori, per esempio sul

tipo di una variabile o sul frammento di programma associato ad un nome di procedura, ecc. Queste associazioni costituiscono l'ambiente, sono permanenti ed il loro *ciclo di vita* è la durata dell'attivazione della unità di programma in cui esse compaiono. In altre parole, l'effetto di una dichiarazione permane per tutto il tempo di attivazione dell'unità di programma in cui tale dichiarazione si trova. Per esempio, se una variabile A è stata dichiarata di tipo T all'interno di una procedura P, tale variabile avrà un ciclo di vita durante ogni attivazione di P, cioè, per ogni attivazione di P viene creata una associazione:



Alla fine dell'attivazione di P, tale associazione per A viene distrutta, e, se P viene attivata di nuovo, ne verrà creata una nuova; per cui non c'è alcuna correlazione tra i valori che A assume durante le varie attivazioni di P.

Correlata con la nozione di ciclo di vita di un identificatore, ma non coincidente con essa, è la nozione di *campo d'azione* di una dichiarazione. Il campo d'azione di una dichiarazione è l'insieme delle unità di programma in cui tale dichiarazione ha effetto, cioè l'insieme delle unità di programma che possono utilizzare l'identificatore così dichiarato.

La definizione del Pascal dice che il campo d'azione della dichiarazione di un identificatore è costituita dalle seguenti unità: l'unità di programma in cui essa compare e tutte le unità di programma in questa inclusa, a meno che tale identificatore non sia di nuovo dichiarato in un'unità di programma più interna. In tal caso ha effetto la nuova dichiarazione e si dice che essa maschera la precedente.

Si dice anche che un identificatore A è visibile in una unità di programma se questa è nel campo d'azione della dichiarazione di A. Per esempio, se P, Q1, Q2, R1 ed R2 sono unità di programma dichiarate come in fig 2, si avrà che tutti gli identificatori dichiarati in P sono visibili in Q1, Q2, R1 ed R2, gli identificatori dichiarati in Q2 sono visibili in R1 ed R2 (a meno che una nuova dichiarazione non li abbia mascherati).

Gli identificatori visibili si distinguono in locali e non locali. Si dice che un identificatore è locale ad una unità di programma se è dichiarato al suo interno, non locale o globale se è dichiarato in una unità di programma che la contiene e non è ridichiarato al suo interno (in alcuni testi il nome globale è riservato a quegli identificatori che sono dichiarati solo nel programma principale).

Per i linguaggi che si comportano come il Pascal si dice che il campo d'azione

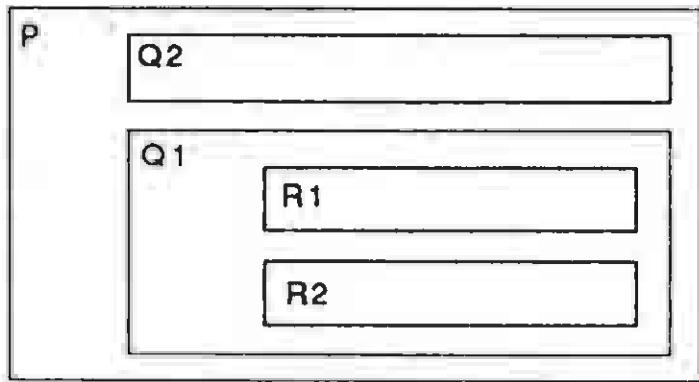


Fig. 2 - Esempio di nidificazione di dichiarazioni

```

program scope (...);
  var x : integer;
  procedure P1;
    begin
      writeln (x)
    end;
  procedure P2;
    var x : real;
    begin
      x := 2.13;
      P1
    end;
  begin
    x := 1;
    P2
  end.

```

Fig. 3 - Il programma scope

delle dichiarazioni è determinato staticamente. La strategia alternativa a questa, ed adottata da altri linguaggi, è quella del campo d'azione determinato dinamicamente. In quest'ultimo caso il legame tra un identificatore e la sua dichiarazione viene fatto dinamicamente, cioè considerando la sequenza delle attivazioni delle unità di programma.

Per apprezzare la differenza tra le due strategie per la determinazione del campo d'azione, si consideri il programma Pascal mostrato in fig 3.

L'effetto dell'esecuzione del programma *scope* è quello di stampare il valore della variabile *x*. Poiché il Pascal determina il campo d'azione staticamente, la

variabile *x* il cui valore viene stampato è quella dichiarata intera e posta ad 1. Per cui il programma *scope* stampa il numero 1. Ben diverso sarebbe il risultato se il Pascal determinasse il campo d'azione delle dichiarazioni dinamicamente. Il programma *scope* stamperebbe il valore di *x*, ma sarebbe l'*x* dichiarato all'interno dell'ultima attivazione di procedura che contiene una dichiarazione per tale identificatore. Quindi la variabile di cui sarebbe stampato il valore sarebbe quella dichiarata reale dentro P2: il valore stampato sarebbe 2.13.

Nel par. 2.7 vedremo come viene realizzato in Pascal il meccanismo che permette di determinare staticamente il campo d'azione di una dichiarazione.

2.5. Tecniche di legame dei parametri

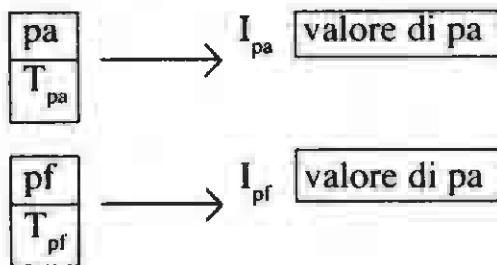
Nei linguaggi di programmazione la comunicazione di informazione tra una unità di programma e quella che l'ha attivata avviene attraverso le variabili non locali, ma soprattutto attraverso i parametri. Le procedure e funzioni vengono definite su parametri formali e attivate su parametri attuali. Per legame dei parametri si intende l'associazione tra parametri attuali e parametri formali che avviene al momento dell'attivazione di un'unità di programma. In questo paragrafo illustriamo e confrontiamo i meccanismi più comuni di legame dei parametri. Essi sono:

1. legame per valore;
2. legame per riferimento;
3. legame per valore-risultato.

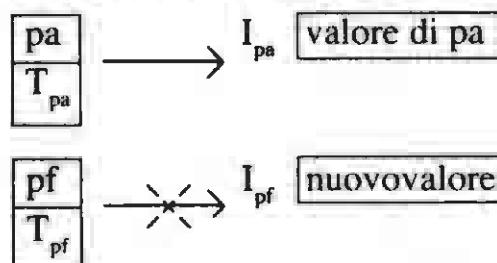
I primi due sono quelli presenti in Pascal (parametri valore e parametri variabile, rispettivamente), il terzo, utilizzato in altri linguaggi, viene qui presentato essenzialmente con lo scopo di illustrare meglio il comportamento dei legami adottati dal Pascal.

Supponiamo che una procedura di nome *P* sia stata dichiarata con un parametro formale *pf* e che venga attivata con un parametro attuale *pa*. Legare *pa* a *pf* per valore significa, al momento dell'attivazione di *P*, valutare *pa* (che è in generale un'espressione), creare una locazione di indirizzo *pf* e memorizzare in tale locazione il valore ottenuto valutando *pa*. Il parametro formale *pf* si comporta quindi come una variabile locale a *P*, creata al momento della sua attivazione ed automaticamente inizializzata al valore di *pa*. Alla fine dell'esecuzione di *P*, la memoria riservata per *pf* viene rilasciata ed il suo valore si perde; il valore delle variabili che compaiono nell'espressione *pa* non viene quindi alterato dall'esecuzione di *P*.

Si noti che pa può essere una variabile. In questo caso la situazione al momento dell'attivazione di P può essere così schematizzata:

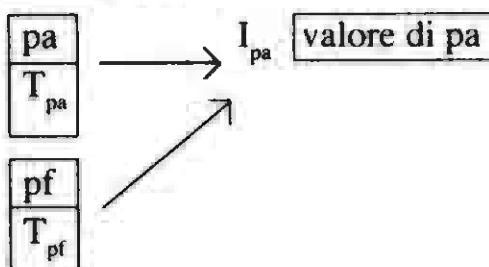


Alla fine dell'attivazione di P abbiamo:

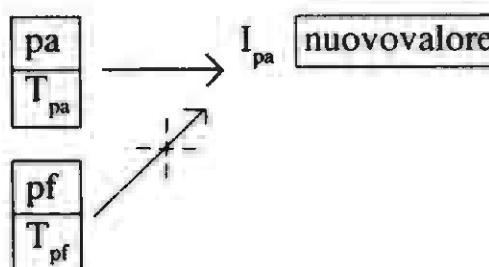


Legare pa a pf per riferimento (parametri variabile in Pascal) significa identificare due riferimenti in memoria. Innanzitutto, affinché sia possibile legare pa a pf per riferimento, pa deve essere una variabile. Il riferimento di memoria di pa (cioè l'indirizzo) viene calcolato al momento dell'attivazione di P e pf viene creato come riferimento alla stessa locazione di memoria. Graficamente possiamo rappresentare questa situazione come:

- all'attivazione di P:



- alla fine dell'attivazione di P:

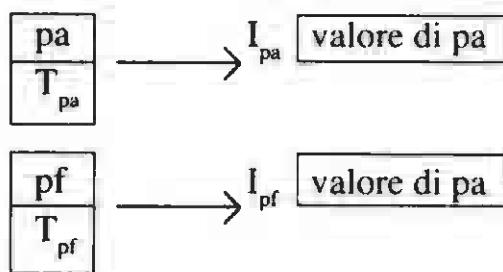


Risulta quindi evidente come tutte le modifiche eventualmente fatte a pf durante l'attivazione di P sono anche modifiche fatte a pa e per questo esse sopravvivono alla attivazione di P ; pertanto pa può rimanere modificato dall'attivazione di P .

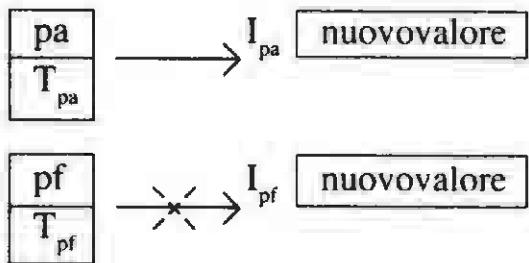
Si noti che nel legame per riferimento, se il parametro attuale è una variabile con indice, il valore dell'indice viene calcolato al momento del legame tra il parametro formale e quello attuale; per cui, se per esempio il parametro attuale è $mat[exp]$, dove mat è una variabile di tipo **array**, ed al momento del legame per riferimento tra parametro formale ed attuale il valore di exp è 3, l'indirizzo individuato in memoria è quello di $mat[3]$. Quindi se durante l'attivazione dell'unità di programma il valore di exp viene modificato, questa modifica non influenza il legame tra il parametro formale e $mat[3]$.

Dalle considerazioni precedenti si vede come i parametri legati per valore consentano di importare valori nell'unità attivata (e perciò sono talvolta detti parametri di input), quelli per riferimento consentano di esportare valori all'unità attivante (e perciò sono talvolta detti parametri di output).

Un terzo metodo di legame dei parametri è il cosiddetto legame per valore-risultato. Esso consiste nel calcolare il valore di pa al momento dell'attivazione di P , e memorizzarlo in una nuova locazione di indirizzo pf . Al momento in cui l'esecuzione di P è terminata il valore contenuto in pf viene trasferito in pa e la memoria riservata per pf viene rilasciata. La situazione al momento dell'attivazione di P può essere così schematizzata:



Durante l'attivazione di P tutte le modifiche fatte a pf non influenzano pa ; alla fine dell'attivazione di P si avrà:



```

program legaparametri (...,...);
  var n : integer;
  procedure P (? x : integer);
    begin
      x := x+1;
      writeln (n);           {1}
      writeln (x)            {2}
    end;
    begin
      n := 3;
      P (n);
      writeln (n)           {3}
    end.

```

Fig. 4 - Il programma legaparametri

Questa tecnica, che è un misto tra il legame per riferimento e quello per valore, ha una semantica diversa da entrambi, come risulta dall'esempio mostrato in fig. 4.

Nell'esempio il parametro x della procedura P è stato marcato con "?" in quanto non si vuole specificare se è legato per valore, per riferimento o per valore-risultato. L'effetto dell'esecuzione del programma sarà quello di far tre stampe, rispettivamente dei valori di n , x durante l'attivazione di P ed ancora di n dopo tale attivazione.

Se x è legato per valore i tre valori stampati sono:

{1}	3
{2}	4
{3}	3

se x è legato per riferimento i tre valori stampati sono:

{1}	4
{2}	4
{3}	4

Se il Pascal permettesse per x il legame per valore-risultato, i tre valori stampati sarebbero:

{1}	3
{2}	4
{3}	4.

```

program sideeffect (...,...);
  var b: integer;
  function f(var a: integer): integer;
    begin
      a := 2*a;
      f := a
    end;
    begin
      b := 1;
      writeln (2*f(b));      {1}
      b := 1;
      writeln (f(b) + f(b)); {2}
    end.

```

Fig. 5 - Esempio di funzione con effetto collaterale

2.6. Effetti collaterali in procedure e funzioni

In generale si chiama effetto collaterale (*side effect*) provocato dall'attivazione di una unità di programma una qualunque modifica delle variabili non locali a tale attivazione. Gli effetti collaterali, accettabili nelle procedure, sono invece da evitare nelle funzioni.

Infatti, l'attivazione di una funzione dovrebbe soltanto fornire un valore come risultato e non dovrebbe modificare il contenuto di nessuna locazione di memoria. Il Pascal non fornisce gli strumenti linguistici per evitare effetti collaterali nelle funzioni, il compito è lasciato al programmatore.

I parametri legati per riferimento e le variabili non locali, in quanto possono esportare valori dall'attivazione di unità di programma, vanno usati con cautela perché possono produrre effetti collaterali indesiderati. L'esempio mostrato in fig. 5 mette in evidenza come un effetto collaterale in una funzione, in particolare la modifica di un parametro legato per riferimento, possa far sì che questa fornisca valori diversi in attivazioni diverse, pur se sullo stesso parametro attuale.

Le funzione *f*, attivata sul parametro attuale *b*, ne calcola il doppio. Per cui l'istruzione {1} stampa il quadruplo di *b*, quindi il valore 4, in quanto *b* è posto ad 1. Anche l'istruzione {2} dovrebbe stampare il quadruplo di 1, quindi il valore 4, ma a causa dell'effetto collaterale che si ha durante la prima attivazione di *f* in *f(b)+f(b)*, la seconda attivazione di *f* fornisce il valore 4, per cui l'istruzione {2} stampa il valore 6.

```
program alias (...,...);
  var a, c : integer;
  procedure P (var b : integer);
    begin
      c := c+1;
      ...
      b := ...
      ...
    end;
  begin
    c:= 0;
    ...
    P (a);
    ...
    P (c);
    ...
  end.
```

Fig. 6 - Esempio di programma con alias

A ulteriore conferma dei problemi che possono sorgere nell'interazione tra una unità ed il resto del programma si consideri il programma mostrato in fig.6.

In tale programma la variabile *c* è intesa contare il numero di attivazioni della procedura *P*. Tuttavia durante l'attivazione di *P(c)*, *c* è un sinonimo per *b*, in quanto entrambi identificano lo stesso indirizzo in memoria; pertanto l'uso inteso per *c*, viene violato in quanto tutte le modifiche che vengono fatte a *b* durante tale attivazione si riflettono anche su *c*.

L'identificazione delle locazioni di memoria associate a due variabili diverse prende il nome di *alias*; come l'esempio mostra, questo fenomeno può produrre effetti indesiderati.

2.7. Implementazione di procedure e funzioni

In questo paragrafo descriviamo brevemente la gestione delle unità di programma nei linguaggi del tipo del Pascal. In questi linguaggi le attivazioni sono gestite seguendo una disciplina a pila (*stack*). Una pila (vedere cap. 3) è un multiinsieme ordinato di elementi cui si accede mediante operazioni di inserimento e di estrazione con il vincolo che l'ultimo elemento inserito è il primo estratto (vedere fig. 7).

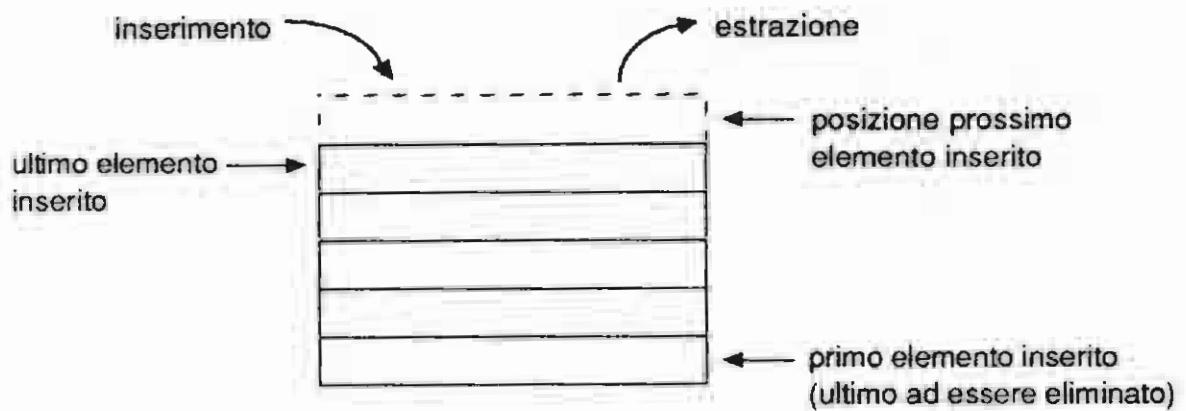


Fig. 7- Esempio di pila

Ad ogni attivazione di un'unità di programma viene creato un record di attivazione e messo in cima alla pila. Un record di attivazione rappresenta tutte le informazioni relative ad una specifica attivazione, in particolare:

1. nome della unità di programma attivata e riferimento ad una zona di memoria dove è memorizzato il corpo di istruzioni da eseguire;
2. riferimento di catena statica (il cui significato verrà illustrato nel seguito);
3. punto di ritorno, cioè informazione relativa all'istruzione cui tornare quando l'attivazione è finita;
4. parametri formali e loro legame con i corrispondenti attuali;
5. variabili locali con riferimento alla memoria per esse allocata.

Un esempio di record di attivazione è mostrato in fig. 8. Quando l'attivazione di un'unità di programma P è finita, il record è tolto dalla pila, viene rilasciata la memoria allocata per le variabili locali di P e si perde traccia dei legami tra

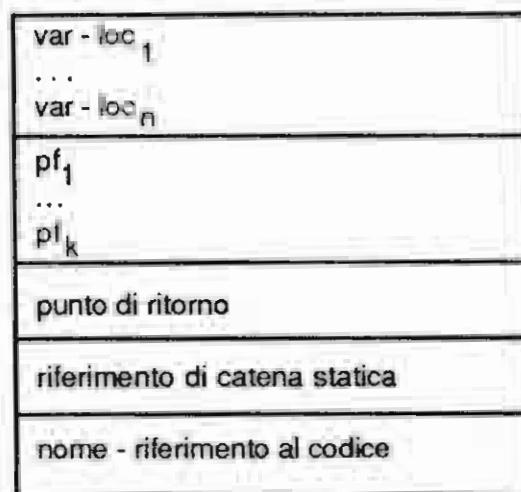


Fig. 8 - Struttura del record di attivazione

i parametri formali e gli attuali. L'esecuzione prosegue dall'istruzione indicata al punto di ritorno di P, nell'ambiente determinato dal record di attivazione che ora è in cima alla pila.

Se P ha attivato Q, che ha attivato R, avremo una situazione del tipo di quella mostrata in fig. 9.

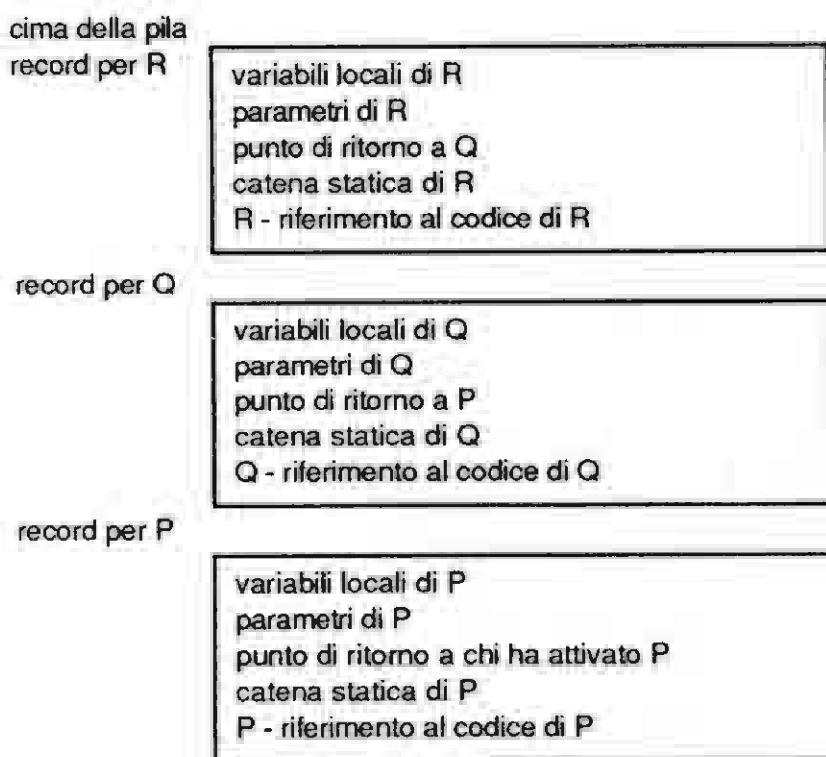


Fig. 9 - Pila di record di attivazione per P, Q e R

La catena realizzata attraverso la sequenzializzazione delle attivazioni nella pila si chiama catena dinamica: essa rappresenta la storia delle attivazioni delle unità di programma.

La catena statica invece fa riferimento alla gerarchia creata al momento della dichiarazione e indica dove, nella pila, vanno cercati i riferimenti per le variabili non locali.

Essi vanno cercati come segue: il riferimento per una variabile non locale di nome A va cercato nel record relativo all'ultima attivazione dell'unità di programma che contiene la dichiarazione di A a livello gerarchicamente superiore nella nidificazione determinata dal testo del programma. Se nell'esempio appena mostrato la nidificazione delle dichiarazioni è quella di fig. 10:

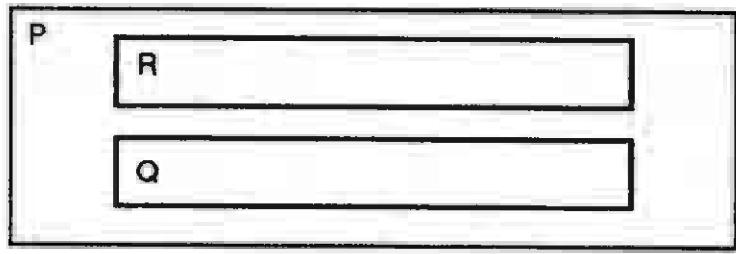


Fig. 10 - Nidificazione di dichiarazioni

il riferimento di catena statica di R è il record di attivazione di P e non quello di Q. Si avrà quindi la situazione mostrata in fig. 11 e schematizzata in fig. 12, dove la catena dinamica è stata indicata con frecce continue e quella statica con frecce tratteggiate.

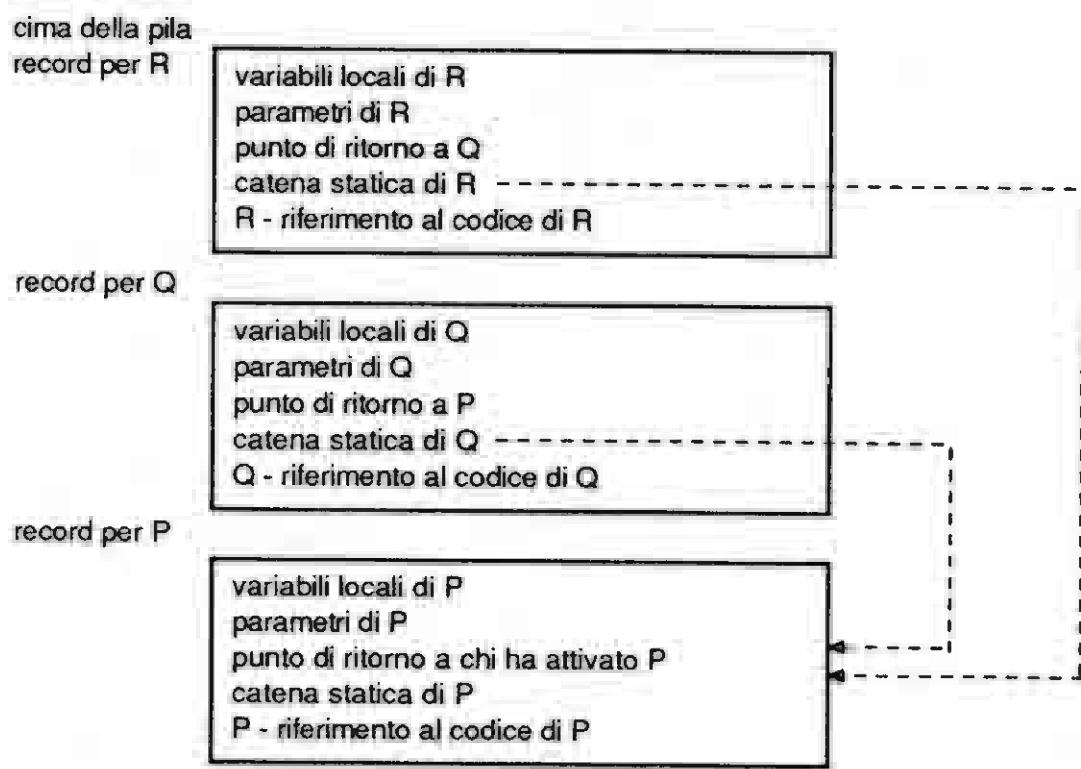


Fig. 11- Esempio di catena statica

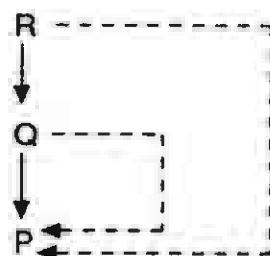


Fig. 12 - Catena statica e dinamica per P, Q ed R

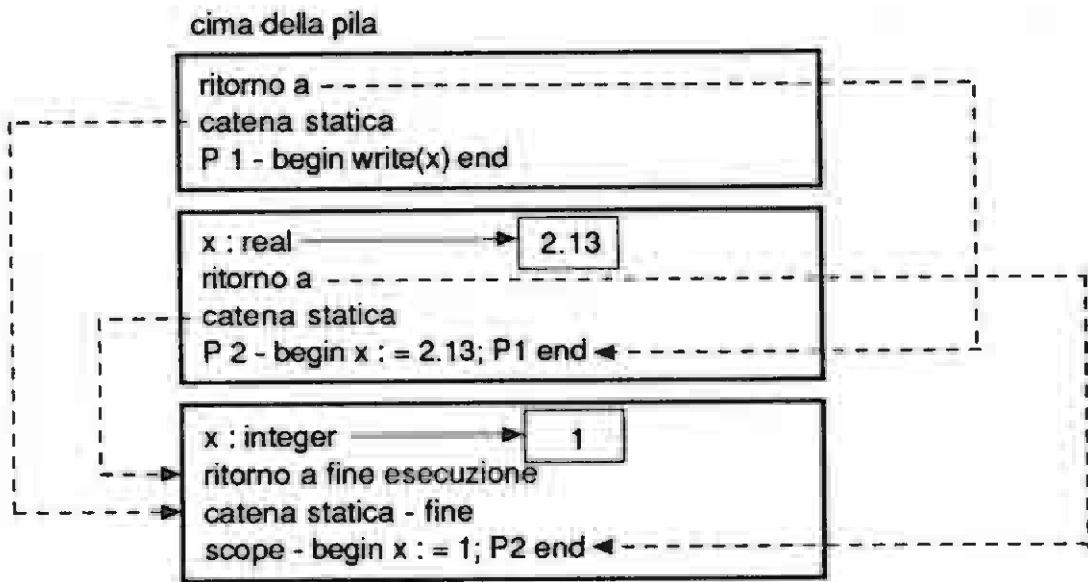


Fig. 13 - Catena statica e dinamica per il programma scope

Ovviamente, se per le variabili non locali di R non viene trovato un riferimento in P questo va cercato proseguendo nella catena statica di P stesso.

Come ulteriore esempio si consideri la pila delle attivazioni del programma *scope*, mostrato in fig. 3, nel momento in cui è in esecuzione la procedura *P1*. Essa è presentata in fig. 13.

Nei record di attivazione non è presente alcuna informazione sui parametri, in quanto assenti. Si noti inoltre che, piuttosto che un riferimento alle istruzioni da eseguire durante una attivazione abbiamo scritto le istruzioni stesse con una freccia nel punto in cui tornerà il controllo quando l'attivazione sarà riesumata. Siccome il riferimento di catena statica relativo a *P1* è il record relativo all'attivazione di *scope* (come emerge dalla nidificazione delle dichiarazioni nel testo del programma) quando durante l'attivazione di *P1* viene eseguita la stampa della variabile non locale *x* il valore stampato è quello che si trova nel record di attivazione di *scope*, e non in quello di *P2*, anche se *P2* ha una variabile locale di nome *x*.

2.8. La ricorsione

Una funzione matematica è definita ricorsivamente quando nella sua definizione compare un riferimento a se stessa. Ad esempio la funzione fattoriale sugli interi non negativi:

$$f(n) = n!$$

è definita ricorsivamente come segue:

$$f(n) = \begin{cases} 1 & \text{se } n=0 \\ n*f(n-1) & \text{se } n>0 \end{cases}$$

Una funzione che opera su dati di un tipo induttivo (vedere il par. 4.1 dell'appendice) viene definita ricorsivamente specificando come tale funzione si comporta sugli elementi della base della costruzione (caso base) e come si comporta nel generico passo della costruzione induttiva (caso induttivo). Nel caso della funzione fattoriale il caso ' $f(n)=1$ se $n=0$ ' è il caso base, mentre il caso ' $f(n)=n*f(n-1)$ se $n>0$ ' è il caso induttivo.

Informalmente, la definizione ricorsiva della funzione fattoriale ci dice che: il calcolo di f sul numero n si riconduce al calcolo di f sul predecessore, prosegue nel passaggio da un numero al suo predecessore fin quando non arriva al calcolo di f su 0, dove la ricorsione termina.

La ricorsione è una tecnica di programmazione che si rivela particolarmente efficace per esprimere sinteticamente operazioni su tipi di dato ricorsivi, cioè tipi di dato i cui domini sono definiti in maniera induttiva. Un uso estensivo della ricorsione come tecnica di programmazione verrà fatto nel cap. 3, dove verranno presentate molte funzioni e procedure ricorsive per la manipolazione di strutture dati definite induttivamente, quali liste, pile, code ed alberi.

Facendo riferimento al modello di attivazione di procedure e funzioni presentato nel paragrafo precedente si comprende come il Pascal gestisca la ricorsione come una normale attivazione di procedura o funzione: infatti quando nel corpo di una funzione o procedura di nome F si incontra una richiesta di attivazione di F , questa viene gestita mediante la normale disciplina a pila. A titolo di esempio mostriamo in fig. 14 una funzione Pascal per il calcolo del fattoriale ed in fig. 15 la pila delle attivazioni di $fatt(n)$ per la valutazione di $fatt(2)$.

Si noti che nei record di attivazione in fig. 15 abbiamo omesso la parte

```

function fatt (n: interononneg): interononneg;
begin
  if n=0
    then fatt:=1
    else fatt:=n*fatt (n-1)
  end;

```

Fig. 14 - Funzione Pascal ricorsiva per il calcolo del fattoriale

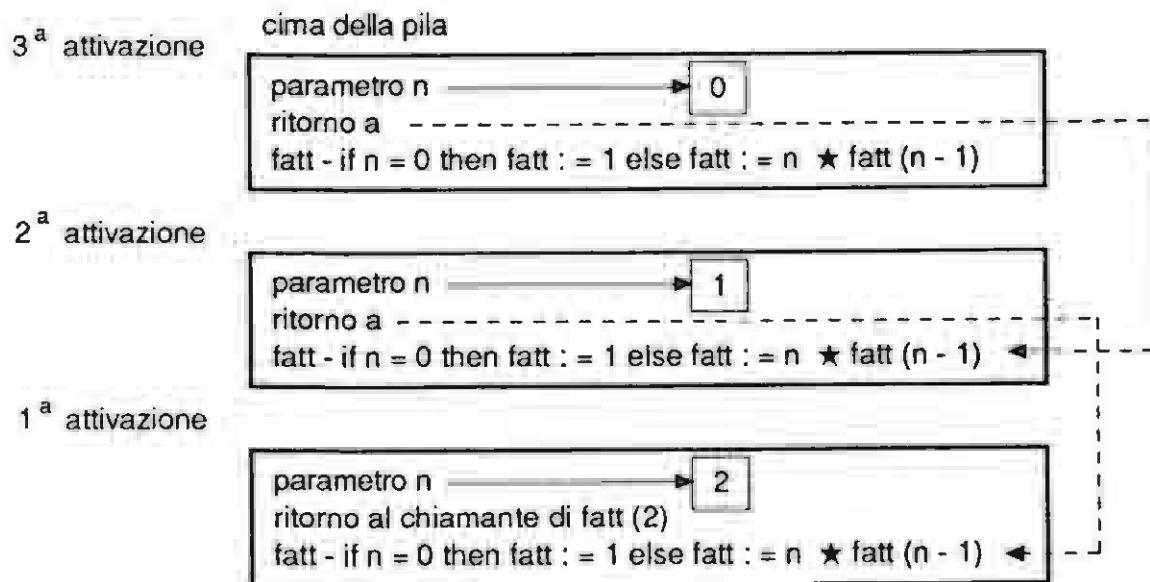
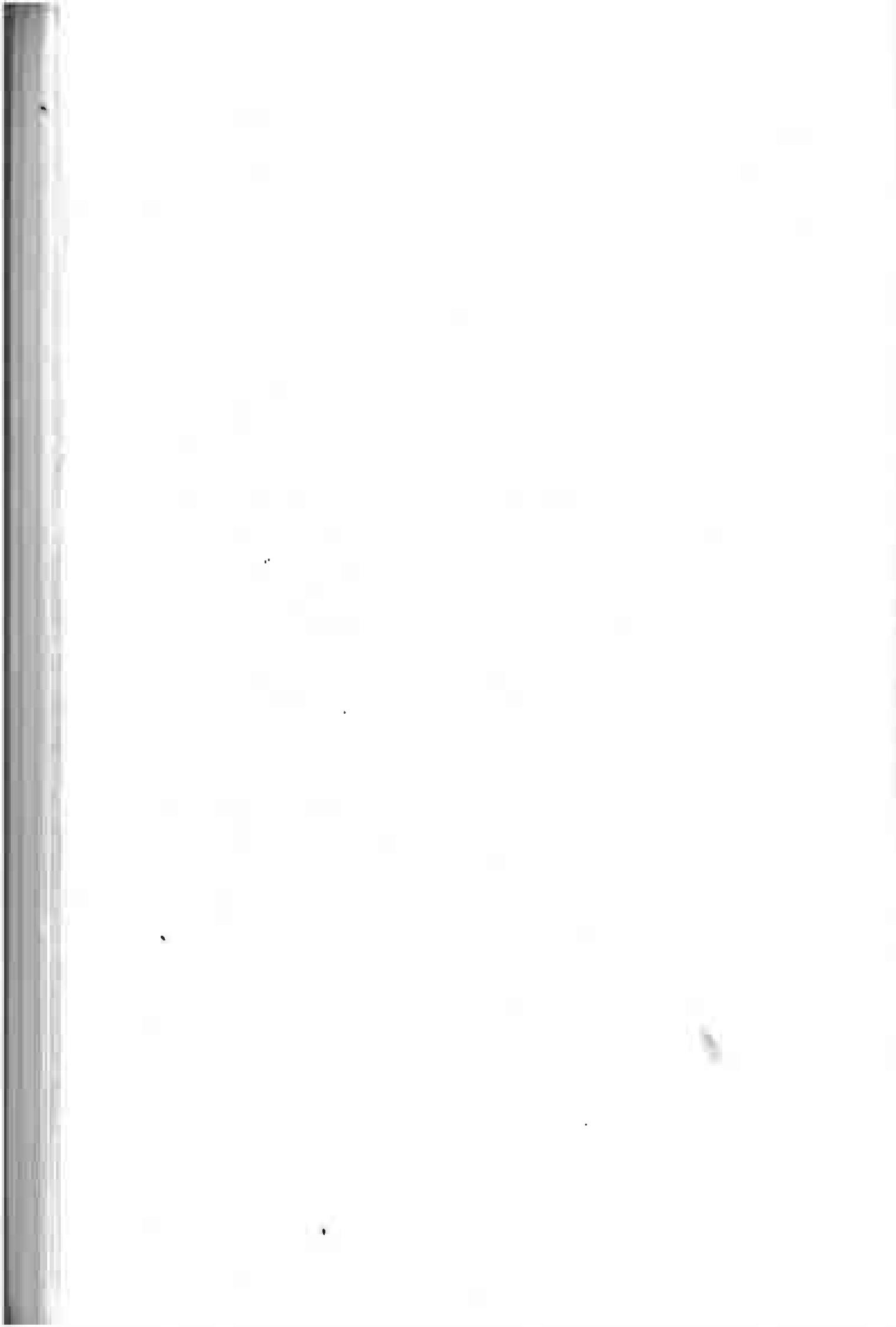


Fig. 15 - Pila di attivazioni per la valutazione di fatt (2)

riguardante le variabili locali, in quanto assenti, e quella riguardante la catena statica, in quanto anche le variabili non locali sono assenti. Si noti inoltre che, piuttosto che un puntatore al codice da eseguire durante una attivazione abbiamo scritto il codice stesso con una freccia nel punto in cui si tornerà quando l'attivazione sarà riesumata. In fig. 15 sono mostrate tre attivazioni: quella di fatt(2), quella di fatt(1) ed infine quella di fatt(0). Siccome il predicato $n=0$ è soddisfatto, durante l'attivazione di fatt(0), la ricorsione termina ed i record vengono via via tolti dalla pila ritornando i seguenti valori: l'attivazione di fatt(0) ritorna il valore 1; l'attivazione di fatt(1) ritorna il valore $1+1$, cioè 1; l'attivazione di fatt(2) ritorna il valore $2*1$, cioè 2.



Parte II

Strutture di dati

3. Strutture di dati

3.

Strutture di dati

Scopo di questo capitolo è approfondire la nozione di tipo o struttura di dato ed illustrare l'uso dei tipi di dato nel progetto dei programmi. Discuteremo, dapprima, il concetto di tipo astratto di dato e le problematiche relative alla rappresentazione di un tipo astratto. Successivamente studieremo diversi tipi di dato ampiamente utilizzati nella programmazione: matrici, liste, insiemi, pile, code, alberi, grafi e tavole. Per ognuno di essi discuteremo diversi metodi di rappresentazione, illustrandone i vantaggi e gli svantaggi.

1. TIPI ASTRATTI DI DATO

Quando si affrontano problemi complessi sorge l'esigenza di condurre il progetto di un programma in modo articolato, distinguendo tra due fasi fondamentali:

- la specifica dell'algoritmo;
- la realizzazione dell'algoritmo in termini di un programma scritto in un linguaggio di programmazione.

La prima fase coinvolge due aspetti complementari: la specifica dei dati che l'algoritmo deve manipolare e la specifica delle operazioni da eseguire per realizzare l'algoritmo. Corrispondentemente, la seconda fase richiede un'attività di traduzione che riguarda sia i dati che le operazioni. Nel cap. 4 approfondiremo la tematica generale del progetto dei programmi, analizzando i metodi e le tecniche per condurre in modo efficace le due fasi suddette. In questo capitolo ci concentriamo su uno dei due aspetti del progetto dei programmi: la specifica dei dati e la traduzione di tale specifica in un linguaggio di programmazione.



Lo strumento che si utilizza in fase di specifica per esprimere i dati in maniera indipendente dai vincoli imposti dal linguaggio di programmazione è il tipo astratto di dato.

Come abbiamo visto nel cap. 2, un *tipo astratto di dato* (o semplicemente tipo astratto) è un oggetto matematico costituito da tre componenti: un insieme di valori, detto il dominio del tipo, un insieme di operazioni, che si applicano a valori del dominio o che hanno come risultato valori del dominio, ed un insieme di costanti, che denotano valori del dominio.

Esempio. Possiamo definire il tipo astratto “booleano” come costituito da:

- l’insieme *{true, false}*;
- gli operatori “*or*”, “*and*”, e “*not*”;
- le costanti *true* e *false*, che denotano rispettivamente i valori *true* e *false* del dominio.

Si noti che in questo caso abbiamo definito tante costanti quanti sono i valori del dominio. Al contrario, nell’esempio del cap. 2 relativo ai numeri naturali il dominio è infinito, e sono definite solo 2 costanti.

Vogliamo sottolineare che tutte e tre le componenti di un tipo astratto sono indipendenti dalla rappresentazione e dall’uso del tipo stesso nei linguaggi di programmazione. Questo è coerente con l’impostazione metodologica a cui accennavamo prima: la fase di specifica ha lo scopo di formulare la struttura dei dati di interesse in modo astratto, privilegiando gli aspetti di chiarezza e naturalezza di espressione, rispetto ai vincoli imposti dai linguaggi e alla efficienza degli algoritmi.

Nel par. 1.1 approfondiremo le problematiche relative alla specifica dei tipi astratti, riformulando in maniera più precisa anche la definizione data sopra. Prima, però, è opportuno svolgere alcune considerazioni sulla terminologia che abbiamo introdotto, confrontandola con quella usata in altri testi.

Noi parliamo indifferentemente di tipo o struttura di dati. Alcuni testi però, usano il termine struttura per indicare un tipo il cui dominio è composito, cioè costituito da valori più elementari, mentre riservano il termine stesso di tipo per indicare tipi il cui dominio è elementare, cioè costituito da elementi atomici, non decomponibili. Secondo questa terminologia, l’*integer* del Pascal è un tipo di dato, mentre, ad esempio, il *record* è una struttura.

Una distinzione che invece noi consideriamo è quella tra tipo astratto e tipo concreto. Il termine *tipo concreto* viene usato per riferirsi alla definizione e all’uso di un tipo di dato in un linguaggio di programmazione. Per caratteriz-

zare un tipo concreto è necessario specificare non solo le proprietà astratte, cioè dominio, operazioni e costanti, ma anche i vincoli imposti dal linguaggio sulla definizione e l'uso del tipo stesso. Ad esempio, il tipo **array** del Pascal è un tipo concreto: per definirlo in modo completo è necessario specificare il modo in cui le variabili di tipo **array** sono dichiarate in un programma, i vincoli sul tipo degli indici e sulle dimensioni, i metodi per accedere alle varie componenti, e così via.

1.1. La specifica dei tipi astratti

Per i tipi concreti di un linguaggio di programmazione le tre componenti che caratterizzano il tipo (dominio, operazioni e costanti) sono implicitamente definite nell'ambito del linguaggio, in termini dei meccanismi di definizione del tipo e della semantica degli operatori applicabili ai suoi valori. Anche per i tipi astratti è necessario fornire una definizione comprensibile, completa e non ambigua.

Esempio. Vogliamo definire il tipo astratto *insieme*, cioè il tipo di dato che consente di rappresentare collezioni di elementi di un tipo *V*.

Il tipo di dato *insieme di elementi di tipo V*, o semplicemente *insieme su V* è caratterizzato da:

1. il dominio, indicato con *ins*, che consiste di tutti gli insiemi di elementi di tipo *V*;
2. le operazioni, che sono definite come segue:
 - verifica se un insieme è vuoto; questa operazione, dato un insieme *T* (cioè un elemento di *ins*), verifica se esso contiene o meno elementi,
 - inserimento di un elemento: dato un insieme *T* ed un elemento *E* di tipo *V*, fornisce come risultato l'insieme composto da tutti gli elementi di *T* più l'elemento *E*,
 - cancellazione di un elemento: dato un insieme *T* ed un elemento *E* di tipo *V*, restituisce come risultato l'insieme composto da tutti gli elementi di *T* meno l'elemento *E*,
 - verifica di appartenenza: dato un insieme *T* ed un elemento *E* di tipo *V*, verifica se *E* appartiene a *T*;
3. la costante *insieme_vuoto*, che denota l'insieme che non contiene elementi.

L'esempio suggerisce diversi spunti interessanti. La prima osservazione

riguarda il fatto che nella specifica del dominio del tipo di dato è spesso necessario fare riferimento ad altri domini. Ad esempio, il tipo di dato *insieme* è definito in termini di un altro dominio V , quello cui appartengono gli elementi che formano gli insiemi.

La seconda osservazione riguarda le operazioni: anche la loro definizione richiede, in generale, il riferimento a tipi di dato diversi da quello che si sta definendo. Ad esempio, l'operazione di verifica se un insieme è vuoto restituisce un valore di tipo booleano.

La terza osservazione riguarda ancora le operazioni: nell'esempio precedente le abbiamo definite informalmente, fornendo una descrizione a parole del risultato che esse devono calcolare. Tuttavia, una descrizione informale non è, in genere, considerata sufficiente per definire in modo completo e non ambiguo il significato dell'operazione. Ad esempio, la definizione che abbiamo dato per il tipo insieme non specifica in alcun modo qual è l'effetto dell'operazione di cancellazione se il valore E non è presente nell'insieme T .

Nel seguito affronteremo separatamente i problemi sollevati dalle tre osservazioni precedenti.

I primi due problemi ci suggeriscono una nuova definizione di tipo astratto di dato, più precisa di quella data all'inizio di questo paragrafo.

Definizione. Un tipo astratto di dato è una tripla $\langle S, F, C \rangle$, dove:

- S è un insieme di domini $\{V_1, V_2, \dots, V_n\}$, tra i quali viene individuato un dominio speciale, chiamato "dominio di interesse", che rappresenta l'insieme di valori del tipo che si sta definendo;
- F è un insieme di funzioni (o operazioni) $\{F_1, F_2, \dots, F_m\}$; ciascuna F_i è del tipo:

$$F_i : V_{i_1} \times V_{i_2} \times \dots \times V_{i_h} \rightarrow V_k$$

- dove ogni elemento di $\{V_{i_1}, \dots, V_{i_h}, V_k\}$ è un elemento di S , ed almeno uno tra i domini $\{V_{i_1}, \dots, V_{i_h}, V_k\}$ è il dominio di interesse. In altre parole, il dominio ed il codominio di ciascuna F_i sono formati da domini contenuti in S , con il vincolo che il dominio di interesse o è il codominio di F_i oppure appare tra gli insiemi che definiscono il suo dominio;
- C è un insieme di elementi che denotano valori del dominio di interesse.

Possiamo quindi riformulare la definizione del tipo di dato *insieme* nel modo seguente.

Il tipo *insieme su V* è un tipo astratto di dato $\langle S, F, C \rangle$, dove S è $\{ins,$

*boolean, V}, F è {*test_insieme_vuoto*, *inserisci*, *cancella*, *test_appartenenza*}, e C è {*insieme_vuoto*}; *ins* è il dominio di interesse, cioè rappresenta l'insieme di tutti i valori di tipo “insieme di elementi di V”, mentre *boolean* è l'insieme dei valori di tipo booleano. Le operazioni in F sono definite sui seguenti domini:*

test_insieme_vuoto: *ins* → *boolean*
inserisci: *ins* × *V* → *ins*
cancella: *ins* × *V* → *ins*
test_appartenenza: *ins* × *V* → *boolean*

Infine, la costante *insieme_vuoto* rappresenta l'insieme che non contiene elementi.

Per illustrare il significato delle suddette operazioni, dobbiamo specificarne il risultato in funzione del valore degli operandi. Nelle considerazioni che seguono, I rappresenta un elemento di *ins*, e v rappresenta un elemento di *V*.

La prima operazione verifica se un insieme è vuoto: *test_insieme_vuoto*(I) restituisce il valore *true* se I non contiene elementi, *false* altrimenti.

La seconda operazione consente di inserire un elemento in un insieme: se v è elemento di I, allora il risultato di *inserisci*(I, v) è I stesso, altrimenti il risultato è l'insieme ottenuto aggiungendo ad I l'elemento v.

La terza operazione consente di cancellare un elemento da un insieme: se I non contiene v tra i suoi elementi, allora il risultato di *cancella*(I, v) è I stesso, altrimenti il risultato è l'insieme ottenuto eliminando l'elemento v da I.

Infine, la quarta operazione verifica se un elemento appartiene ad un insieme: il risultato di *test_appartenenza*(I, v) è *true* se e solo se v è elemento di I.

Si noti che ciò che abbiamo appena definito è spesso chiamata *struttura algebrica eterogenea*, dove il termine eterogenea si riferisce al fatto che la struttura algebrica coinvolge più insiemi diversi tra loro. La nozione di struttura algebrica data nel par. 2.1 dell'appendice, è un caso particolare della definizione che abbiamo appena dato, il caso, cioè, in cui S è composto da un singolo dominio.

Il terzo problema, cioè quello di specificare la semantica delle operazioni in modo preciso, coinvolge aspetti molto più vasti e complessi, che esulano dagli scopi di questo testo. Nella definizione delle strutture di dati che presentiamo nel seguito, specificheremo il significato delle varie operazioni mediante una descrizione in linguaggio naturale, così come abbiamo fatto per la definizione del tipo *insieme*.

Concludiamo questa sezione con un'ulteriore osservazione sulle operazioni.

unione (A,B):
se *test_insieme_vuoto (A)* è true
allora il risultato è *B*
altrimenti considera un qualunque elemento *a* di *A*;
se *test_appartenenza (B,a)* è true
allora il risultato è *unione(cancella(A,a),B)*
altrimenti il risultato è
inserisci(unione(cancella(A,a),B),a)

Fig. 1 - L'operazione *unione*

Le operazioni che si specificano quando si definisce un tipo astratto di dato, vengono dette *operazioni primitive*. È importante osservare che a partire dalle operazioni primitive è possibile definire altre operazioni sul tipo astratto. Ad esempio, nel caso del tipo *insieme*, l'unione

$$\text{unione: } \text{ins} \times \text{ins} \rightarrow \text{ins}$$

non compare tra le operazioni primitive, ma può essere espressa in termini di esse, ad esempio nel modo specificato in fig. 1.

1.2. La rappresentazione dei tipi astratti

In questo paragrafo affrontiamo le problematiche relative alla rappresentazione di un tipo di dato mediante un altro tipo di dato. Questo problema si manifesta quando un tipo astratto deve essere utilizzato in un programma, ed è quindi necessario rappresentare il tipo stesso in termini dei costrutti (tipi, variabili, ecc.) del linguaggio di programmazione in cui il programma viene scritto.

Un metodo di rappresentazione, o semplicemente una rappresentazione, di un tipo astratto $\langle S, F, C \rangle$, deve fornire le regole per rappresentare:

1. il dominio di interesse e tutti gli altri domini in *S*;
2. le operazioni in *F*;
3. le costanti in *C*.

Il primo punto riguarda la rappresentazione dei domini su cui il tipo astratto è definito. Supponiamo, ad esempio, di voler rappresentare il tipo *insieme su V* in Pascal, assumendo che *V* sia il tipo intervallo di interi [1..5]. Una possibile rappresentazione è la seguente: al generico valore *I* di tipo *insieme I*,

nel seguito) viene fatto corrispondere un array A con indici da 1 a 5, i cui elementi sono di tipo *boolean*. Il valore del generico elemento $A[e]$ sarà *true* se e appartiene a I, *false* altrimenti. Un esempio di insieme I e del corrispondente array che lo rappresenta compare in fig. 2.

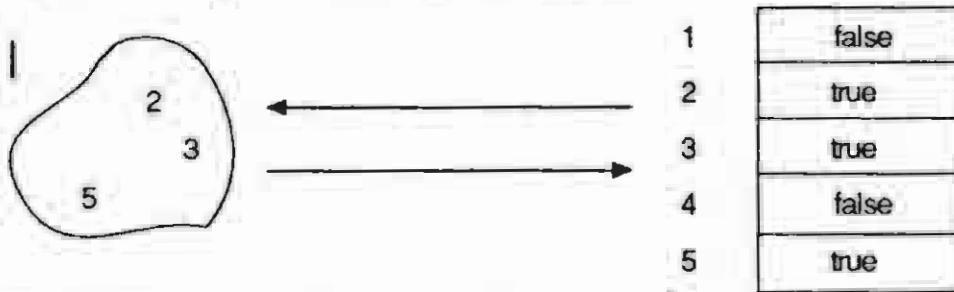


Fig. 2 - Esempio di rappresentazione di un insieme mediante un array

Si noti che ciò che abbiamo appena definito è una corrispondenza tra l'insieme dei valori del tipo astratto *insieme* su V (con $V=\{1,2,3,4,5\}$) e l'insieme dei valori di tipo "array con 5 componenti di tipo *boolean*". In questo modo, dato un qualunque valore del tipo *insieme* su V possiamo far corrispondere ad esso almeno un valore del tipo

insieme_array = array [1.. 5] of boolean

Generalizzando, possiamo affermare che un metodo di rappresentazione di un tipo T mediante un altro tipo T' definisce una corrispondenza *rapp* tra i valori dei domini di T ed i valori dei domini di T'.

Il secondo punto richiede che ogni operazione primitiva del tipo T venga realizzata mediante una operazione sul tipo di dato scelto per rappresentare T. Quando T deve essere rappresentato mediante un tipo concreto di un linguaggio di programmazione, si fa spesso uso di astrazioni funzionali (in Pascal utilizzando procedure o funzioni) per realizzare le operazioni primitive di T. Riferiamoci ancora al caso del tipo astratto *insieme* rappresentato mediante il tipo *insieme_array*. Per realizzare in Pascal l'operazione *test_insieme_vuoto*, si potrà definire una funzione di tipo *boolean*, con un parametro A di tipo *insieme_array*, che scandisce l'array A e restituisce *true* se e solo se il valore di tutte le sue componenti è *false*. Per realizzare l'operazione *inserisci(I,e)* si potrà definire una procedura che assegna il valore *true* alla componente di indice e dell'array che rappresenta I (si veda la fig. 3).

È evidente che le istruzioni che realizzano una operazione sul tipo astratto dipendono dalla semantica di tale operazione. Ciò sottolinea la necessità di disporre di una specifica completa e corretta delle funzioni definite su un tipo astratto: quanto più tale specifica è comprensibile ed esauriente, tanto più è

```
procedure inserisci(var A: insieme_array; e: V);
    { inserisce l'elemento e nell'insieme rappresentato dall'array A }
begin
    A[e] := true
end;
```

Fig. 3 - La procedura inserisci

facile formulare le funzioni in esame in termini di istruzioni del linguaggio e verificare che queste ultime eseguano effettivamente le operazioni previste nella formulazione astratta.

Il terzo punto, infine, richiede che le costanti definite per il tipo astratto vengano codificate in maniera opportuna nel linguaggio di programmazione. Ad esempio, per rappresentare la costante *insieme_vuoto* in un programma, si può definire una variabile *ins_vuoto* di tipo *insieme_array*, assegnarle il corretto valore mediante l'istruzione:

```
for i := 1 to 5
do ins_vuoto[i] := false
```

e stabilire che tale valore non potrà mai cambiare nel corso del programma (ricordiamo che in Pascal non è possibile definire una costante di tipo **array**).

Tenendo presente le considerazioni fatte, possiamo dare la seguente definizione:

Definizione. Una rappresentazione di un tipo $T = \langle S, F, C \rangle$ è una tripla $\langle T', rapp, OP \rangle$, dove:

1. $T' = \langle S', F', C' \rangle$ è il tipo di dato mediante il quale si rappresenta T ;
2. *rapp* è una corrispondenza tra i valori dei domini del tipo T ed i valori dei domini del tipo T' ;
3. *OP* è una funzione che associa ad ogni operazione primitiva definita per il tipo T una operazione (primitiva o no) definita per il tipo T' .

In generale esistono differenti rappresentazioni di un tipo astratto. Ad esempio, il tipo astratto *insieme* su V , con $V = \{1, 2, 3, 4, 5\}$, può essere rappresentato in Pascal utilizzando direttamente il costruttore di tipo **set**. Altri metodi di rappresentazione per gli insiemi verranno analizzati nel par. 4.

La scelta di una rappresentazione deve basarsi sulla valutazione e sul confronto rispetto a due caratteristiche fondamentali, la *correttezza* e l'*efficienza*.

Riguardo al primo aspetto, affinchè una rappresentazione $\langle T', rapp, OP \rangle$ di un tipo T mediante un altro tipo T' sia corretta, è necessario che:

1. per ogni valore di tipo T esista almeno un valore di tipo T' che lo rappresenta e, viceversa, per ogni valore di tipo T' sia possibile risalire in modo univoco al corrispondente valore di T . Più precisamente, è necessario che per ogni elemento d di D (dominio di interesse del tipo T) esista almeno un elemento d' di D' (dominio di interesse del tipo T') tale che $rapp(d)=d'$, e per ogni elemento d' di D' esista uno ed un solo elemento d di D tale che $rapp(d)=d'$.
2. Ogni operazione primitiva o di T sia correttamente realizzata dalla corrispondente operazione $o' = OP(o)$ definita per T' . Per illustrare cosa significhi che una operazione di T sia correttamente realizzata da una operazione di T' , consideriamo una operazione o di T :

$$o: D_1 \times D_2 \times \dots \times D_n \rightarrow D$$

e sia $o' = OP(o)$ la corrispondente operazione di T' :

$$o': D'_1 \times D'_2 \times \dots \times D'_n \rightarrow D'$$

Diciamo che o' realizza correttamente o se per ogni $d_1 \in D_1, \dots, d_n \in D_n$, il risultato che si ottiene applicando o' ai valori $rapp(d_1), \dots, rapp(d_n)$ è la rappresentazione del valore di D che si otterrebbe applicando direttamente l'operazione o ai valori d_1, \dots, d_n , cioè:

$$o'(rapp(d_1), \dots, rapp(d_n)) = rapp(o(d_1, \dots, d_n))$$

Nel caso della rappresentazione del tipo *insieme* su V che abbiamo illustrato, si può facilmente verificare che, dato un valore di tipo *insieme* su V esiste un valore di tipo *insieme_array* che lo rappresenta, e, viceversa, dato un valore A di tipo *insieme_array*, esso rappresenta uno ed un solo valore di tipo *insieme* su V , in particolare l'insieme formato da tutti i valori E di V tale che la componente di A di indice E è uguale a *true*. Analogamente, si può dimostrare che la procedura "inserisci" realizza correttamente l'operazione *inserisci* definita sul tipo astratto. Infatti, eseguendo la procedura "inserisci" su un qualunque valore A di tipo *insieme_array* (sia t il corrispondente valore di tipo *insieme*) ed un valore E di tipo V , si ottiene un valore A' in cui la componente di A di indice E vale *true*, mentre le altre componenti hanno lo stesso valore di A . Quindi il nuovo valore A' rappresenta il valore t' di tipo *insieme* tale che: E è elemento di t' e qualunque altro valore f di V è elemento di t' se e solo se f è elemento di t . Ciò significa che:

$$t' = inserisci(t, E)$$

Abbiamo, quindi, dimostrato che il valore restituito dall'attivazione:

inserisci(rapp(t), rapp(E))

è uguale a *rapp(inserisci(t,E))*, e quindi la procedura "inserisci" realizza correttamente la corrispondente operazione primitiva.

Si noti che il primo dei requisiti descritti per la correttezza di una rappresentazione impone che la relazione *rapp* sia totale su D. Se *rapp* non è totale, cioè esistono valori di T ai quali non corrisponde alcun valore di T', significa che non tutti i valori di tipo T sono rappresentabili mediante il metodo di rappresentazione. Questo comporta che non tutti i tipi astratti siano rappresentabili correttamente in termini dei tipi di un linguaggio di programmazione. Ad esempio, il tipo *integer* del Pascal non consente di rappresentare tutto il dominio del tipo astratto intero, che è infinito: in ogni implementazione, il tipo *integer* è formato da un sottoinsieme degli interi, e perciò la correttezza della rappresentazione è garantita solo per tale sottoinsieme.

L'efficienza di una rappresentazione si misura generalmente facendo riferimento a due parametri: l'occupazione di memoria ed il costo di esecuzione delle operazioni.

Per avere una misura dell'efficienza di una rappresentazione rispetto all'occupazione di memoria, è necessario valutare quanta area di memoria è richiesta per rappresentare il generico valore del tipo astratto. Ad esempio, rappresentando il tipo *insieme* su V con il tipo *insieme_array*, per memorizzare il generico elemento W di *insieme* si richiede una occupazione di memoria proporzionale al numero di elementi contenuti in V. È evidente che, a parità di condizioni, è preferibile una rappresentazione che necessita di una minore occupazione di memoria.

Riguardo al secondo punto, la scelta di una rappresentazione richiede una valutazione del costo delle varie operazioni (ad esempio in termini del numero di operazioni elementari da eseguire) tenendo presente le caratteristiche del metodo di rappresentazione usato. In generale, una rappresentazione si può rivelare efficiente rispetto a certe operazioni, ed inefficiente rispetto ad altre. È quindi utile un'analisi di quali sono le operazioni che si prevede di eseguire più spesso; la scelta cadrà sulla rappresentazione che si rivela più efficiente rispetto a tali operazioni.

Come vedremo nei successivi paragrafi, le esigenze di efficienza rispetto ai due parametri suddetti sono talora incompatibili tra loro. Se questo accade, la scelta dovrà tenere presente le caratteristiche del singolo problema ed i vincoli imposti dalle risorse di calcolo, privilegiando un aspetto a discapito dell'altro.

L'efficienza deve essere tenuta in considerazione anche nella realizzazione delle operazioni non primitive. In linea di principio, una operazione non primitiva può sempre essere realizzata mediante composizione di operazioni primitive. Può, però, accadere che le caratteristiche del metodo di rappresentazione consentano di realizzare una operazione non primitiva in modo molto efficiente, che non corrisponda alla formulazione astratta, necessariamente espressa in termini delle operazioni primitive. Consideriamo, ad esempio, l'unione di due insiemi, di cui abbiamo precedentemente fornito la formulazione astratta. Tenendo presente le caratteristiche della rappresentazione del tipo *insieme* mediante il tipo *insieme_array*, si può realizzare l'operazione *unione* in Pascal nel modo indicato in fig. 4.

```
procedure unione (A,B: insieme_array; var C: insieme_array);
  | calcola l'unione di due insiemi A e B e memorizza il risultato in C; gli
    insiemi sono rappresentati mediante array |
  var i: integer;
  begin
    for i := 1 to 5
      do C[i] := A[i] or B[i]
  end;
```

Fig. 4 - La procedura *unione*

Si noti che la procedura effettua l'unione in modo molto semplice, mediante un ciclo di scansione degli elementi dei due array, assegnando il valore true alla generica componente dell'array risultante se e solo se almeno una delle corrispondenti componenti di A e B è pari a *true*. Tale formulazione è molto diversa da quella data precedentemente in termini delle operazioni primitive sul tipo *insieme*.

Quando si utilizza un tipo concreto T' per rappresentare un tipo astratto T in un linguaggio di programmazione imperativo, i valori del tipo stesso vengono, in generale, memorizzati nelle variabili di tipo T'. Nel realizzare un'operazione definita per T, si dovrà scegliere la modalità con cui l'operazione stessa restituisce il risultato. Sono possibili, in linea di principio, due modalità:

- realizzando l'operazione mediante una unità di programma di tipo **function**; in questo caso il valore restituito dall'unità di programma è la rappresen-

tazione del valore calcolato dall'operazione astratta;

- realizzando l'operazione mediante una unità di programma di tipo **procedure**; in questo caso l'effetto dell'operazione è quello di memorizzare in una variabile (una variabile globale oppure un parametro di tipo **var**) la rappresentazione del risultato.

Sebbene la prima modalità sia quella che più naturalmente corrisponde alla formulazione astratta dell'operazione, le limitazioni dei linguaggi di programmazione obbligano, talvolta, a scegliere la seconda modalità.

Consideriamo, come esempio, il problema di realizzare l'operazione *inserisci* definita per il tipo *insieme*. Secondo la prima modalità si dovrebbe realizzare l'operazione mediante una **function** in cui il risultato sarebbe di tipo *insieme_array*. Tuttavia, poiché in Pascal una **function** non può fornire come risultato un valore di tipo **array**, la seconda modalità è obbligata.

La seconda modalità può essere realizzata in vari modi. Ad esempio, nel caso della realizzazione della operazione *inserisci*, si può scegliere tra le seguenti possibilità:

1. il risultato dell'operazione viene memorizzato nella stessa variabile in cui è memorizzato il valore originario, cioè l'effetto dell'operazione è quello di modificare la variabile in cui è memorizzato il valore originario. In questo caso la procedura che realizza l'operazione *inserisci* è quella precedentemente mostrata in questo paragrafo;
2. il risultato viene memorizzato in una nuova variabile, in generale diversa da quella in cui è memorizzato il valore originario; in questo caso la procedura è:

```
procedure inserisci2(A:insieme_array; e: V; var ris: insieme_array);
var i: integer;
begin
  for i := 1 to 5
    do ris[i] := A[i];
    ris[e] := true
end;
```

Si noti che questo secondo metodo consente di ottenere il risultato della operazione senza modificare il valore della variabile in cui è memorizzato il valore originario. Ad esempio, l'attivazione:

inserisci2(mio_insieme, 5, altro_insieme)

non modifica il valore della variabile *mio_insieme*.

2. I VETTORI E LE MATRICI

Le *matrici* sono tipi di dato i cui valori rappresentano una corrispondenza tra un insieme di dati detti *indici*, ed un insieme di valori di un tipo *V*. Il valore di tipo *V* che è in corrispondenza con il valore *i* dell'indice in una matrice *M*, viene detto l'*elemento* (o la *componente*) di *M* di indice *i*.

Il tipo matrice è presente in molti linguaggi di programmazione. In questa sezione noi analizziamo tale tipo di dato senza riferirci ad alcun linguaggio in particolare, e, successivamente, studiamo alcuni metodi per la sua rappresentazione.

Definizione. Il tipo astratto *matrice* è definito da:

- $S = \{indice, V, mat\}$, dove *mat* è il dominio di interesse, e *indice* e *V* sono insiemi di valori di tipo qualunque con *indice* numerabile;
- $F = \{accedi, memorizza\}$, dove:

$$accedi: mat \times indice \rightarrow V$$

$$memorizza: mat \times indice \times V \rightarrow mat$$

- $C = \emptyset$

In generale, *indice* è il prodotto cartesiano di *n* insiemi:

$$indice = ind_1 \times ind_2 \times \dots \times ind_n$$

cioè i suoi valori sono *n*-ple della forma

$$\langle i_1, \dots, i_n \rangle$$

dove ogni i_j appartiene all'insieme ind_j . Il valore *n* stabilisce la dimensione della matrice. Se $n=1$, la matrice si dice *vettore*.

L'operazione *accedi* ha il seguente significato: se *M* è un valore di tipo *matrice* ed *I* è un valore di tipo *indice*, *accedi*(*M*,*I*) restituisce il valore dell'elemento di *M* che è in corrispondenza con il valore *I* dell'indice. Si noti che *accedi*(*M*,*I*) viene spesso scritto come *M*[*I*].

Invece, *memorizza* (*M*,*I*,*E*) restituisce il valore *M'*, di tipo *matrice*, in cui gli elementi di indice diverso da *I* sono uguali a quelli di *M*, mentre l'elemento di indice *I* è uguale a *E*.

2.1. Il tipo *matrice* nei linguaggi di programmazione

In questo paragrafo affrontiamo alcuni aspetti relativi alla rappresentazione

e l'uso del tipo matrice nei linguaggi di programmazione, riferendoci in particolare al linguaggio Pascal.

Il tipo *matrice* può essere rappresentato in Pascal mediante il tipo concreto **array**, creando una corrispondenza tra gli elementi della matrice e gli elementi dell'array. Come noto, il tipo degli indici di un **array** deve essere un tipo intervallo o un tipo enumerato, e deve essere specificato all'atto della dichiarazione del tipo **array** nel programma.

Al fine di realizzare le operazioni astratte *accedi* e *memorizza*, il Pascal offre un semplice meccanismo che consente di riferirsi alla singola componente dell'array: se A è un array, e i è un valore per il suo indice, il termine A[i] denota la componente di A di indice i. Così, se A è di tipo "matrice a due dimensioni di interi, con indici di tipo [1..10]", per effettuare l'operazione astratta *memorizza* (A,<1,3>,120), si eseguirà la seguente istruzione di assegnazione:

A[1,3] := 120;

mentre per accedere alla componente di indice <3,5>, ad esempio per stamparne il valore, si scriverà:

write(A[3,5])

Facciamo ora alcune considerazioni sul metodo che in genere si utilizza per rappresentare le varie componenti di un array nella memoria e per realizzare effettivamente l'accesso ed il riferimento ad esse. Si noti che nella gestione degli array nei linguaggi di programmazione ad alto livello questi aspetti sono completamente a carico del traduttore del linguaggio, e possono, quindi, essere ignorati dai programmati. Prenderemo in esame solo il caso di array a due dimensioni, con indici di valori interi. Le considerazioni che seguono sono facilmente generalizzabili.

Supponiamo che A sia una variabile di tipo **array** con due indici di tipo 'intervallo di interi, il primo con valori da 1 a N, ed il secondo con valori da 1 a M. L'array può essere rappresentato graficamente mediante una tabella, in cui il primo indice corrisponde alle righe, ed il secondo alle colonne (si veda la fig. 5).

Supponiamo, senza perdita di generalità, che ogni elemento dell'array occupi una locazione di memoria. In queste condizioni, l'array viene rappresentato in una zona di memoria contigua, cioè composta da locazioni di indirizzi consecutivi, riservando una locazione ad ogni componente. Un possibile metodo per rappresentare A in una zona di memoria contigua è il

	1	2	3	...	M
1	5	21	3	...	6
2	2	7	6	...	18
3	15	12	20	...	26
.
.
.
N	5	8	14		30

Fig. 5 - Rappresentazione tabellare di un array a due dimensioni

seguente: gli elementi di A vengono memorizzati per colonne a partire da una certa locazione iniziale, che indichiamo con $ind(A)$, seguendo, nell'ambito della stessa colonna, l'ordine dato dall'indice di riga. Così, le prime N locazioni a partire da $ind(A)$ sono destinate agli elementi della prima colonna, le seconde N locazioni agli elementi della seconda colonna, e così via fino alla M-esima colonna. Nell'ambito della generica colonna j, viene prima memorizzato l'elemento in prima riga, poi quello in seconda, e così via fino a quello in N-esima riga.

È facile verificare che, adottando il metodo descritto, l'accesso al generico elemento di A, di indice $\langle i,j \rangle$, viene tradotto nell'accesso alla locazione di indirizzo:

$$ind(A) + (j - 1)*N + i - 1$$

Si noti che nel caso dei vettori, l'espressione precedente si semplifica: l'accesso al generico elemento di indice i (che supponiamo intero) viene tradotto nell'accesso alla locazione di indirizzo:

$$ind(A) + i - 1$$

2.2. Rappresentazioni compatte di matrici

Generalmente, nei linguaggi di programmazione, il numero delle componenti di un array, e quindi l'occupazione di memoria per esso richiesta, è fisso, e viene stabilito in fase di compilazione. Ad esempio, il numero di locazioni necessarie per la memorizzazione di un array a due dimensioni con indici [1..N] e [1..M], è pari a N*M (ipotizzando che per la singola componente si

utilizzi una locazione di memoria).

Ci sono dei casi, però, in cui le caratteristiche dei valori di tipo matrice che si utilizzano in un programma consentono una rappresentazione più efficiente rispetto a quella appena descritta. Ad esempio, per una matrice di interi in cui sia noto che la grande maggioranza degli elementi è uguale a zero, si potrebbero memorizzare i soli elementi diversi da zero. In genere si usa il termine di *matrice sparsa* per indicare una matrice in cui la gran parte degli elementi ha un valore prefissato (detto valore predominante). Per tali matrici si possono utilizzare rappresentazioni ad hoc, dette *compatte*. L'idea fondamentale di tali rappresentazioni è quella di memorizzare solo gli elementi con valore diverso dal valore predominante, associando ad essi il valore del corrispondente indice. Nel seguito, per analizzare tali metodi, ci riferiremo a matrici di due dimensioni con due indici di tipo intero.

Un primo metodo prevede di rappresentare ogni elemento della matrice sparsa A diverso dal valore predominante mediante tre informazioni: l'indice di riga, l'indice di colonna ed il valore.

Queste terne sono a loro volta rappresentate mediante un array C di record, ognuno con tre campi. C viene detta rappresentazione *compatta* di A. Le terne di C vengono in genere memorizzate in modo ordinato, ad esempio ordinandole per riga e, nell'ambito della stessa riga, per colonna.

Una componente di C (ad esempio la prima) è utilizzata per memorizzare, nei campi relativi all'indice di riga e di colonna, il numero di righe N ed il numero di colonne M di A, e nel campo relativo al valore, il valore dell'elemento predominante.

Se *max* è il numero di componenti di C, e m (con $m < \max - 1$) è il numero di elementi di A memorizzati in C, le informazioni ad essi relative sono rappresentate negli elementi di indice da 2 a $m+1$ di C, e nelle eventuali componenti successive di C (cioè quelle di indice maggiore di $m+1$) viene memorizzata una terna di valori che non corrisponde ad alcun componente della matrice A.

In fig. 6 compare una matrice sparsa A di componenti intere, in cui la maggioranza degli elementi ha valore 0, e la corrispondente matrice compatta C a tre informazioni. Per C si è scelto un numero di righe pari a 8. I sei elementi diversi da 0 di A sono memorizzati nelle componenti di indici da 2 a 7. La terna in posizione 8 contiene valori degli indici di riga e colonna che sono maggiori rispettivamente di N ed M: tali valori indicano che la terna non corrisponde ad alcuna componente di A.

Passiamo alla realizzazione delle operazioni *accedi* e *memorizza*. Faremo

	1	2	3	4	5	6	RIGA	COLONNA	VALORE	
1	8	0	0	11	0	0	1	4	6	0
2	0	0	21	0	0	0	2	1	1	8
3	0	0	0	15	0	0	3	1	4	11
4	0	0	3	16	0	0	4	2	3	21
							5	3	4	15
							6	4	3	3
							7	4	4	16
							8	10	10	0

Fig. 6 - Rappresentazione compatta a tre informazioni

riferimento alla seguente dichiarazione di tipi:

```

elem_mat_compatta = record
    riga, colonna: integer;
    valore: V
end;
tipo_mat_compatta = array [1..nmax_compatta] of elem_mat_compatta;

```

dove si è assunto che *nmax_compatta* sia stata dichiarata come costante. Assumiamo anche che la matrice compatta contenga almeno una terna finale in cui i valori degli indici di riga e di colonna sono maggiori rispettivamente di N e M; questo semplifica le condizioni di uscita del ciclo di scansione della matrice compatta, come vedremo in seguito.

Le due operazioni *accedi* e *memorizza* possono essere realizzate mediante le unità di programma mostrate nelle figg. 7 e 8.

Riguardo alla correttezza della rappresentazione compatta a tre informazioni, si osservi che, dato un valore della matrice sparsa, esiste un valore della matrice compatta che lo rappresenta se e solo se il numero di componenti della matrice compatta è sufficiente alla memorizzazione degli elementi diversi dal valore predominante. Viceversa, dato un valore della matrice compatta è possibile risalire in modo univoco al corrispondente valore della matrice sparsa. Lasciamo al lettore la verifica che la funzione *accedi* e la procedura *memorizza* realizzano correttamente le corrispondenti operazioni sul tipo astratto.

```

function accedi (c: tipo_mat_compatta; i,j: integer): V;
{ realizza l'operazione accedi su una matrice rappresentata mediante la
matrice compatta c a tre informazioni }
var h: integer;
begin
  h := 2;
  while (c[h].riga < i)
    do h := h + 1;
  while (c[h].riga = i) and (c[h].colonna < j)
    do h := h + 1;
  if (c[h].riga = i) and (c[h].colonna = j)
    then accedi := c[h].valore
    else accedi := c[1].valore
end;

```

Fig. 7 - La funzione accedi

Passando all'efficienza, si può verificare facilmente che il metodo descritto necessita di uno spazio di memoria proporzionale al numero di elementi della matrice compatta. Se *max_a* è il numero massimo di elementi di A che si devono memorizzare esplicitamente, l'occupazione di memoria richiesta per rappresentare la matrice originaria A è:

$$3 * (\text{max_a} + 2)$$

Confrontando questo dato con il numero di locazioni necessarie per la matrice sparsa (cioè $N \times M$), si può notare che si ha risparmio di memoria solo se:

$$\text{max_a} < (N \times M - 6)/3$$

A fronte di un potenziale risparmio di memoria, si registra, però, un maggiore costo di esecuzione per le operazioni *accedi* e *memorizza* sulla matrice sparsa. Nel caso di rappresentazione non compatta, l'accesso ad un elemento è diretto e, perciò, molto efficiente. Al contrario, nel caso di rappresentazione compatta l'accesso ad un elemento necessita di un numero di operazioni che dipende dal numero di elementi esplicitamente memorizzati. Considerando come misura del costo di esecuzione il numero di confronti effettuati, si vede che la funzione *accedi*, nel caso più sfavorevole, richiede di accedere a tutti gli elementi della matrice originaria esplicitamente memorizzati nella matrice compatta.

```

procedure memorizza (var c: tipo_mat_compatta; i, j: integer;
                    val: V);
{ realizza l'operazione memorizza su una matrice rappresentata mediante
la matrice compatta c a tre informazioni }
var h,q: integer;
begin
  h := 2;
  while (c[h].riga < i)
  do h := h + 1;
  while (c[h].riga = i) and (c[h].colonna < j)
  do h := h + 1;
  { controllo sul valore val da memorizzare }
  if val = c[1].valore
  then begin
    if (c[h].riga=i) and (c[h].colonna=j)
    then { elimina l'elemento di c in posizione h }
    for q := h+1 to nmax_compatta
    do begin
      c[q-1].riga := c[q].riga;
      c[q-1].colonna := c[q].colonna;
      c[q-1].valore := c[q].valore
    end
  end
  else if (c[h].riga=i) and (c[h].colonna =j)
  then c[h].valore := val
  else { se c'è posto, memorizza il nuovo elemento in c }
    if c[nmax_compatta-1].riga <= c[1].riga
    then write ('memorizzazione impossibile')
    else begin
      for q := nmax_compatta-1 downto h+1
      do begin
        c[q].riga := c[q-1].riga;
        c[q].colonna := c[q-1].colonna;
        c[q].valore := c[q-1].valore
      end;
      c[h].riga := i;
      c[h].colonna := j;
      c[h].valore := val
    end
  end;

```

Fig. 8 - La procedura *memorizza*

Anche il caso di memorizzazione di un elemento può richiedere un tempo di esecuzione considerevole, dovuto alla necessità di spostare elementi nella rappresentazione compatta. Se, ad esempio, si vuole assegnare il valore 22 all'elemento in posizione <3,1> nella matrice A di fig. 6, occorrerà inserire un nuovo elemento in posizione 5 nel corrispondente array C, e ciò provocherà lo spostamento degli elementi di C che si trovano nelle posizioni successive.

La rappresentazione descritta è anche inefficiente rispetto alle operazioni non primitive di scansione di una riga o di una colonna. La scansione di una colonna, in particolare, è problematica, in quanto i suoi elementi si trovano in posizioni non adiacenti nella matrice compatta. Se si vuole migliorare l'efficienza della rappresentazione rispetto a tale operazione, si può modificare il metodo di rappresentazione descritto, associando ad ogni elemento E memorizzato nella matrice compatta, una nuova informazione, che indichi la posizione nell'array del successivo elemento diverso dal valore predominante con lo stesso indice di colonna di E (all'ultimo elemento di una colonna si può associare il valore 0). La rappresentazione che ne risulta viene chiamata *rappresentazione compatta a quattro informazioni*. Mostriamo in fig. 9 tale rappresentazione per la matrice A di fig.6.

RIGA	COLONNA	VALORE	RIFERIMENTO
1	4	6	0
2	1	1	8
3	1	4	11
4	2	3	21
5	3	4	15
6	4	3	0
7	4	4	16
8	10	10	0

Fig. 9 - Rappresentazione compatta a quattro informazioni

Un'ulteriore variante della rappresentazione compatta prevede di utilizzare un array R (vettore di accesso) per memorizzare informazioni utili per sempli-

ficare l'accesso ad un elemento. Il numero di componenti di R è pari al numero di righe della matrice sparsa, e la generica componente di indice I indica la prima posizione nella matrice compatta in cui si trova un elemento della matrice sparsa il cui indice di riga è pari a I . Mostriamo in fig. 10 la rappresentazione della matrice A di fig. 9 secondo questo metodo. È ovvio che l'uso dell'array R semplifica l'accesso ad un elemento: volendo accedere all'elemento della matrice sparsa di indici $\langle i, j \rangle$, si accederà a $R[i]$, il cui valore ci indicherà da dove iniziare la scansione della matrice compatta. Se r è il numero medio di elementi in una stessa riga della matrice sparsa che hanno un valore diverso dal valore predominante, l'accesso ad un elemento richiederà al massimo $r+1$ confronti. Si noti che l'uso dell'array R consente di eseguire efficientemente l'operazione di scansione di una riga.

R	RIGA	COLONNA	VALORE
2	1	4	6
4	2	1	1
5	3	1	4
6	4	2	3
	5	3	4
	6	4	3
	7	4	4
	8	10	10
			0

Fig. 10 - Rappresentazione con vettore di accesso

In fig. 11 compare una tabella riassuntiva delle caratteristiche dei metodi di rappresentazione delle matrici sparse. Nella tabella si fa riferimento ad una matrice a due dimensioni con N righe e M colonne; k rappresenta il numero di elementi della matrice compatta, m rappresenta il numero di elementi della matrice sparsa il cui valore è diverso dal valore predominante, ed r rappresenta il numero medio per riga di elementi diversi dal valore predominante.

L'analisi della tabella indica che le rappresentazioni compatte di una matrice sono vantaggiose solo se lo spazio di memoria necessario per memorizzare la matrice compatta è molto minore di $N+M$: in questo caso, infatti, il risparmio di memoria compensa l'inefficienza dell'operazione di accesso ad un elemento della matrice.

	<i>Matrice sparsa</i>	<i>Rappresentazione a 3 informazioni</i>	<i>Rappresentazione con vett. di accesso</i>
<i>Occupazione di memoria</i>	$N \cdot M$	$3 \cdot k$	$3 \cdot k + N$
<i>Numero massimo di confronti per l'accesso ad un elemento</i>	0	$m+1$	$r+1$

Fig. 11 - Confronto tra le rappresentazioni di una matrice sparsa

3. LE LISTE

In questo paragrafo ci occupiamo del tipo astratto *lista*, e dei metodi per la sua rappresentazione. Il par. 3.1 è dedicato al tipo lista semplice i cui valori sono sequenze di valori elementari, detti atomi. Illustreremo la definizione di lista semplice (detta anche lista di atomi) e descriveremo diversi metodi per la sua rappresentazione nei linguaggi di programmazione. Nel par. 3.5 parleremo di liste i cui elementi possono a loro volta essere liste, ed affronteremo anche per esse il problema della rappresentazione.

3.1. Le liste semplici

Il tipo astratto lista consente di trattare sequenze di elementi di un determinato tipo. Ricordiamo che per sequenza si intende un multinsieme finito e ordinato di elementi. In questa e nella prossima sezione ci occupiamo del tipo astratto *lista semplice*.

Definizione. Il tipo lista semplice è un tipo astratto di dato $\langle S, F, C \rangle$, dove:

1. $S = \{ lis, atomo, boolean \}$; *lis* è il dominio di interesse, e *atomo* è il dominio degli elementi che formano le liste;

2. $F = \{ cons, car, cdr, null \}$, con:

cons: *atomo* \times *lis* \rightarrow *lis*

car: *lis* \rightarrow *atomo*

cdr: *lis* \rightarrow *lis*

null: *lis* \rightarrow *boolean*

3. $C = \{ lista_vuota \}$, dove *lista_vuota* è la costante che denota la lista che non contiene alcun elemento.

L'operazione *cons* consente di inserire un dato elemento in testa alla lista. Se L è un valore appartenente al dominio *lis* e A è un valore appartenente al dominio *atomo*, *cons(A,L)* restituisce la lista semplice costituita da A seguito da tutti gli elementi di L.

L'operazione *car* consente di individuare il primo elemento di una lista. Se L è la lista vuota, allora *car(L)* non è definita. Se L è una lista non vuota, *car(L)* restituisce il primo elemento di L.

L'operazione *cdr* restituisce la lista ottenuta da un'altra lista ignorando il primo elemento. Se L è la lista vuota, allora *cdr(L)* non è definita. Se L è una lista non vuota, *cdr(L)* restituisce la lista semplice ottenuta da L ignorandone il primo elemento.

Infine, *null* verifica se una lista è vuota: *null* restituisce *true* se la lista L è vuota, *false* altrimenti.

Le liste vengono di solito rappresentate graficamente elencandone gli elementi, racchiusi tra parentesi tonde. Tale notazione è detta rappresentazione parentetica.

Esempio. I seguenti sono esempi di liste semplici di interi rappresentate mediante la notazione parentetica:

1. ()
2. (8 25 6 90 6)
3. (52)

La lista 1 è la lista vuota. La lista 2 è formata da 5 elementi. Si noti che, sebbene il terzo ed il quinto elemento abbiano lo stesso valore, essi rappresentano due elementi distinti della lista. Infine, la lista 3 è formata da un singolo elemento.

Utilizzando la notazione parentetica possiamo ora presentare alcuni esempi di applicazione delle operazioni sulle liste.

Esempio. Assumiamo ancora di operare su liste di interi.

L'operazione *car* applicata a (9 3 32 21 3) restituisce il valore 9.

L'operazione *cdr* applicata a (9 3 32 21 3) restituisce la lista (3 32 21 3).

L'operazione *cons* applicata a 8 e (9 3 32 21 3) restituisce la lista (8 9 3 32 21 3).

Un'ulteriore caratterizzazione dei valori di tipo lista semplice si ottiene

mediante una definizione induttiva del dominio *lis*: ogni valore di tipo lista semplice o è una sequenza vuota di elementi oppure è una sequenza formata da un elemento appartenente al dominio *atomo* seguito a sua volta da un valore di tipo lista semplice. Questa definizione si rivela particolarmente utile per esprimere semplici algoritmi ricorsivi sulle liste.

Affrontiamo ora il problema di rappresentare le liste semplici mediante i tipi concreti dei linguaggi di programmazione. È bene osservare che pochi linguaggi di programmazione dispongono del tipo concreto lista. Il *lisp* è uno di questi, come si vedrà nel cap. 10.

Nei linguaggi che non dispongono del tipo concreto lista, si utilizzano diversi metodi di rappresentazione. Si distingue tra due rappresentazioni fondamentali: la rappresentazione sequenziale e la rappresentazione collegata.

3.2. La rappresentazione sequenziale

Una lista può essere rappresentata mediante un array ad una dimensione: ogni valore di tipo lista viene rappresentato mediante un valore di tipo **array** in modo che ogni elemento della lista venga memorizzato in una componente dell'array. Poiché il numero di elementi che compongono la lista può variare, si utilizza una variabile (che possiamo chiamare *primo*) per il valore dell'indice della componente dell'array in cui è memorizzato il primo elemento della lista, ed un'altra variabile (*lunghezza*) che indica di quanti elementi è composta la lista rappresentata. Gli elementi della lista diversi dal primo sono memorizzati consecutivamente nelle componenti dell'array che lo seguono. La condizione di lista vuota è rappresentata dal valore 0 per le variabili *primo* e *lunghezza*.

Ad esempio, supponendo di far uso di un array di N (con N uguale a 12) componenti intere, con indici da 1 a N, possiamo rappresentare un valore di tipo lista semplice nel modo mostrato in fig. 12.

Si noti che nell'esempio di fig. 12, le componenti dell'array il cui indice non cade nell'intervallo [1..5], non sono significative e saranno utilizzate se e quando la lunghezza della lista crescerà oltre il valore 5.

Il metodo di rappresentazione che stiamo illustrando viene detto rappresentazione sequenziale mediante array, o semplicemente rappresentazione sequenziale di una lista semplice, ed il termine stesso evidenzia che l'ordine degli elementi della lista corrisponde alla sequenzialità delle componenti dell'array.

Illustriamo ora come le operazioni definite per il tipo di dato lista possono essere realizzate in Pascal mediante operazioni definite per il tipo **array**.

Lista	Rappresentazione sequenziale		
(4 5 1 21 45)	1	4	<i>Primo</i> 1
	2	5	
	3	1	
	4	21	
	5	45	<i>Lunghezza</i> 5
	6	78	
	7	12	
	8	1	
	9	-5	
	10	0	
	11	-2	
	12	61	

Fig. 12 - Rappresentazione sequenziale di una lista semplice

Consideriamo una lista di elementi interi e supponiamo di disporre della seguente dichiarazione:

```

const nmax_lista = 30;
type tipo_atomi = integer;
tipo_lista = record
    elementi: array [1..nmax_lista] of
        tipo_atomi;
    primo, lunghezza: integer
end;

```

Le operazioni *null* e *car* possono essere realizzate mediante le funzioni mostrate nelle figg. 13 e 14.

L'operazione *cons* può invece essere realizzata assumendo che la lista risultante venga ottenuta modificando l'array che rappresenta la lista originaria, ed aggiornando le variabili *primo* e *lunghezza*; si veda la fig. 15.

La procedura controlla dapprima che la lista originaria non occupi già tutto l'array, e poi effettua l'operazione, decrementando di uno il valore di *lis.primo*, e memorizzando il nuovo elemento nella componente dell'array in posizione corrispondente al nuovo valore di *lis.primo*. Il decremento è effettuato modulo *nmax_lista*: se il valore di *lis.primo* è 1 ad esso viene assegnato come nuovo valore *nmax_lista*, altrimenti il nuovo valore è semplicemente *lis.primo* -1. Si

```
function null (lis: tipo_lista ): boolean;
{ verifica se la lista semplice lis è vuota; lis è rappresentata mediante array }
begin
    null := (lis.lunghezza = 0)
end;
```

Fig. 13 - La funzione *null*

```
function car(lis: tipo_lista): tipo_atomi;
{ esegue l'operazione car sulla lista lis; per la lista si usa la rappresentazione sequenziale mediante array }
begin
    if null(lis)
        then { l'operazione non può essere effettuata }
            writeln ('operazione non eseguibile')
        else car := lis.elementi[lis.primo]
    end;
```

Fig. 14 - La funzione *car*

```
procedure cons (e: tipo_atomo; var lis: tipo_lista);
{ effettua l'operazione cons su e e lis }
begin
    if lis.lunghezza = nmax_lista
        then write('insufficiente dimensione dell'array')
    else begin
        if null (lis)
            then lis.primo:= 1
        else
            if lis.primo = 1
                then lis.primo := nmax_lista
            else lis.primo := lis.primo - 1;
            lis.elementi[lis.primo] := e;
            lis.lunghezza := lis.lunghezza + 1
    end
end;
```

Fig. 15 - La procedura *cons*

```

procedure cdr(var lis: tipo_lista );
{ effettua l'operazione cdr su lis }
begin
  if null(lis)
  then write ('operazione non eseguibile')
  else begin
    if lis.lunghezza = 1
    then lis.primo := 0
    else lis.primo := (lis.primo mod nmax_lista) + 1;
    lis.lunghezza := lis.lunghezza - 1
  end
end;

```

Fig. 16 - La procedura *cdr*

noti che la soluzione alternativa di mantenere sempre il primo elemento della lista in prima posizione nell'array, richiederebbe, al momento di effettuare l'operazione *cons* in una lista non vuota, lo spostamento di una posizione di tutti gli elementi della lista.

Analogamente, l'operazione *cdr* può essere realizzata come mostrato in fig. 16.

Si noti che anche in questo caso, l'incremento di uno del valore di *lis.primo* è effettuato modulo *nmax_lista*.

Per verificare la correttezza della rappresentazione descritta è necessario verificare che:

1. dato un qualunque valore di tipo lista, esiste almeno un valore di tipo **array** che lo rappresenta e, viceversa, dato un valore di tipo **array**, è possibile risalire in modo univoco al valore di tipo lista che esso rappresenta;
2. le operazioni primitive definite sulle liste sono correttamente realizzate dalle funzioni e dalle procedure che abbiamo descritto.

Riguardo alla prima condizione, si può notare che se l'array ha un numero di componenti maggiore o uguale alla lunghezza della lista da rappresentare, vale la condizione che, dato un valore di tipo lista, esiste un valore di tipo array che lo rappresenta.

Inoltre, data una qualunque lista e l'array che la rappresenta, insieme alla variabile che stabilisce il numero di elementi significativi nell'array, è facile verificare che, se il valore di tale variabile è minore del numero di componenti

dell'array, ad ogni elemento della lista corrisponde una ed una sola componente dell'array. Inoltre, l'ordine con cui si susseguono gli elementi nella lista è conservato nell'array. Quindi, dato un valore di tipo **array**, e due variabili, una per la posizione del primo elemento della lista, e l'altra per la sua lunghezza, è univocamente determinato il valore di tipo lista da esso rappresentato.

Riguardo alla seconda condizione, verifichiamo che la procedura *cons* realizza correttamente l'operazione primitiva *cons*. Sia *a* un elemento di *atomo* e *L* un elemento di *lis*. Indichiamo con *L'* il valore di tipo *tipo-lista* che rappresenta *L* secondo la rappresentazione sequenziale. Dobbiamo dimostrare che l'attivazione:

cons(a, v)

dove *v* è una variabile di tipo *tipo-lista* il cui valore prima dell'attivazione è *L'*, modifica la variabile *v* assegnandole un valore *L''* che rappresenta il valore del tipo astratto *lista* risultato di *cons* (*a,L*). A questo scopo, analizziamo la procedura distinguendo i tre casi possibili:

- se *lis.lunghezza=nmax_lista*, allora la procedura non modifica il valore di *lis*;
- se *lis.lunghezza< nmax_lista* e *lis.primo=1*, allora *lis.primo* viene posto a *nmax_lista*;
- se *lis.lunghezza< nmax_lista* e *lis.primo>>1*, allora il valore di *lis.primo* viene decrementato di 1.

Inoltre, nel secondo e terzo caso, a *lis.elementi[lis.primo]* viene assegnato il valore *a*, ed il valore di *lis.lunghezza* viene incrementato di 1. Quindi, nel primo caso la procedura lascia inalterato il valore di *v*, mentre nel secondo e nel terzo caso la procedura modifica il valore di *v* (si noti il passaggio per riferimento per il parametro *lis*) costruendo la rappresentazione di un nuovo valore di tipo *lista*, ottenuto dal valore originario aggiungendo *a* come elemento iniziale. Ne segue che la procedura *cons* realizza correttamente l'operazione *cons* se e solo se il numero di elementi della lista originaria *L* è minore di *nmax_lista*.

Analoghe considerazioni valgono per le funzioni *null* e *car*, e per la procedura *cdr*.

Osserviamo che nella rappresentazione sequenziale le operazioni primitive vengono realizzate in modo molto efficiente. Infatti, analizzando le procedure e le funzioni presentate sopra, si vede che tutte le operazioni primitive sul tipo astratto *lista* richiedono l'esecuzione di un numero costante di istruzioni, che non dipende, cioè, dal numero di elementi che compongono la lista.

La rappresentazione sequenziale delle liste consente di eseguire efficientemente anche altre operazioni. Ad esempio, per l'accesso ad un elemento della

lista di cui sia nota la sua posizione nella sequenza di elementi che formano la lista stessa, si può sfruttare il meccanismo di accesso diretto alle componenti di un array: se si vuole accedere all' i -esimo elemento della lista, basterà accedere all'elemento dell'array che si trova in posizione $primo + i - 1$ (dove la somma è effettuata, come al solito, modulo $nmax_lista$). Anche in questo caso il costo di tale operazione è costante: in particolare, è indipendente sia dal numero di elementi che formano la lista che dalla posizione dell'elemento a cui si vuole accedere.

Uno degli inconvenienti maggiori della rappresentazione sequenziale è costituito dal fatto che le dimensioni dell'array sono fisse. Ciò crea dei problemi se, durante l'esecuzione del programma, la lista cresce oltre tali dimensioni. Inoltre, il metodo descritto può comportare un notevole spreco di memoria: infatti la quantità di memoria occupata da una variabile di tipo **array** è proporzionale al numero di componenti dell'array, e quindi lo spazio di memoria utilizzato per rappresentare una lista è indipendente dalla sua lunghezza.

Un ulteriore svantaggio riguarda le operazioni (non primitive) di inserimento e cancellazione di un elemento in una posizione diversa dalla prima e dall'ultima, che comportano necessariamente lo spostamento di elementi nell'array. È evidente che per queste operazioni la rappresentazione sequenziale si rivela particolarmente inefficiente.

3.3. La rappresentazione collegata

L'idea fondamentale della rappresentazione collegata di una lista è quella di memorizzare i suoi elementi associando ad ognuno di essi una particolare informazione (detta *riferimento*), che permetta di individuare la locazione in cui è memorizzato l'elemento successivo. In questo modo, la sequenzialità degli elementi della lista non è rappresentata mediante l'adiacenza delle locazioni di memoria in cui essi sono memorizzati (come invece accade nella rappresentazione sequenziale), ma da una apposita informazione ad essi associata: da qui il nome di rappresentazione collegata. Vedremo che questo principio consente di superare alcuni degli svantaggi della rappresentazione sequenziale.

Per visualizzare la rappresentazione collegata di una lista viene utilizzata una notazione grafica in cui gli elementi sono rappresentati mediante nodi ed i riferimenti mediante archi che li collegano. Un esempio di tale notazione è

mostrato in fig. 17. Da essa si può notare che si utilizza un riferimento al primo elemento della lista (riferimento iniziale), e si usa un simbolo speciale come riferimento associato all'ultimo nodo. Nel caso in cui la lista sia vuota, tale simbolo compare direttamente nel riferimento iniziale.

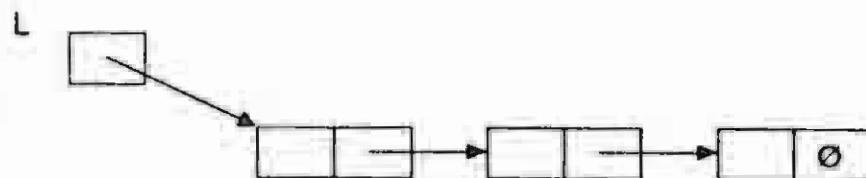


Fig. 17 - Notazione grafica per la rappresentazione collegata di una lista

In termini di questa notazione, è molto semplice caratterizzare le azioni necessarie per eseguire le operazioni primitive sulle liste. Ad esempio, l'operazione *cdr* applicata alla lista di fig. 18, può essere effettuata aggiornando il riferimento iniziale nel modo indicato in fig. 19.



Fig. 18 - Una lista

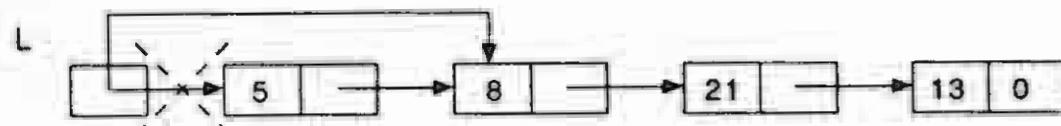


Fig. 19 - La lista di fig. 18 dopo l'operazione *cdr*

Anche l'operazione *cons* può essere effettuata semplicemente aggiornando il riferimento iniziale, in modo che in esso venga memorizzato il riferimento ad un nuovo nodo, come mostrato in fig. 20, in cui si assume di eseguire l'operazione *cons*(16,L).

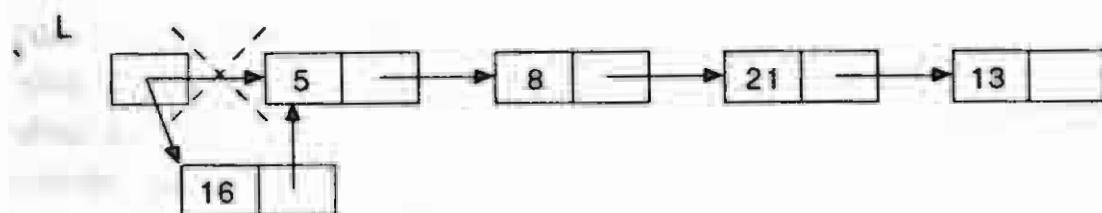


Fig. 20 - L'operazione *cons*

Analizziamo anche il caso in cui si voglia effettuare un'operazione di inserimento e cancellazione di elementi in una generica posizione della lista. L'eliminazione di un elemento che non sia il primo comporta la modifica del riferimento associato all'elemento che lo precede. Se, ad esempio, si vuole

eliminare il terzo elemento della lista L, si deve procedere come in fig. 21: il riferimento associato al secondo nodo assume il valore pari al riferimento che era associato al terzo nodo, cioè quello da eliminare.

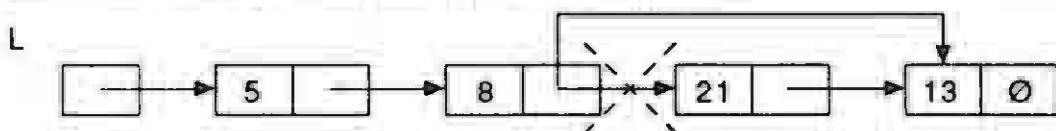


Fig. 21 - Eliminazione di un elemento

Per l'operazione di inserimento, si veda la fig. 22: volendo inserire un elemento dopo il nodo N, è necessario disporre di un nuovo nodo M, memorizzarvi il riferimento al nodo che segue N, e memorizzare in N il riferimento a M.

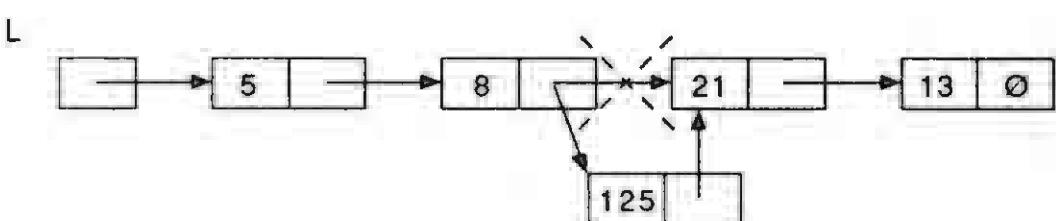


Fig. 22 - Inserimento di un elemento

Dagli esempi presentati si può facilmente comprendere che con la rappresentazione collegata non è più necessario spostare elementi per realizzare le operazioni di inserimento e cancellazione: tali operazioni, infatti, vengono eseguite effettuando opportune modifiche al valore dei riferimenti. D'altra parte, l'operazione di accesso ad un elemento di cui sia nota la posizione nella lista è ora molto più dispendiosa, poiché richiede la scansione di tutti gli elementi che precedono quello voluto.

Torneremo successivamente sull'efficienza della rappresentazione collegata delle liste. Esaminiamo ora due metodi per realizzare la rappresentazione collegata: il primo utilizza gli array, mentre il secondo fa uso del tipo puntatore.

3.3.1. Rappresentazione collegata mediante array

La rappresentazione collegata mediante array consiste nell'associare ad ogni elemento della lista una componente dell'array, costituita da:

- il valore del corrispondente elemento della lista;
- il riferimento all'elemento successivo, cioè il valore dell'indice dell'array in corrispondenza del quale esso è memorizzato.

In questo modo, gli elementi della lista sono memorizzati in ordine qualunque nell'array e il loro ordinamento nella lista è ricostruibile seguendo i riferimenti. Un'ulteriore informazione è necessaria per conoscere l'indice della componente dell'array in cui è memorizzato l'elemento iniziale della lista. Il valore 0 per il riferimento indica che il corrispondente elemento è l'ultimo della lista. Se tale valore compare nel riferimento iniziale, la lista è vuota.

Come esempio, consideriamo la lista di fig. 23, ed il corrispondente array, che la rappresenta secondo il metodo che abbiamo appena illustrato.

La sequenza degli elementi che formano la lista è ricostruibile nel seguente modo: si inizia dalla componente dell'array corrispondente al valore di *inizio*, dove è memorizzato, nel campo *dato*, il valore del primo elemento. Seguendo il riferimento ad esso associato (valore 6 memorizzato nel campo *prox* nella componente di indice 3) si trova il secondo elemento della lista, il cui valore è 5. Gli altri elementi vengono trovati in modo analogo. Il valore 0 per il riferimento associato all'elemento in posizione 4 indica che esso è l'ultimo elemento della lista. Si noti che il simbolo “-” nell'array di fig. 23 denota un valore non significativo per la lista.

Una possibile dichiarazione di tipi in Pascal per la rappresentazione collegata mediante array è:

```
const nmax_lista = 100;
type tipo_atomi = integer;
    tipo_riferimento = integer;
    tipo_array_lista = array [1..nmax_lista] of
        record
            dato: tipo_atomi;
            prox: tipo_riferimento
        end;
```

in cui:

- *nmax_lista* è la dimensione massima dell'array e, conseguentemente, la lunghezza massima ammissibile per la lista;
- *tipo_array_lista* è il tipo dell'array;
- *tipo_atomi* è il tipo degli elementi della lista, che in questo caso si considerano interi;
- *tipo_riferimento* è il tipo dei riferimenti, compreso quello iniziale.

Passando alla realizzazione delle operazioni, è immediato verificare che *null*

<i>Lista</i>	<i>Array</i>																																	
$(4, 5, 1, 21, 45)$	$inizio = 3$																																	
	<table border="1"> <thead> <tr> <th></th> <th><i>dato</i></th> <th><i>prox</i></th> </tr> </thead> <tbody> <tr><td>1</td><td>-</td><td>-</td></tr> <tr><td>2</td><td>21</td><td>4</td></tr> <tr><td>3</td><td>4</td><td>6</td></tr> <tr><td>4</td><td>45</td><td>0</td></tr> <tr><td>5</td><td>-</td><td>-</td></tr> <tr><td>6</td><td>5</td><td>8</td></tr> <tr><td>7</td><td>-</td><td>-</td></tr> <tr><td>8</td><td>1</td><td>2</td></tr> <tr><td>9</td><td>-</td><td>-</td></tr> <tr><td>10</td><td>-</td><td>-</td></tr> </tbody> </table>		<i>dato</i>	<i>prox</i>	1	-	-	2	21	4	3	4	6	4	45	0	5	-	-	6	5	8	7	-	-	8	1	2	9	-	-	10	-	-
	<i>dato</i>	<i>prox</i>																																
1	-	-																																
2	21	4																																
3	4	6																																
4	45	0																																
5	-	-																																
6	5	8																																
7	-	-																																
8	1	2																																
9	-	-																																
10	-	-																																

Fig. 23 - Rappresentazione collegata mediante array

e *car* sono realizzabili correttamente in modo molto semplice. Maggiori difficoltà presenta, invece, la realizzazione dell'operazione *cons*. Così come abbiamo fatto per la rappresentazione sequenziale, realizzeremo questa operazione in modo che la lista risultante sia memorizzata nello stesso array che rappresenta la lista originaria.

Poichè l'operazione *cons* richiede di inserire un nuovo elemento nella lista, per realizzare tale operazione è necessario disporre di una componente dell'array in cui non sia memorizzato alcun elemento significativo della lista. Nel seguito chiameremo *libera* una tale componente. Conoscendo il valore dell'indice di una componente libera, possiamo inserire il nuovo elemento nel campo *dato* di tale componente ed aggiornare opportunamente i riferimenti affinché sia rispettata la sequenza degli elementi della nuova lista. Sorge allora il problema di individuare la posizione di una componente libera nell'array.

Il metodo più comune per individuare tale componente fa uso della cosiddetta *lista libera*. La lista libera è memorizzata nello stesso array usato per memorizzare la lista e raccoglie in modo collegato le componenti libere dell'array. Quando si deve inserire un nuovo elemento nella lista, si preleva una componente dalla lista libera (in genere la prima) e la si utilizza per memorizzare il nuovo elemento, collegandolo in modo opportuno agli altri elementi della lista. Analogamente, quando si elimina un elemento dalla lista si inserisce nella lista libera la componente in cui tale elemento è memorizzato (aggiornando di

nuovo i vari riferimenti sia della lista principale che della lista libera), in modo che lo spazio da esso occupato sia recuperabile successivamente.

Esempio. Assumiamo di disporre dell'array di fig. 24, che rappresenta una lista semplice (con la relativa variabile *inizio*) ed una lista libera, la cui prima componente è quella in posizione corrispondente al valore della variabile *inizio_lista_libera*.

inizio_lista_libera = 8
inizio = 2

	<i>dato</i>	<i>prox</i>	
1	56	0	<i>lista rappresentata:</i>
2	28	4	
3	-	6	(28, 32, 14, 56)
4	32	7	
5	-	10	
6	-	5	
7	14	1	
8	-	3	
.	.	.	
.	.	.	
.	.	.	

Fig. 24 - Un array che rappresenta una lista

Si vuole inserire un nuovo elemento (di valore 44) in prima posizione nella lista. Per memorizzare il nuovo elemento si fa uso della prima componente della lista libera. La situazione che ne risulta è quella mostrata in fig. 25.

Nella notazione grafica della rappresentazione collegata, le operazioni da effettuare per inserire l'elemento 44 sono schematizzabili nel modo indicato dalla fig. 26.

La lista libera rappresenta quindi un serbatoio da cui prelevare componenti libere dell'array e in cui riversare le componenti dell'array che non sono più utilizzate per la lista. Le operazioni eseguibili sulla lista libera sono:

1. la costruzione (o inizializzazione), che serve a collegare tra loro le varie componenti dell'array che fanno parte inizialmente della lista libera;
2. il prelievo di una componente della lista libera, se esiste;
3. la restituzione di una componente alla lista libera.

inizio_lista_libera = 3
inizio = 8

	<i>dato</i>	<i>prox</i>	
1	56	0	<i>lista rappresentata:</i>
2	28	4	(44, 28, 32, 14, 56)
3	-	6	
4	32	7	
5	-	10	
6	-	5	
7	14	1	
8	44	2	
.	.	.	
.	.	.	
.	.	.	

Fig. 25 - L'array dopo l'inserimento di un nuovo elemento

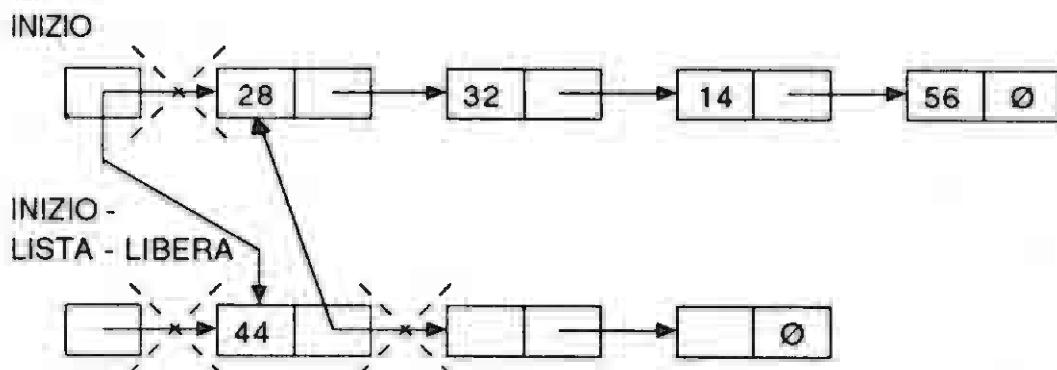


Fig. 26 - Rappresentazione grafica dell'operazione di inserimento

Nel seguito mostreremo l'uso della lista libera nelle operazioni *cdr* e *cons*. Prima, però, illustriamo un possibile metodo per realizzare l'operazione di inizializzazione di una lista libera (si veda la fig. 27).

Avendo illustrato i principi su cui si basano l'utilizzo e la gestione della lista libera, abbiamo ora tutti gli elementi per realizzare le operazioni *cons* e *cdr*. Esse effettueranno in modo opportuno le operazioni di prelievo e rilascio di componenti dalla lista libera, aggiornando i riferimenti delle due strutture collegate. Le due procedure sono mostrate nelle figg. 28 e 29.

Lasciamo al lettore la verifica della correttezza della rappresentazione collegata mediante array, ed analizziamone l'efficienza.

```

procedure inizializza_liste_libera (var iniz_ll: tipo_riferimento;
                                    var elementi: tipo_array_lista);
{ inizializza la lista libera memorizzata in "elementi" il cui riferimento
iniziale è "iniz_ll"; l'inizializzazione viene effettuata collegando tra loro
tutte le componenti dell'array nell'ordine corrispondente al valore del
l'indice }
var i: tipo_riferimento
begin
    for i := 1 to nmax_lista
        do elementi[i].prox := i + 1;
        iniz_ll := i;
        elementi[nmax_lista].prox := 0
end;

```

Fig. 27 - La procedura *inizializza_liste_libera*

```

procedure cons(e: tipo_atomo;
                var elementi: tipo_array_lista;
                var lis, iniz_ll: tipo_riferimento);
{ effettua l'operazione cons su una lista rappresentata in modo collegato
mediante array; elementi è l'array e lis il riferimento iniziale; iniz_ll è il
riferimento iniziale della lista libera; l'effetto dell'operazione è quello di
modificare l'array in cui è memorizzata la lista affinché esso rappresenti
la lista risultante }
var temp: tipo_riferimento
begin
    if iniz_ll = 0
        then write("dimensione_insufficiente dell'array")
        else begin
            { il nuovo elemento viene memorizzato prelevando una compo-
            nente (la prima) dalla lista libera }
            temp := iniz_ll;
            iniz_ll := elementi[iniz_ll].prox;
            elementi[temp].dato := e;
            elementi[temp].prox := lis;
            lis := temp
        end
    end;

```

Fig. 28 - La procedura *cons*

```

procedure cdr (var lis, iniz_ll: tipo_riferimento;
                var elementi: tipo_array_lista);
{ effettua l'operazione cdr su una lista una lista rappresentata in modo
  collegato mediante array; elementi è l'array e lis il riferimento iniziale;
  iniz_ll è il riferimento iniziale della lista libera; l'effetto dell'operazio-
  ne è quello di modificare l'array in cui è memorizzata la lista affinché
  esso rappresenti la lista risultante }
var temp: integer;
begin
  if null(lis)
    then write('operazione non applicabile')
  else begin
    { si elimina il primo elemento della lista e si rilascia alla lista
      libera la componente non più utilizzata }
    temp := lis;
    lis := elementi[lis].prox;
    elementi[temp].prox := iniz_ll;
    iniz_ll := temp
  end
end;

```

Fig. 29 - La procedura *cdr*

È immediato verificare che uno dei principali svantaggi della rappresentazione sequenziale è superato: infatti, l'inserimento e l'eliminazione di un elemento non richiedono lo spostamento di altri elementi della lista. Ciò è ottenuto mediante l'uso della lista libera, così come abbiamo descritto precedentemente. La complicazione data dalla necessità di gestire la lista libera è compensata dal fatto che le operazioni di modifica della lista possono essere effettuate eseguendo un limitato numero di operazioni di aggiornamento dei riferimenti associati agli elementi.

Nella rappresentazione collegata mediante array rimangono, invece, i problemi connessi all'uso dell'array: in particolare, la dimensione dell'array rappresenta un limite alla crescita della lista e la quantità di memoria utilizzata non dipende dalla lunghezza che di volta in volta ha la lista, ma è costantemente pari alla dimensione dell'array.

Riguardo alla occupazione di memoria, si noti anche che la rappresentazione collegata richiede più spazio rispetto alla rappresentazione sequenziale, a causa della necessità di memorizzare i riferimenti.

Nonostante questi problemi, comunque, il metodo che abbiamo descritto è

importante perché consente la rappresentazione collegata di liste in quei linguaggi di programmazione in cui l'unico tipo di dato strutturato è l'**array** (come, ad esempio, il Fortran).

3.3.2. Rappresentazione collegata mediante puntatore

Il tipo puntatore è un tipo di dato i cui valori rappresentano indirizzi di locazioni di memoria. Le operazioni usualmente disponibili su una variabile *p* di tipo puntatore sono:

1. l'accesso alla locazione il cui indirizzo è memorizzato in *p*;
2. la richiesta di una nuova locazione di memoria (istruzione *new* in Pascal) e memorizzazione dell'indirizzo di tale locazione in *p*;
3. il rilascio della locazione di memoria il cui indirizzo è memorizzato in *p* (istruzione *dispose* in Pascal).

Mediante l'uso congiunto del tipo puntatore e del tipo **record**, si può realizzare in modo efficace la rappresentazione collegata di una lista: si fa corrispondere ad ogni elemento della lista un record, e si collegano i vari record mediante un campo di tipo puntatore. Il riferimento iniziale della lista sarà rappresentato da una variabile di tipo puntatore.

In Pascal, una dichiarazione di tipi che consente questa rappresentazione è:

```
type tipo_atomi = integer;
    punt = ^record_lista { tipo puntatore per la lista }
    record_lista = record
        val: tipo_atomi;
        next: punt
    end; { tipo record per gli elementi
           della lista }
    tipo_lista = punt;
```

Tenendo presente gli operatori che in genere i linguaggi mettono a disposizione per l'uso dei puntatori, si può facilmente verificare che l'uso della lista libera da parte del programmatore non è più necessario. In Pascal, ad esempio, si possono utilizzare le istruzioni *new* e *dispose* per richiedere e rilasciare i record che si utilizzano per la lista.

Presentiamo ora le procedure e le funzioni che realizzano le operazioni primitive sulle liste rappresentate in modo collegato mediante record e puntatori (si vedano le figg. 30, 31, 32 e 33).

```
function null(lis: tipo_lista): boolean;
begin
    null := lis=nil
end;
```

Fig. 30 - La funzione *null*

```
function car(lis: tipo_lista): tipo_atomi;
begin
    if null(lis)
    then write(' operazione non eseguibile')
    else car := lis^.val
end;
```

Fig. 31 - La funzione *car*

```
procedure cons(e: tipo_atomi; var lis: tipo_lista);
{ effettua l'operazione cons su una lista rappresentata in modo collegato
mediante record e puntatori; lis è il puntatore iniziale }
var temp: punt;
begin
    new(temp);
    temp^.val := e;
    temp^.next := lis;
    lis := temp
end;
```

Fig. 32 - La procedura *cons*

```
procedure cdr ( var lis: tipo_lista);
{ effettua l'operazione cdr su una lista rappresentata in modo collegato
mediante record e puntatori; lis è il puntatore iniziale della lista }
var temp: punt;
begin
    if null(lis)
    then write(' operazione non eseguibile')
    else begin
        temp := lis;
        lis := lis^.next;
        dispose(temp)
    end
end;
```

Fig. 33 - La procedura *cdr*

È facile verificare che la rappresentazione di liste che abbiamo presentato è corretta.

Come esempio di realizzazione di una operazione non primitiva, consideriamo la scansione di una lista, cioè l'analisi di tutti i suoi elementi. Questa operazione è particolarmente importante, perché è quella che viene eseguita quando si ricerca un valore in una lista.

Supponiamo che *tipo_atomi* sia un tipo i cui valori possono essere stampati in Pascal (ad esempio *integer*); vogliamo scandire la lista per stampare tutti i suoi elementi, nell'ordine in cui essi compaiono. Una procedura che risolve questo problema è mostrata in fig. 34.

```
procedure stampa(p: tipo_lista);
  { scandisce la lista il cui puntatore iniziale è “p”, per stampare il valore
    dei suoi elementi }
begin
  while p <> nil
  do begin
    write (p^.val);
    p := p^.next
  end
end;
```

Fig. 34 - La procedura *stampa*

La procedura utilizza il parametro *p* (passato per valore) per puntare ai vari record, iniziando dal primo e procedendo in sequenza. Ogni volta che *p* punta ad un record, viene eseguita l'istruzione che stampa il valore del corrispondente campo *val*.

La stessa operazione può essere realizzata in modo ricorsivo, osservando che scandire una lista *L* significa: controllare che la lista non sia vuota, analizzare il primo elemento della lista (cioè *car*(*L*)), e poi scandire ricorsivamente la lista che si ottiene ignorando il primo elemento di *L* (cioè scandire la lista *cdr*(*L*)). Avremo quindi la procedura in fig. 35.

Si noti che nella formulazione ricorsiva si è assunto che l'operazione *cdr* sia realizzata mediante una funzione. Tale realizzazione deve essere radicalmente diversa dalla quella che abbiamo precedentemente illustrato perché la sua esecuzione non deve modificare la sequenza di record originaria, cioè l'esecuzione dell'operazione non deve causare effetti collaterali.

Tenendo conto di questi requisiti, possiamo realizzare l'operazione *cdr* nel

```
procedure stampa(lis: tipo_lista);
  { procedura ricorsiva che scandisce la lista il cui puntatore iniziale è lis,
    per stampare il valore dei suoi elementi }
begin
  if not(null(lis))
  then begin
    write(lis^.val);
    stampa(cdr(lis))
  end
end;
```

Fig. 35 - Un'altra versione della procedura stampa

```
function cdr(lis: tipo_lista): tipo_lista;
  { realizza l'operazione cdr senza side effects; la lista è rappresentata
    mediante record e puntatori }
begin
  if not(null(lis))
  then cdr := lis^.next
end;
```

Fig. 36 - La funzione cdr

modo mostrato in fig. 36.

Riguardo all'efficienza della rappresentazione collegata mediante puntatori, si può facilmente osservare che utilizzando in maniera adeguata i puntatori e le operazioni su di essi, si usufruisce dei tipici vantaggi della rappresentazione collegata e, allo stesso tempo, si superano gli svantaggi della realizzazione mediante array: infatti, da una parte non è più necessario imporre a priori un limite massimo alla dimensione della lista e dall'altra si ha la possibilità di gestire la lista in modo che lo spazio di memoria da essa occupato sia proporzionale al numero dei suoi elementi. Per quest'ultimo punto è necessario rilasciare i record che eventualmente non vengono più utilizzati nel programma.

L'uso dei puntatori nella rappresentazione delle liste ci consente di fare una considerazione che ha validità generale: le variabili di tipo puntatore devono essere usate con molta accortezza dal programmatore. Il fatto che in tale variabili è memorizzato un riferimento ad una locazione di memoria può infatti

causare diversi problemi. Nel seguito, analizziamo due tipici problemi connessi all'uso del tipo puntatore in Pascal:

1. l'esecuzione di una unità di programma con un parametro di tipo puntatore può causare effetti collaterali anche se il parametro è passato per valore. Consideriamo l'esempio di fig. 37, in cui si assumono note le dichiarazioni di tipo precedentemente illustrate.

procedure *esempio_punt(p: punt);*

{ *p è il puntatore iniziale di una sequenza di record rappresentante una lista; l'esecuzione di questa procedura provoca effetti collaterali anche se l'unico parametro è passato per valore* }

begin

if *p* <> *nil*

then begin

p^.val := 10;

if *p^.next* <> *nil*

then *p^.next* := *p^.next^.next*

end

end;

Fig. 37 - La procedura *esempio*

È facile verificare che, sebbene la procedura non modifichi il valore del parametro attuale corrispondente a *p*, essa provoca una modifica del primo e del secondo record della lista. Ciò significa che l'esecuzione della procedura modifica alcuni oggetti visibili nell'unità chiamante pur non utilizzando né variabili globali né parametri passati per riferimento. È, quindi, particolarmente importante che il programmatore verifichi attentamente l'uso dei parametri di tipo puntatore:

2. utilizzando il tipo puntatore è possibile che diverse variabili contengano il riferimento allo stesso oggetto. L'effetto dell'esecuzione dell'istruzione *dispose(p)* (dove *p* è una variabile di tipo puntatore) è di rilasciare la locazione di memoria di indirizzo pari al valore di *p*. Può accadere che, al momento dell'esecuzione di tale istruzione, altre variabili abbiano lo stesso valore di *p*. In questo caso si dice che tali variabili, dopo l'esecuzione dell'istruzione *dispose* contengono un puntatore appeso (*dangling reference* in inglese), ed un eventuale accesso alla locazione di memoria di indirizzo pari al loro valore è considerato un errore. Il problema dei puntatori appesi deve essere, perciò, tenuto ben presente; per risolverlo è necessario avere il

controllo completo delle variabili di tipo puntatore nel programma, in modo che sia sempre possibile conoscere quali variabili puntatore abbiano uguale valore.

3.4. Alcune varianti della rappresentazione collegata

In questo paragrafo analizziamo due varianti della rappresentazione collegata delle liste semplici che presentano diversi aspetti interessanti e che vengono utilizzate per migliorare l'efficienza di alcune operazioni.

La prima variante prende il nome di *rappresentazione circolare*: essa è una rappresentazione collegata in cui l'ultimo elemento, invece di contenere un riferimento nullo, contiene il riferimento al primo nodo. In fig. 38 mostriamo un esempio di rappresentazione circolare della lista (12 14 8 2).

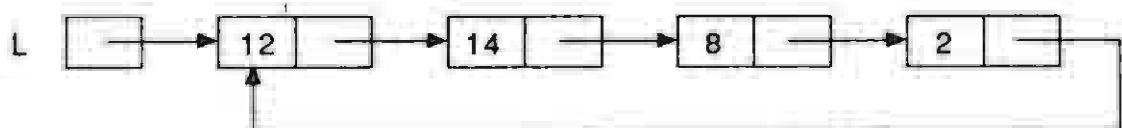


Fig. 38 - Rappresentazione circolare

Spesso si parla, impropriamente, di lista circolare per riferirsi a questa rappresentazione. Ovviamente, la rappresentazione circolare può essere realizzata sia con array che con record e puntatori.

Una seconda variante della rappresentazione collegata è la cosiddetta *rappresentazione simmetrica*: in essa ogni elemento contiene, oltre al riferimento al successivo nodo, anche il riferimento al precedente (esempio in fig. 39).

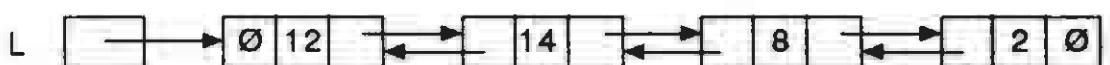


Fig. 39 - Rappresentazione simmetrica

In genere si usa il termine lista simmetrica per riferirsi ad una struttura come quella mostrata in fig. 39. Con questa rappresentazione si ha il vantaggio di poter scandire in modo semplice la lista in entrambe le direzioni, in avanti e all'indietro. Più in generale, la rappresentazione simmetrica viene usata per facilitare l'individuazione dell'elemento che precede un determinato elemento. In questo modo si possono effettuare le operazioni di inserimento e cancellazione senza dover utilizzare variabili aggiuntive. D'altra parte, le stesse operazioni richiedono un maggior numero di istruzioni; inoltre la rappresentazione simmetrica richiede un maggior spazio di memoria per i

riferimenti all'indietro.

Anche le liste simmetriche, come quelle circolari, possono essere realizzate sia mediante array, sia mediante record e puntatori.

Come esempio analizziamo il problema di inserire in una lista non vuota un elemento x , in posizione precedente ad un dato elemento di cui si conosce il riferimento p . Assumendo di utilizzare record e puntatori per la rappresentazione simmetrica, ed assumendo che per memorizzare il puntatore all'elemento precedente si utilizzi un campo aggiuntivo nei record, di nome *back*, il problema si può risolvere nel seguente modo:

```
procedure inserisci (var lis: tipo_lista; x: tipo_atomo; p: punt);
{ inserisce x nella lista non vuota lis; per lis si utilizza la rappresentazione
simmetrica con record e puntatori; l'inserimento avviene prima del nodo
puntato da p }
var nuovo: punt;
begin
  new (nuovo);
  nuovo^.val := x;
  nuovo^.back := p^.back;
  nuovo^.next := p;
  if lis = p
  then lis := nuovo
  else nuovo^.back^.next := nuovo;
  p^.back := nuovo
end;
```

3.5. Le liste composite

In questa sezione estendiamo la nozione di lista semplice a quella di lista i cui elementi possono a loro volta essere liste. Il tipo astratto di dato che ne risulta è sufficientemente generale da essere utilizzato per rappresentare altri tipi astratti, come vedremo nelle sezioni dedicate agli alberi e ai grafi. In questa sezione presentiamo dapprima la definizione di *lista composita* (o semplicemente *lista*) e successivamente descriviamo i principali metodi di rappresentazione.

Definizione. Il tipo lista è un tipo astratto di dati $\langle S, F, C \rangle$, dove:

- 1) $S = \{ \text{lista}, \text{atomo}, \text{boolean} \}$, in cui *lista* è il dominio di interesse;

2) $F = \{cons, car, cdr, null, test_atomo\}$

con:

$cons: (lista \cup atomo) \times lista \rightarrow lista$

$car: lista \rightarrow (lista \cup atomo)$

$cdr: lista \rightarrow lista$

$null: lista \rightarrow boolean$

$test_atomo: lista \cup atomo \rightarrow boolean$

3) $C = \{lista_vuota\}$; $lista_vuota$ denota la lista che non contiene alcun elemento.

Le operazioni $car, cdr, cons$ e $null$ sono generalizzazioni delle corrispondenti operazioni sulle liste semplici. Si noti che car restituisce un valore che può appartenere o all'insieme $atomo$, o all'insieme $lista$. Analogamente, il primo argomento dell'operazione $cons$ può essere o un atomo o una lista. La funzione $test_atomo$ che non compare tra le operazioni primitive delle liste semplici, si applica ad un elemento della lista e restituisce $true$ se tale elemento è un atomo, $false$ se è una lista.

Analogamente al caso delle liste semplici, anche per le liste si può usare una notazione parentetica per descriverne i valori.

Esempio. Mostriamo la rappresentazione parentetica di una lista i cui atomi sono di tipo intero.

(5 () 6 (7 8) (9 (12) 3) 14)

La lista è composta da 6 elementi. Il primo, il terzo ed il sesto elemento sono atomi, gli altri sono liste. In particolare, il secondo elemento è la lista vuota, il quarto elemento è una lista composta da due atomi, ed il quinto elemento è una lista composta da tre elementi, il secondo dei quali è a sua volta una lista composta da un singolo atomo.

Utilizzando la rappresentazione parentetica, mostriamo alcuni esempi di applicazione delle operazioni primitive.

Esempio. Faremo riferimento a liste in cui $atomo$ è il dominio dei valori interi. Se L è la lista (4 () 2), l'operazione car applicata a L restituisce 4. Invece, $cdr (L)$ restituisce () 2. Se L è la lista (12 3), si ha che $test_atomo (car (L)) = true$. Infine, se L è la lista vuota, si ha che $test_atomo (L) = false$, e $cons ((), L)$ restituisce (()).

Si noti che la definizione induttiva che abbiamo usato per descrivere il dominio delle liste semplici, può essere applicata anche al caso delle liste: una

lista o è una sequenza vuota di elementi o è una sequenza formata da un elemento seguito a sua volta da una lista. Ogni elemento o è un atomo (cioè un oggetto appartenente ad un certo insieme prefissato) oppure è, a sua volta, una lista.

La rappresentazione più comune per il tipo astratto lista è quella collegata, in cui, come per le liste semplici, ad ogni elemento corrisponde un nodo in cui viene memorizzato il riferimento al nodo che rappresenta l'elemento successivo. Poiché ogni elemento può essere o un atomo o una lista, ogni nodo, oltre al suddetto riferimento, contiene:

- un valore, se il corrispondente elemento è un atomo;
- il riferimento iniziale di una lista, se il corrispondente elemento è a sua volta una lista.

In fig. 40 mostriamo (in forma grafica) la rappresentazione collegata della seguente lista:

(5 () 6 (7 8) (9 (12) 3) 14)

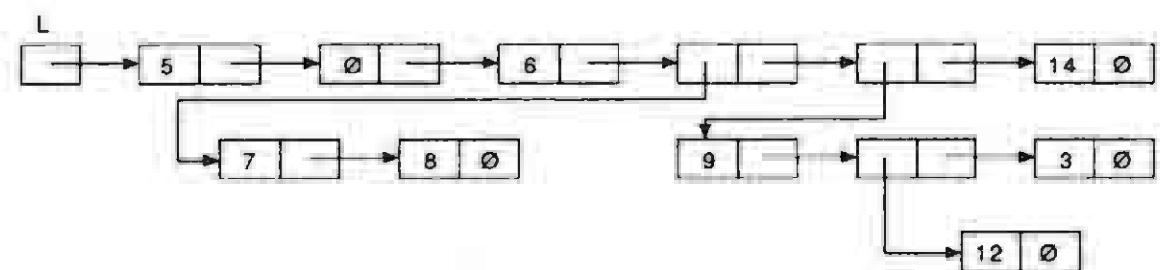


Fig. 40 - Rappresentazione collegata di una lista

L'insieme dei nodi che formano la rappresentazione collegata di una lista viene spesso chiamato lista doppia, ad indicare il fatto che ogni nodo può contenere due riferimenti. In generale si parla di lista multipla quando i nodi di una rappresentazione collegata contengono n (con $n > 1$) riferimenti. Il nome lista doppia (e lista multipla) non deve generare confusione: esso si riferisce ad un tipo di rappresentazione collegata che, nel caso che stiamo trattando, si utilizza per realizzare il tipo astratto lista.

Quando per realizzare la rappresentazione collegata di una lista si utilizzano record e puntatori, sorge il problema di definire in modo corretto il tipo **record**, considerando che in ogni record viene memorizzato alternativamente un atomo o un puntatore. Una possibile soluzione consiste nel definire il tipo **record** in modo che, oltre al campo per il puntatore al successivo elemento della lista, esso contenga due campi, uno per ognuna delle due alternative. Un ulteriore campo del record indica, mediante un prefissato valore, quale dei due

campi è significativo, cioè se nel record è memorizzato un atomo oppure il puntatore ad una lista. Una possibile dichiarazione di tipi e variabili in Pascal per rappresentare una lista è la seguente:

```
type tipo_atomo = ...;
punt = ^record_lista;
record_lista = record
    tipo_elemento: (atomo, lista);
    val: tipo_atomo;
    punt_lista: punt;
    next: punt
end;
```

Il campo *tipo_elemento* di un record R assume il valore *atomo* se l'elemento della lista rappresentato da R è un atomo, il valore *lista* nel caso contrario. Il campo *val* è significativo quando R rappresenta un atomo, nel qual caso in tale campo viene memorizzato il valore dell'atomo stesso. Quando, invece, R rappresenta una lista, nel campo *punt_lista* è memorizzato il puntatore alla lista che costituisce l'elemento rappresentato da R. Infine, nel campo *next* viene memorizzato il puntatore al record corrispondente all'elemento della lista successivo rispetto a quello rappresentato da R. Come nel caso delle liste di atomi, è necessario far uso di una variabile per memorizzare il puntatore iniziale della lista.

Un'ulteriore possibilità è offerta dall'uso dei record con parte variante. In questo caso si può utilizzare direttamente *tipo_elemento* come campo discriminante.

Si noti che la rappresentazione collegata può essere realizzata anche mediante array. In questo caso sarà necessario definire l'array in modo che ogni sua componente contenga due campi riferimento, ed un campo che discrimini se la componente stessa rappresenta un atomo oppure il riferimento ad una lista. Come nella analoga rappresentazione delle liste semplici, tali riferimenti rappresentano valori degli indici nell'array. Inoltre, anche in questo caso sarà necessario gestire una lista libera che raccoglie le componenti dell'array non utilizzate.

Riguardo alla realizzazione delle operazioni, presentiamo le procedure che realizzano le operazioni *car* e *test_atomo* sulla rappresentazione mediante record e puntatori (figg. 41 e 42). Si assume nota la dichiarazione di tipo descritta sopra.

Lasciamo al lettore la realizzazione delle altre operazioni primitive.

```
procedure car (lista: punt; var risultato: record _ lista);
{ realizza l'operazione car su una lista realizzata mediante record e puntatori; lista è il puntatore iniziale della lista; il risultato è fornito mediante un record della stessa forma utilizzata per memorizzare gli elementi della lista}
begin
  if null (lista)
  then write ('operazione non eseguibile')
  else begin
    risultato.tipo_elemento := lista^.tipo_elemento;
    risultato.val := lista^.val;
    risultato.punt_lista := lista^.punt_lista
    risultato.next := nil
  end
end;
```

Fig. 41 - La procedura car

```
function test_atomo (p: punt): boolean;
{ verifica se un elemento di una lista è un atomo o una lista }
begin
  test_atomo := p^.tipo_elemento=atomo
end;
```

Fig. 42 - La funzione test_atomo

Per la correttezza e l'efficienza della rappresentazione collegata di una lista valgono considerazioni analoghe a quelle fatte per la rappresentazione delle liste semplici.

3.6. Recupero della memoria

In questo paragrafo discutiamo un problema che sorge quando per le liste si utilizza la rappresentazione collegata. Esso è relativo alla corretta gestione del prelievo e del rilascio delle zone di memoria in cui vengono memorizzati gli elementi della lista. Abbiamo precedentemente analizzato il problema della gestione della lista libera nella rappresentazione collegata mediante array. Abbiamo anche visto che, quando la rappresentazione collegata viene realizzata mediante record e puntatori, il programmatore dispone di istruzioni (*new* e *dispose*, nel caso del Pascal) per ottenere e restituire i record utilizzati per la

memorizzazione di elementi della lista. In entrambi i casi può essere necessario richiedere spazio di memoria per la memorizzazione di ulteriori elementi della lista. È importante, perciò, disporre di un meccanismo che consenta di recuperare eventuale spazio di memoria che non è più utilizzato dalla lista. Questa operazione prende il nome di *garbage collection* (letteralmente: recupero di spazzatura).

Per risolvere questo problema si può effettuare sistematicamente l'operazione di rilascio degli elementi non più utilizzati (mediante restituzione alla lista libera, o mediante l'istruzione *dispose*). In questo caso si dice che il recupero della memoria viene fatto *on the fly* (al volo).

In alternativa, il recupero delle memoria può essere effettuato *post mortem*, cioè al momento in cui si richiede spazio di memoria, ma esso non è più disponibile. In queste condizioni, un possibile metodo per realizzare la garbage collection è basato su un algoritmo chiamato *mark and sweep* (marca e raccogli), che si compone di tre fasi fondamentali:

1. fase di marcatura, in cui vengono marcate le componenti della memoria dove sono memorizzati elementi facenti attualmente parte della lista (o delle liste) gestite dal programma;
2. fase di recupero, in cui le componenti della memoria che non sono state marcate nella fase 1 vengono raccolte ed inserite nella lista libera;
3. fase di ripristino, in cui si eliminano le marche, in modo che l'algoritmo possa essere riapplicato in modo corretto.

Nel caso in cui la lista libera sia gestita dal programma, quest'ultimo è, ovviamente, il responsabile della effettuazione della garbage collection; quindi l'algoritmo deve essere attivabile esplicitamente dal programma. Al contrario, nel caso in cui la lista libera sia gestita automaticamente dal traduttore del linguaggio (in Pascal questo è il caso in cui si utilizzino le istruzioni *new* e *dispose* per la richiesta ed il rilascio di record), l'algoritmo di garbage collection viene effettuato in modo completamente trasparente per il programmatore.

4. GLI INSIEMI

Il tipo astratto *insieme* è già stato illustrato nel par. 1. Riportiamo di seguito la sua definizione.

Definizione. L'insieme è un tipo astratto di dati $\langle S, F, C \rangle$, dove S è $\{ins,$

*boolean, V}, F è {*test_vuoto*, *inserisci*, *cancella*, *test_appartenenza*}, e C è {*insieme_vuoto*}. *ins* è il dominio di interesse. Ogni elemento di *ins* rappresenta un insieme di tipo V, che viene detto tipo base. Il dominio ed il codominio delle operazioni in F sono:*

test_insieme_vuoto: *ins* → *boolean*
inserisci: *ins* × V → *ins*
cancella: *ins* × V → *ins*
test_appartenenza: *ins* × V → *boolean*

Per il significato di tali operazioni, si rimanda al par. 1. Infine, la costante *insieme_vuoto* rappresenta l'insieme che non contiene elementi.

A partire dalle operazioni primitive, si può facilmente giungere alla definizione di altre operazioni tipiche sugli insiemi, quali l'unione, l'intersezione e la differenza di due insiemi:

unione: *ins* × *ins* → *ins*
intersezione: *ins* × *ins* → *ins*
differenza: *ins* × *ins* → *ins*

Ricordiamo che l'unione di due insiemi A e B è l'insieme composto dagli elementi che appartengono ad A oppure a B. L'intersezione di A e B è, invece, l'insieme composto dagli elementi che appartengono sia ad A che a B. Infine, la differenza tra A e B è l'insieme composto dagli elementi di A che non sono elementi di B.

Un tipo astratto di dati simile all'insieme è il *multiinsieme*. Ogni valore del tipo multiinsieme è una collezione di elementi, non necessariamente distinti tra loro. Ad esempio, {5, 3, 5, 8, 3, 12} è un valore di tipo multiinsieme di interi. Le operazioni primitive sui multiinsiemi sono analoghe a quelle definite sugli insiemi. La differenza fondamentale è che l'inserimento di un elemento V in un multiinsieme S viene effettuato anche se S contiene V come elemento. I metodi di rappresentazione per i multiinsiemi sono naturali estensioni dei metodi per la rappresentazione di insiemi, che ci accingiamo ad analizzare.

Alcuni linguaggi di programmazione consentono di utilizzare direttamente il tipo insieme nei programmi. In Pascal, ad esempio, si possono definire insiemi mediante la primitiva *set*. I valori di un tipo così definito sono insiemi di elementi appartenenti ad un tipo base. Esistono, tuttavia, alcune limitazioni all'uso di valori di tipo *set*, quali:

1. il tipo base deve essere un tipo ordinale;
2. il numero di elementi che compongono l'insieme dei valori del tipo base non

deve superare un valore massimo prefissato (in genere 128).

Queste limitazioni fanno sì che la primitiva `set` venga utilizzata solo in casi particolarmente semplici. Nel prossimo paragrafo analizziamo due metodi di rappresentazione più generali per il tipo `insieme`, uno basato sull'uso di matrici, e l'altro basato sull'uso di liste.

4.1. Metodi di rappresentazione di insiemi

Il primo metodo che presentiamo è quello che abbiamo già illustrato nel par. 1: per rappresentare il tipo `insieme` su `V` si utilizza un vettore di componenti booleane il cui tipo indice è posto in corrispondenza biunivoca con il tipo base `V`. Ogni valore del vettore rappresenta un insieme, ed è chiamato il vettore caratteristico di quell'insieme. Un vettore caratteristico `M` rappresenta un insieme `S` secondo questa regola: la componente di indice `I` del vettore vale `true` se il valore di `V` corrispondente ad `I` è un elemento di `S`, vale `false` altrimenti.

In fig. 43 mostriamo un esempio di rappresentazione di un insieme mediante vettore caratteristico, in cui si assume che il tipo base sia tipo intervallo di interi [1..10].

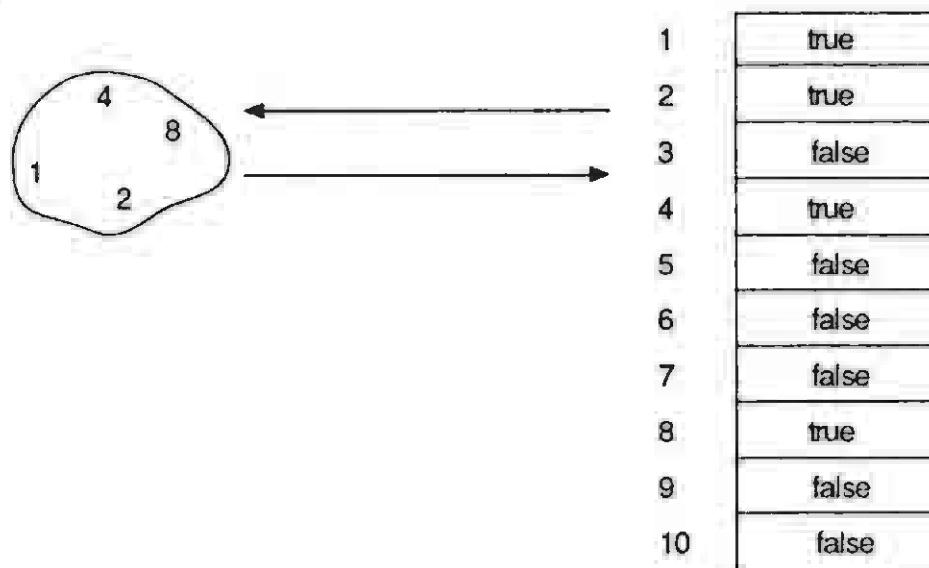


Fig. 43 - Rappresentazione mediante vettore caratteristico

Le operazioni del tipo astratto sono facilmente realizzabili in termini della rappresentazione mediante vettore caratteristico. Ad esempio, la verifica se l'insieme è vuoto viene realizzata con una scansione del vettore per verificare che il valore di tutte le sue componenti sia `false`. L'inserimento del valore `v` nell'insieme si realizza assegnando il valore `true` alla componente del vettore corrispondente a `v`. Definendo in questo modo tutte le operazioni sul tipo astratto, si ottiene una rappresentazione la cui correttezza può essere facilmente

verificata.

Riguardo all'efficienza, il principale problema di questo metodo è relativo allo spazio di memoria richiesto: infatti il numero di componenti del vettore caratteristico è pari al numero di valori del tipo base. Lo spazio di memoria richiesto per memorizzare il generico valore S del tipo insieme è quindi costante, e non dipende dal numero di elementi di S. Se il tipo base è composto da molti valori, il metodo mediante vettore caratteristico si rivela troppo costoso in termini di occupazione di memoria.

Per valutare l'efficienza rispetto al tempo di esecuzione delle operazioni, è necessario distinguere due casi fondamentali. Nel primo caso il tipo base viene direttamente utilizzato come indice del vettore caratteristico (ad esempio, in Pascal, se il tipo base è un tipo ordinale): in questa situazione, dato un elemento del tipo base, si può accedere direttamente alla corrispondente componente del vettore, e le operazioni di inserzione e cancellazione si possono realizzare efficientemente. Se, invece, il tipo base non può essere utilizzato come indice del vettore, è necessario memorizzarne esplicitamente i valori: ogni componente del vettore conterrà sia il corrispondente elemento del tipo base, sia il valore booleano che indica se tale elemento è presente o no nell'insieme rappresentato. In questo caso, le operazioni tipiche sugli insiemi (quali *inserisci*, *cancella*, ecc.) richiedono dapprima la scansione del vettore per individuare la componente cercata, e poi l'esecuzione delle opportune azioni. È evidente che la rappresentazione mediante vettore caratteristico è, in questi casi, molto inefficiente, sia dal punto di vista dell'occupazione di memoria che del tempo di esecuzione delle operazioni.

Un insieme può essere rappresentato anche mediante una lista semplice: il generico insieme S viene rappresentato mediante la lista che contiene ogni elemento di S una ed una sola volta.

Tutte le operazioni sul tipo astratto sono facilmente esprimibili mediante operazioni sulle liste. L'insieme vuoto è rappresentato mediante la lista vuota. Assumendo di disporre della seguente definizione di tipi:

```
type punt_insieme = ^elem_insieme;
  tipo_base = integer;
  elem_insieme = record
    info: tipo_base;
    next: punt_insieme
  end;
```

mostriamo la definizione della funzione *test_appartenenza* (fig. 44) e della

```

function test_appartenenza (a: punt_insieme;
                           elem: tipo_base): boolean
  { verifica se elem è presente nell'insieme a; a è rappresentato mediante
    lista semplice, realizzata con record e puntatori }
begin
  if test_insieme_vuoto (a)
  then test_appartenenza := false
  else if a^.info = elem
    then test_appartenenza := true
    else test_appartenenza := test_appartenenza (a^.next, elem)
end;

```

Fig. 44 - La funzione *test-appartenenza*

```

procedure inserisci(var a: punt_insieme; elem: tipo_base);
  { inserisce elem nell'insieme a; a è rappresentato mediante lista
    semplice, realizzata con record e puntatori; elem viene inserito
    in testa alla lista }
var b: punt_insieme;
begin
  if not (test_appartenenza (a, elem) )
  then begin
    new (b); b^.info := elem;
    b^.next := a; a := b
  end
end;

```

Fig. 45 - La procedura *inserisci*

procedura *inserisci* (fig. 45), che realizzano le omonime operazioni sugli insiemi.

Riguardo alla correttezza della rappresentazione mediante liste, si osservi che, dato un insieme, esiste almeno un valore di tipo lista che lo rappresenta (in particolare, diverse liste possono rappresentare lo stesso insieme, come, ad esempio, (1 4 8) e (4 8 1)). Al contrario, data una lista, è immediato risalire al corrispondente insieme. La correttezza della funzione *test_appartenenza* e della procedura *inserisci* è facilmente verificabile: in particolare, si noti che la procedura *inserisci* assicura (mediante la chiamata a *test_appartenenza*) che

nella lista non venga inserito un elemento già presente. Lasciamo al lettore la realizzazione delle altre operazioni primitive e la verifica della loro correttezza.

Dell'efficienza di questa rappresentazione e della rappresentazione mediante vettore caratteristico ci occupiamo nel prossimo paragrafo.

4.2. Confronto tra i metodi di rappresentazione di insiemi

Scopo di questo paragrafo è confrontare l'efficienza delle due rappresentazioni di insiemi mediante vettore caratteristico e mediante lista semplice.

Riguardo all'occupazione di memoria, abbiamo già osservato che la rappresentazione mediante vettore caratteristico necessita di uno spazio proporzionale al numero di elementi del tipo base V. Al contrario, nella rappresentazione mediante lista, realizzata con record e puntatori, l'occupazione di memoria è proporzionale al numero di elementi dell'insieme.

L'efficienza delle operazioni primitive si può confrontare analizzando il numero di confronti richiesti per eseguire l'operazione *test_appartenenza*. Nel caso della rappresentazione con vettore caratteristico, questa operazione può essere in generale effettuata accedendo direttamente ad una componente dell'array e, quindi, non richiede alcun confronto. Nella rappresentazione mediante lista, invece, l'operazione richiede, nel caso peggiore, l'accesso a tutti gli elementi contenuti nell'insieme.

Per tenere conto delle operazioni non primitive, analizziamo anche l'efficienza dell'operazione di unione nelle due rappresentazioni.

Nella rappresentazione mediante vettore caratteristico, l'unione può essere realizzata con un ciclo di scansione degli elementi del tipo base, come descritto nel par. 1: il numero di confronti effettuati è, in questo caso, pari al numero di elementi del tipo base.

Nella rappresentazione mediante lista l'unione di due insiemi può essere realizzata con il seguente metodo. Si copia una delle liste (diciamo la lista A) in una nuova lista (la lista C, che rappresenta il risultato). Si scandisce la seconda lista B elemento per elemento, e per ognuno di essi si controlla se è presente in A. Se l'elemento è presente, si passa al successivo elemento della lista B, se invece non è presente lo si inserisce nella lista C. Se indichiamo con n e m rispettivamente il numero di elementi della lista A e della lista B, si può verificare facilmente che, nel caso più sfavorevole, l'algoritmo richiede

$m * n$

confronti tra elementi. In particolare, il caso più sfavorevole si verifica quando i due insiemi sono disgiunti, cioè quando A e B non hanno elementi comuni. Lo stesso numero di confronti è richiesto per il calcolo dell'intersezione e della differenza.

Per migliorare l'efficienza dell'operazione, è necessario adottare una semplice modifica sulla rappresentazione degli insiemi mediante lista. La modifica è possibile quando sull'insieme base V è definito un ordinamento, e consiste nel memorizzare gli elementi dell'insieme in modo ordinato nella lista (ad esempio in ordine crescente). È chiaro che in questo modo la scansione della lista effettuata per verificare la presenza di un elemento x si può interrompere appena si trova un elemento il cui valore è maggiore o uguale a x . In generale, quindi, il numero di confronti diminuisce. D'altra parte, l'operazione di inserimento risulterà ora più complessa, in quanto si deve conservare l'ordinamento degli elementi.

Con questa nuova rappresentazione l'algoritmo per l'unione di due insiemi può essere più efficiente. Infatti, durante la scansione della lista B non è più necessario confrontare il generico elemento b di B con tutti gli elementi di A, ma solo con quelli minori di b ; inoltre, nell'analisi del successivo elemento di B si può ripartire dall'ultimo elemento analizzato di A, evitando così inutili confronti.

Per il calcolo del numero di confronti necessari all'esecuzione di questo algoritmo si può notare che, nel caso più sfavorevole, per ogni elemento b_i di B, esistono in A N_i elementi minori ed uno maggiore di b_i . In questa situazione, il numero di confronti effettuati sul generico elemento b_i è N_i+1 e, perciò, il numero totale di confronti è:

$$C = (N_1+1) + (N_2+1) + \dots + (N_m+1)$$

Considerando che

$$N_1 + N_2 + \dots + N_m = n$$

si ottiene che il numero massimo di confronti necessari per l'esecuzione dell'algoritmo è $n+m$.

Le considerazioni svolte in questo paragrafo sono riassunte nella fig. 46.

5. PILE E CODE

In questo paragrafo descriviamo le pile e le code, due tipi di dato che consentono di rappresentare sequenze di elementi in cui gli inserimenti e le

	<i>Vettore caratteristico</i>	<i>Lista semplice</i>
<i>Spazio di memoria per rappresentare un insieme di n elementi</i>	k (numero di elementi del tipo base)	n
<i>Numero di confronti per test_appartenenza su un insieme di n elementi</i>	0	n
<i>Numero di confronti per l'unione tra A (con n elementi) e B (con m elementi)</i>	k	n+m

Fig. 46 - Confronto tra le rappresentazioni di insiemi

cancellazioni vengono eseguite con particolari modalità. Daremo la definizione di pile e di code e discuteremo i più comuni metodi per la loro rappresentazione. Esempi di utilizzo di queste strutture di dati verranno presentati nei paragrafi successivi.

5.1. Le pile

Una pila è un tipo astratto che consente di rappresentare un multiinsieme di elementi in cui ogni eliminazione ha per oggetto l'elemento che è stato inserito per ultimo. Questa disciplina di gestione di un insieme di elementi è spesso chiamata *lifo*, abbreviazione di *Last In, First Out*, il cui significato è “l'ultimo entrato è il primo ad uscire”. Possiamo perciò definire una pila come un multiinsieme gestito con la disciplina *lifo*.

Come esempio di utilizzo di questa struttura di dati si pensi alla esecuzione di un programma Pascal, in cui viene attivata un'unità, che può, a sua volta, attivarne altre. Come abbiamo visto nel cap. 2, le informazioni sulle unità e sulle istruzioni di ritorno vengono gestite mediante una pila: l'unità a cui bisogna tornare una volta conclusa l'esecuzione di una procedura è sempre

l'ultima memorizzata.

La definizione del tipo astratto "pila di elementi di tipo *elemento*" è la seguente:

Definizione. La pila è un tipo astratto di dati $\langle S, F, C \rangle$, dove:

- $S = \{pila, elemento, boolean\}$, con *pila* il dominio di interesse.
- $F = \{top, push, pop, test_pila_vuota\}$, dove:

top : *pila* \rightarrow *elemento*
push : *pila* \times *elemento* \rightarrow *pila*
pop : *pila* \rightarrow *pila*
test_pila_vuota : *pila* \rightarrow *boolean*

- $C = \{pila_vuota\}$

Il significato delle operazioni è il seguente: *top* si applica ad una pila non vuota e restituisce l'elemento che era stato inserito per ultimo (detto "elemento affiorante"). *push* si applica ad una pila e ad un valore appartenente all'insieme *elemento*: il suo effetto è di inserire tale valore nella pila. *pop* si applica ad una pila *p* non vuota e restituisce la pila ottenuta da *p* eliminando l'ultimo elemento che era stato inserito. Infine, *test_pila_vuota* verifica se una pila è vuota. Si noti la differenza tra l'operazione *top* e l'operazione *pop*: la prima restituisce il valore dell'elemento affiorante, senza eliminarlo dalla pila, mentre la seconda restituisce come valore la pila che si ottiene estraendo tale elemento.

La disciplina di gestione *lifo* richiede che nella rappresentazione di una pila si debba conservare memoria dell'ordine in cui gli elementi vengono inseriti. Per rappresentare una pila si può quindi utilizzare una lista, facendo in modo che la sequenza con cui si susseguono gli elementi nella lista rifletta l'ordine di inserzione degli elementi nella pila. Riguardo alle operazioni, si può stabilire un semplice parallelo tra le operazioni definite sui due tipi astratti di dato, come illustrato in fig. 47.

Operazioni sulle pile	Operazioni sulle liste
<i>push</i>	<i>cons</i>
<i>pop</i>	<i>cdr</i>
<i>top</i>	<i>car</i>
<i>test_pila_vuota</i>	<i>null</i>

Fig. 47 - Corrispondenza tra le operazioni sulle pile e sulle liste

Per le pile, quindi, si può utilizzare uno dei metodi che abbiamo descritto per la rappresentazione delle liste. Analizziamo, in particolare, la rappresentazione sequenziale, e la rappresentazione mediante record e puntatori, e vedremo come esse si possono adattare al caso in cui la lista è utilizzata per la rappresentazione di una pila.

Nella rappresentazione sequenziale si utilizza una variabile di tipo **array**: l'elemento affiorante è memorizzato in posizione N dell'array, il penultimo in posizione N-1, e così via fino al primo elemento inserito, che è memorizzato in posizione 1. In un'ulteriore variabile si memorizza l'indice dell'elemento affiorante. Il valore 0 per tale variabile indica che la pila è vuota. Si può notare che la rappresentazione descritta è più semplice dalla corrispondente rappresentazione per le liste: la caratteristica delle pile di consentire l'eliminazione del solo elemento affiorante, permette di memorizzare sempre gli elementi presenti nella pila nelle posizioni dell'array comprese tra la prima e la N-esima.

Graficamente possiamo raffigurare l'uso della pila rappresentata mediante array nel modo descritto in fig. 48.

La figura mostra come l'array venga riempito o svuotato sempre da uno stesso lato, e come gli inserimenti e le eliminazioni avvengano mediante l'aggiornamento del valore della variabile *posizione_top*.

Come esempio, mostriamo la realizzazione dell'operazione *push*, assumendo la seguente dichiarazione di tipi per la rappresentazione sequenziale delle pile.

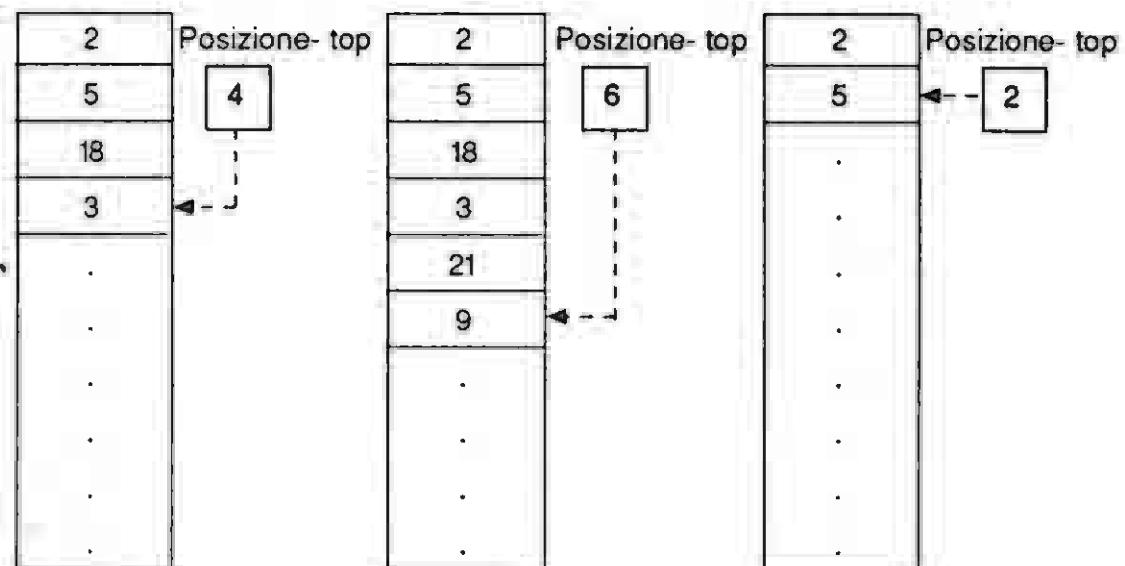


Fig. 48 - Una pila rappresentata come *array*

```

const nmax_array_pila = ....; { numero di componenti dell'array utilizzato
                                per rappresentazione della pila }

type tipo_elementi = ....; { tipo degli elementi della pila }
    tipo_pila = record
        elementi: array [1..nmax_array_pila]
                        of tipo_elementi;
        posizione_top: integer
    end;

procedure push (var p: tipo_pila; e: tipo_elementi );
{ effettua l'operazione push sulla pila p rappresentata mediante array }
begin
    if p.posizione_top = nmax_array_pila
    then writeln('insufficiente dimensione dell'array')
    else begin
        p.posizione_top := p.posizione_top + 1;
        p.elementi[p.posizione_top] := e
    end
end;

```

Analogamente al caso delle liste, la rappresentazione mediante array presenta due svantaggi: da un lato essa richiede di determinare a priori un limite al numero massimo di elementi della pila, dall'altro lo spazio di memoria utilizzato per memorizzare la pila è costante e non dipende dal numero dei suoi elementi. Si noti però che, al contrario del caso delle liste, gli inserimenti e le cancellazioni non potranno mai richiedere spostamenti di elementi, perché, tali operazioni si effettuano solo ad una estremità dell'array.

Il secondo metodo di rappresentazione che analizziamo fa uso di record e puntatori. Esso è analogo al caso di rappresentazione collegata di una lista, con l'avvertenza di utilizzare la disciplina *lifo* per inserimenti e cancellazioni. Un modo semplice ed efficace di realizzare tale disciplina è quello di effettuare l'inserimento e l'eliminazione di elementi in testa alla lista. In questo modo, il puntatore iniziale della lista punterà sempre all'elemento affiorante, se la pila non è vuota; in caso contrario, il suo valore sarà *nil*.

Le operazioni definite sulle pile si realizzano molto facilmente analizzando la corrispondenza illustrata in fig. 47. Come esempio, presentiamo in fig. 49 la

procedura che realizza l'operazione *pop*. Si assume nota alla procedura la seguente dichiarazione di tipi:

```
type tipo_elementi = ....; { tipo degli elementi della pila }
  punt_pila= ^record_pila;
  record_pila = record
    elemento: tipo_elementi;
    next: punt_pila
  end;
```

```
procedure pop (var p: punt_pila );
  { effettua l'operazione pop sulla pila p rappresentata mediante record e
  puntatori }
  var q: punt_pila;
begin
  if test_pila_vuota(p)
  then writeln('operazione non applicabile')
  else begin
    q := p;
    p := p^.next;
    dispose(q)
  end
end;
```

Fig. 49 - La procedura *pop*

5.2. Le code

Una coda è un tipo astratto che consente di rappresentare un multiinsieme di elementi in cui ogni eliminazione ha per oggetto l'elemento che è stato inserito per primo. Questa disciplina di gestione è chiamata *fifo*, abbreviazione di *First In, First Out*, il cui significato è "il primo entrato è il primo ad uscire". Tale disciplina è tipica di molte situazioni della vita di tutti i giorni: agli sportelli degli uffici, ad esempio, il primo ad essere servito è il primo della coda.

La definizione del tipo astratto "coda di elementi di tipo *elemento*" è la seguente:

Definizione. La coda è un tipo astratto di dati <S,F,C>, dove:

- S = {*coda, elemento, boolean* }, con *coda* dominio di interesse.

- $F = \{primo, in_coda, out_coda, test_coda_vuota\}$, dove:
 - $primo : coda \rightarrow elemento$
 - $in_coda : coda \times elemento \rightarrow coda$
 - $out_coda : coda \rightarrow coda$
 - $test_coda_vuota : coda \rightarrow boolean$
- $C = \{coda_vuota\}$

Il significato delle operazioni è il seguente: *primo* si applica ad una coda non vuota e fornisce come risultato l'elemento che, tra quelli presenti nella coda, è stato inserito per *primo*. *in_coda* si applica ad una coda e ad un valore appartenente all'insieme *elemento*: il suo effetto è di inserire tale elemento nella coda. *out_coda* si applica ad una coda non vuota e fornisce come risultato la coda ottenuta da c eliminando l'elemento che, tra quelli presenti nella coda, era stato inserito per *primo*. Infine, *test_coda_vuota* verifica se una coda è vuota.

In linea generale, le possibili rappresentazioni delle code sono analoghe a quelle delle pile, con la differenza che nel caso delle code è conveniente consentire l'accesso sia all'elemento che è stato inserito per *primo*, perché esso sarà il primo ad essere eliminato, sia all'elemento che è stato inserito per *ultimo*, al fine di effettuare correttamente ed efficientemente l'inserimento dei successivi elementi.

Per le code, la rappresentazione sequenziale non è così agevole come per le pile. In questo caso la rappresentazione è molto simile a quella per le liste: l'array è gestito in modo circolare, e si utilizzano due variabili, che chiamiamo *primo* e *ultimo*. Il valore di *primo* indica la posizione dell'array in cui è memorizzato l'elemento inserito per *primo*, mentre il valore di *ultimo* si riferisce all'ultimo elemento inserito. Le operazioni *in_coda* e *out_coda* sono realizzate in modo che le condizioni suddette valgano sempre. Il valore 0 per *primo* e *ultimo* indica che la coda è vuota, mentre la condizione in cui *primo* = *ultimo* + 1 (dove la somma è eseguita modulo N, con N pari al numero di componenti dell'array) indica che l'array è completamente occupato dagli elementi della coda.

Nella rappresentazione collegata mediante record e puntatori si fa generalmente uso di due variabili di tipo puntatore. La prima (che chiamiamo *primo*) contiene il puntatore all'elemento in testa alla coda, la seconda (che chiameremo *ultimo*) contiene il puntatore all'elemento che si trova alla fine della coda. La condizione di coda vuota sarà indicata dal valore *nil* per entrambe queste variabili. Si veda la rappresentazione grafica di quanto detto in fig. 50.

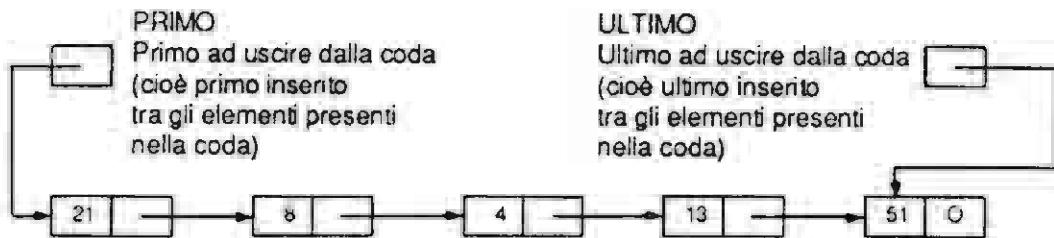


Fig. 50 - Rappresentazione di una coda mediante lista

Si può notare che l'uso della variabile *ultimo* consente di evitare la scansione di tutti i record al momento dell'inserimento di un nuovo elemento. Mostriamo ora le procedure *in_coda* e *out_coda* nel caso di rappresentazione mediante record e puntatori (figg. 51 e 52). Si assuma nota la seguente dichiarazione di tipi:

```
type tipo_elementi= ....; { tipo degli elementi della coda }
punt_coda = ^record_coda;
record_coda = record
    elemento: tipo_elementi;
    next: punt_coda
end;
```

```
procedure in_coda (var p,u: punt_coda; e: tipo_elementi );
{ inserisce l'elemento e in una coda; la coda è rappresentata mediante
record e puntatori; p è il puntatore al primo elemento della coda, u
all'ultimo }
begin
  if test_coda_vuota(p)
  then begin
    new(p);
    u := p
  end
  else begin
    new(u^.next);
    u := u^.next
  end;
  u^.elemento := e;
  u^.next := nil
end;
```

Fig. 51 - La procedura *in_coda*

```

procedure out_coda (var p,u: punt_coda );
  { elimina un elemento dalla coda; la coda è rappresentata mediante
    record e puntatori; p è il puntatore al primo elemento della coda, u
    all'ultimo }
var aux: punt_coda;
begin
  if test_coda_vuota(p)
  then write('coda vuota: operazione non applicabile' )
  else begin
    aux := p;
    p := p^.next;
    if p = nil
    then u := nil;
    dispose(aux)
  end
end;

```

Fig. 52 - La procedura *out coda*

Si noti che nel caso di inserimento in una coda vuota, oltre al puntatore all'ultimo elemento della coda deve essere aggiornato anche il puntatore al primo: il nuovo elemento sarà, infatti, contemporaneamente il primo e l'ultimo. Analogamente, l'eliminazione di un elemento, qualora questo sia l'unico nella coda, provoca l'assegnazione del valore *nil* ad entrambe le variabili.

Riguardo al confronto tra i due metodi che abbiamo descritto per la rappresentazione delle code, valgono considerazioni analoghe a quelle svolte per la rappresentazione delle pile.

Un'ulteriore rappresentazione collegata per le code è la rappresentazione circolare, che si ottiene ponendo nel record corrispondente all'ultimo elemento della coda il puntatore al primo elemento. In questo modo si può evitare l'utilizzo della variabile *primo*, perché al primo elemento della coda si può accedere mediante il puntatore *ultimo*. Se la coda contiene un solo elemento, il record corrispondente contiene un puntatore a se stesso; infine, se la coda è vuota, il valore di *ultimo* è *nil*. Lasciamo al lettore il compito di realizzare le operazioni primitive sulle code per questa rappresentazione.

6. GLI ALBERI

L'albero è un tipo astratto di dati utilizzato per rappresentare relazioni gerarchiche tra oggetti. Nel par. 3 dell'appendice vengono introdotti i concetti di grafo e albero con le relative definizioni. Ad esse faremo riferimento in questo paragrafo. Tratteremo dapprima degli alberi binari, cioè di alberi ordinati in cui ogni nodo ha al massimo due figli, illustrando le loro proprietà ed analizzando i metodi per la loro rappresentazione. Successivamente ci occuperemo degli alberi N-ari, cioè alberi in cui ogni nodo può avere un numero qualunque di figli.

In fig. 53 mostriamo un esempio di albero. In essa compaiono alcuni termini che spesso si utilizzano per descrivere gli alberi. Per la loro definizione, si rimanda all'appendice.

L'albero mostrato in figura è un albero sorgente, perché la radice è un nodo sorgente (ha solo archi uscenti). Le considerazioni che svolgeremo in questa sezione si riferiscono ad alberi sorgente; esse sono, tuttavia, facilmente adattabili al caso di altri tipi di albero.

6.1. Gli alberi binari

Gli *alberi binari* sono alberi in cui ogni nodo ha al massimo due figli. Sui figli di ogni nodo è definito un ordinamento (gli alberi binari sono alberi ordinati): in genere, il figlio che viene prima nell'ordinamento è detto figlio sinistro, mentre l'altro è detto figlio destro.

Definizione. L'albero binario è un tipo astratto di dati $\langle S, F, C \rangle$, dove:

- $S = \{ \text{alb_bin}, \text{nodo}, \text{boolean} \}$, con *alb_bin* dominio di interesse;
- $F = \{ \text{radice}, \text{sinistro}, \text{destro}, \text{costruisci}, \text{test_albero_vuoto} \}$, dove:
 - $\text{test_albero_vuoto}: \text{alb_bin} \rightarrow \text{boolean}$
 - $\text{costruisci}: \text{alb_bin} \times \text{nodo} \times \text{alb_bin} \rightarrow \text{alb_bin}$
 - $\text{radice}: \text{alb_bin} \rightarrow \text{nodo}$
 - $\text{sinistro}: \text{alb_bin} \rightarrow \text{alb_bin}$
 - $\text{destro}: \text{alb_bin} \rightarrow \text{alb_bin}$
- $C = \{ \text{albero_vuoto} \}$

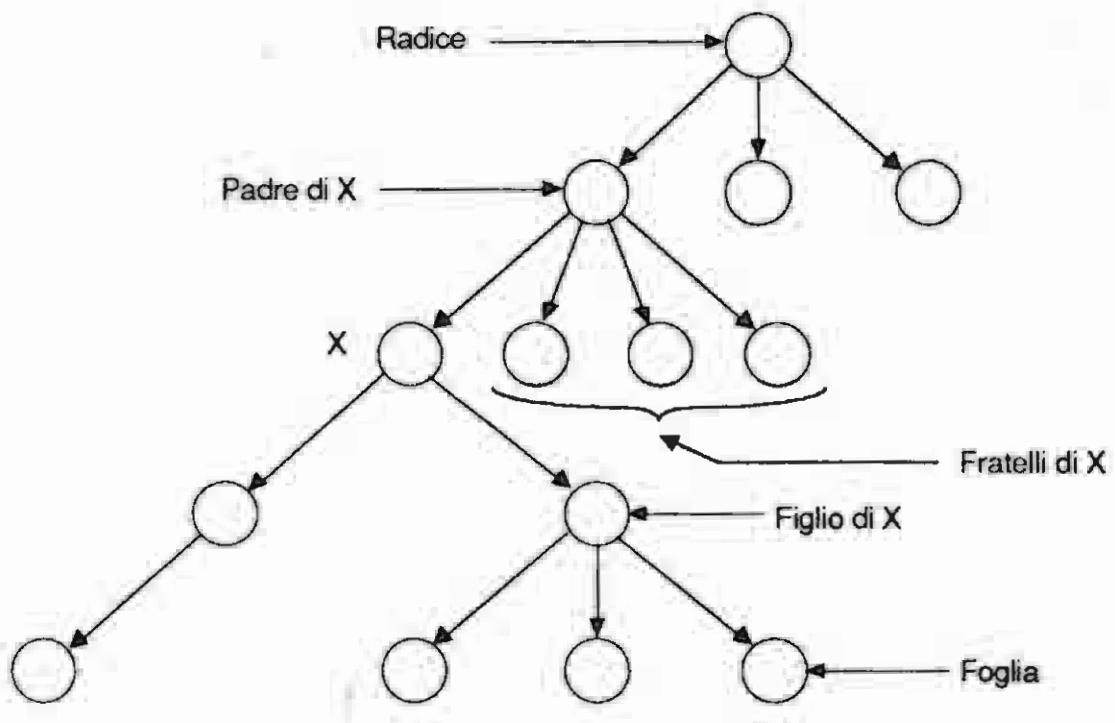


Fig. 53 - Un albero

Per descrivere in maniera induttiva gli elementi del dominio *alb_bin*, si fa spesso riferimento alla seguente definizione:

Definizione. Un albero binario è un grafo orientato che o è vuoto (cioè ha un insieme vuoto di nodi) o è formato da un nodo N (detto la radice) e da due sottoalberi, che sono a loro volta alberi binari, e che vengono chiamati rispettivamente sottoalbero sinistro e sottoalbero destro.

Un esempio di albero binario è mostrato in fig. 54.

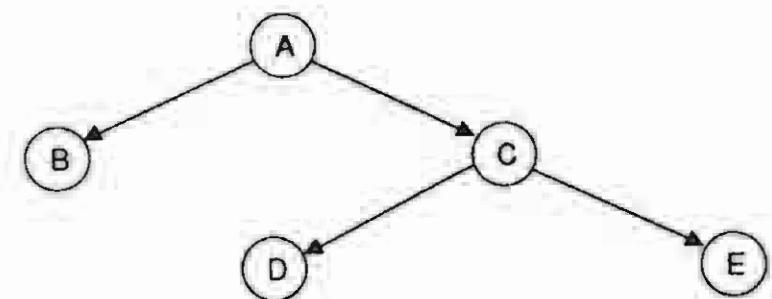


Fig. 54 - Un esempio di albero binario

Descriviamo ora il significato delle operazioni primitive. L'operazione *test_albero_vuoto* si applica ad un albero binario e restituisce *true* se l'albero è vuoto, *false* altrimenti. L'operazione *costruisci* serve a costruire un albero binario: *costruisci* (S,E,D) restituisce l'albero formato dalla radice E, dal sottoalbero sinistro S e dal sottoalbero destro D. *radice* si applica ad un albero non vuoto e restituisce il valore associato alla radice. *sinistro* e *destro* si applicano ad un albero binario, e sono definite solo quando tale albero non è vuoto. *sinistro* restituisce il sottoalbero sinistro dell'albero binario, mentre *destro* restituisce il suo sottoalbero destro.

Oltre alle operazioni citate, per gli alberi in genere e per gli alberi binari in particolare, si definiscono i cosiddetti "algoritmi di visita", cioè algoritmi che consentono di analizzare tutti i nodi dell'albero in un determinato ordine. I più comuni algoritmi di visita per alberi binari sono tre, visita in preordine, in postordine e simmetrica.

La *visita in preordine* (detta anche visita in ordine anticipato) si applica ad un albero non vuoto e richiede dapprima l'analisi della radice dell'albero, e, successivamente, la visita, effettuata con lo stesso metodo, dei due sottoalberi, prima il sinistro, e poi il destro. L'algoritmo può essere formulato ricorsivamente nel seguente modo:

visita in preordine l'albero binario T:

se test_albero_vuoto (T)=false

allora esegui

analizza radice (T);

visita in preordine l'albero binario sinistro(T);

visita in preordine l'albero binario destro(T)

fine

Come esempio, si consideri l'albero di fig. 54. Effettuando la visita in preordine di tale albero, i nodi vengono analizzati nella sequenza:

A B C D E

La *visita in postordine* si applica ad un albero non vuoto, e richiede dapprima la visita, effettuata con lo stesso metodo, dei sottoalberi (prima il sinistro e poi il destro) e, successivamente, l'analisi della radice dell'albero. L'algoritmo può essere così formulato:

visita in postordine l'albero binario T:

se test_albero_vuoto(T) = false

allora esegui

visita in postordine l'albero binario sinistro(T);

visita in postordine l'albero binario destro(T);

analizza radice(T)

fine

Ad esempio, visitando in postordine l'albero di fig. 54, i nodi vengono analizzati nella sequenza:

B D E C A

La *visita simmetrica*, invece, richiede dapprima la visita del sottoalbero sinistro effettuata (sempre con lo stesso metodo), poi l'analisi della radice, e poi la visita del sottoalbero destro:

visita in ordine simmetrico l'albero binario T:

se test_albero_vuoto(T) = false

allora esegui

visita in ordine simmetrico l'albero binario sinistro(T);

analizza radice(T);

visita in ordine simmetrico l'albero binario destro(T)

fine

La sequenza di nodi analizzati eseguendo la visita simmetrica dell'albero di fig. 54 è:

B A D C E

Nei prossimi due paragrafi descriviamo alcuni metodi per la rappresentazione di alberi binari. Successivamente, nel par. 6.4, presentiamo un'applicazione interessante di questa struttura di dati.

6.2. La rappresentazione sequenziale

Nella rappresentazione sequenziale degli alberi binari ogni valore del tipo “albero binario” viene rappresentato mediante un valore di tipo **array** ad una dimensione. Le componenti dell’array sono dello stesso tipo dei nodi dell’albero, mentre gli indici dell’array sono interi. La corrispondenza tra valore di

tipo albero binario e valore di tipo **array** è data dalle seguenti regole:

- la radice dell'albero è memorizzata in prima posizione dell'array;
- se un nodo N dell'albero è memorizzato in posizione i, allora il figlio sinistro di N, se esiste, è memorizzato in posizione $2*i$, ed il figlio destro di N, se esiste, è memorizzato in posizione $2*i+1$.

Come esempio, si veda la fig. 55, in cui si mostra un albero binario ed il corrispondente array che ne realizza la rappresentazione sequenziale.

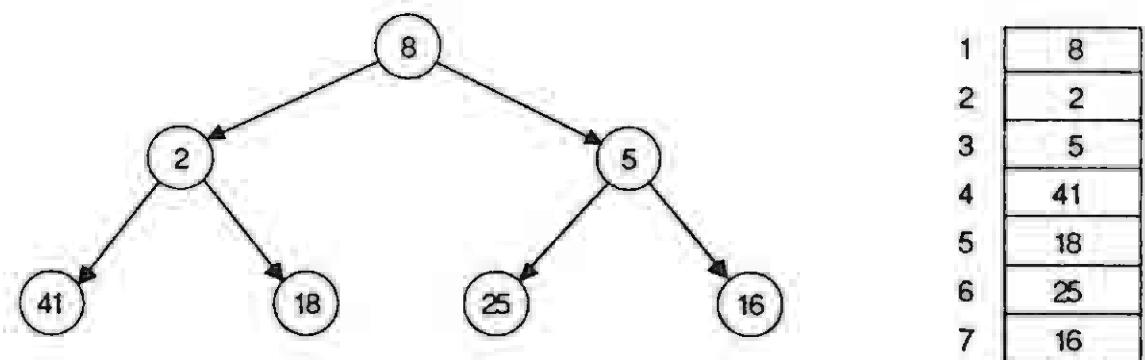


Fig. 55 - Rappresentazione sequenziale di un albero binario completo

L'albero binario di fig. 55 ha la caratteristica che ogni suo nodo che non sia una foglia ha due figli, e tutte le sue foglie sono allo stesso livello. Un albero del genere viene detto "completo".

La rappresentazione mediante array è particolarmente adatta per alberi binari completi, mentre per alberi non completi rivela diversi problemi, che nel seguito analizziamo.

In fig. 56 compare un albero binario non completo e la corrispondente rappresentazione sequenziale.

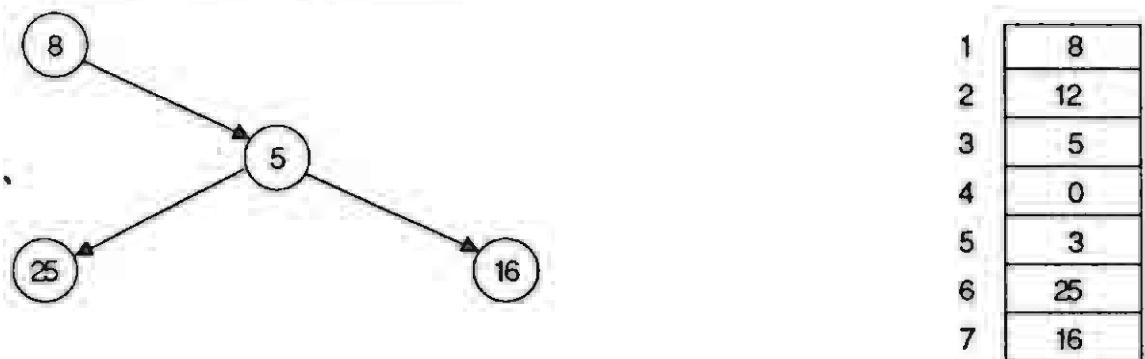


Fig. 56 - Rappresentazione sequenziale di un albero binario non completo

Si può facilmente notare che alcune componenti dell'array non corrispondono ad alcun nodo dell'albero: ad esempio il valore in posizione 2, posizione

riservata all'eventuale figlio sinistro della radice, non corrisponde ad alcun nodo dell'albero, così come i valori memorizzati in posizione 4 e 5, riservati ai figli del figlio sinistro della radice. Al contrario, in posizione 3, 6 e 7 compaiono nodi significativi dell'albero. Per rendere corretta la rappresentazione, dobbiamo quindi prevedere un meccanismo non ambiguo che indichi in quali componenti dell'array sono rappresentati effettivamente i nodi dell'albero. Un possibile metodo potrebbe essere quello di memorizzare un valore particolare, che non è tra i valori possibili per i nodi dell'albero, nelle componenti che non corrispondono a nodi dell'albero. Un secondo metodo, più generale, è quello di associare ad ogni componente dell'array un campo di tipo booleano, che vale *true* se nella componente è effettivamente memorizzato un nodo dell'albero, *false* altrimenti. Si veda in fig. 57 come si modifica la rappresentazione di fig. 56 se si adotta quest'ultimo metodo.

1	<i>true</i>	8
2	<i>false</i>	12
3	<i>true</i>	5
4	<i>false</i>	0
5	<i>false</i>	3
6	<i>true</i>	25
7	<i>true</i>	16

Fig. 57 - Modifica della rappresentazione di fig. 56

Le operazioni definite per il tipo astratto albero binario sono facilmente realizzabili in termini della rappresentazione sequenziale. Lasciamo al lettore la loro realizzazione, ed analizziamo vantaggi e svantaggi di tale rappresentazione rispetto all'efficienza.

È immediato verificare che alberi binari non completi vengono rappresentati con spreco di memoria che, nelle condizioni più sfavorevoli, può diventare molto gravoso. In più, la rappresentazione sequenziale degli alberi binari presenta i tipici problemi di una rappresentazione sequenziale: impone a priori un limite massimo per il numero di nodi dell'albero, e le operazioni di aggiunta ed eliminazione di nodi o di sottoalberi, comportano in generale diversi spostamenti nell'array.

In compenso, alcune operazioni possono essere effettuate molto semplicemente ed efficientemente, come, ad esempio, l'accesso ai figli e l'accesso al padre di un nodo di cui è nota la posizione nell'array.

6.3. La rappresentazione mediante lista

Il metodo di rappresentazione che descriviamo in questo paragrafo è basato su una semplice corrispondenza tra il tipo "albero binario" ed il tipo "lista". Ogni valore T del tipo albero binario può essere rappresentato mediante un valore di tipo lista nel seguente modo:

- se T è vuoto, la lista che lo rappresenta è la lista vuota;
- se T non è vuoto, la lista che lo rappresenta è formata da tre elementi: il primo è un atomo, e rappresenta la radice di T ; il secondo elemento è una lista che rappresenta, con lo stesso metodo, il sottoalbero sinistro di T ; il terzo è un'altra lista, che rappresenta il sottoalbero destro di T .

Mostriamo, in fig. 58, la rappresentazione mediante lista dell'albero di fig. 56. La lista è descritta mediante la rappresentazione parentetica.

(8 () (5 (25 () ()) (16 () ()))) .

Fig. 58 - Rappresentazione di un albero binario mediante lista

Avendo stabilito una corrispondenza tra alberi binari e liste, è ovvio che tutti i metodi di rappresentazione delle liste possono essere utilizzati per la rappresentazione di alberi binari. Tuttavia, analizzando la peculiarità degli alberi binari, si giunge alla definizione di metodi di rappresentazione più specifici e, perciò, più efficienti. Essi tengono conto del fatto che le liste che rappresentano alberi binari o sono liste vuote, oppure sono composte di soli tre elementi: un atomo e due liste.

Un primo metodo richiede di utilizzare un array, in modo che ad ogni nodo dell'albero corrisponda una componente dell'array, in cui sono memorizzate tre informazioni, quella associata al nodo dell'albero, il riferimento al figlio sinistro (se esiste), ed il riferimento al figlio destro (se esiste). Il riferimento al figlio sinistro (o destro) è il valore dell'indice in corrispondenza del quale si trova la componente che rappresenta il figlio sinistro (o destro). Se il figlio non esiste, per il riferimento si sceglie un valore particolare, generalmente 0. Viene poi utilizzata una variabile per memorizzare il riferimento iniziale dell'albero, cioè il valore dell'indice della componente che rappresenta la radice dell'albero. In fig. 59 mostriamo la rappresentazione collegata mediante array dell'albero binario di fig. 56. Nella variabile *inizio* è memorizzato il riferimento iniziale dell'albero.

Il secondo metodo di rappresentazione collegata prevede di utilizzare record

inizio

	<i>sin</i>	<i>nodo</i>	<i>des</i>
1	0	16	0
2	42	32	51
3	0	8	7
4	48	57	65
5	0	25	0
6	64	70	48
7	5	5	1
8	55	60	59
9	49	47	77
10	61	52	60

Fig. 59 - Rappresentazione collegata di un albero binario

e puntatori. Poichè ogni nodo dell'albero ha al massimo due figli, il tipo **record** che si utilizza per rappresentare il singolo nodo dell'albero viene definito con tre campi, uno per l'informazione associata al nodo, uno per il puntatore al sottoalbero sinistro, ed uno per il puntatore al sottoalbero destro. Il valore *nil* per un campo puntatore indica che il corrispondente sottoalbero è vuoto. Al record che rappresenta la radice si accede mediante una variabile di tipo puntatore. Nella fig. 60 compare la rappresentazione grafica di un albero rappresentato mediante record e puntatori.

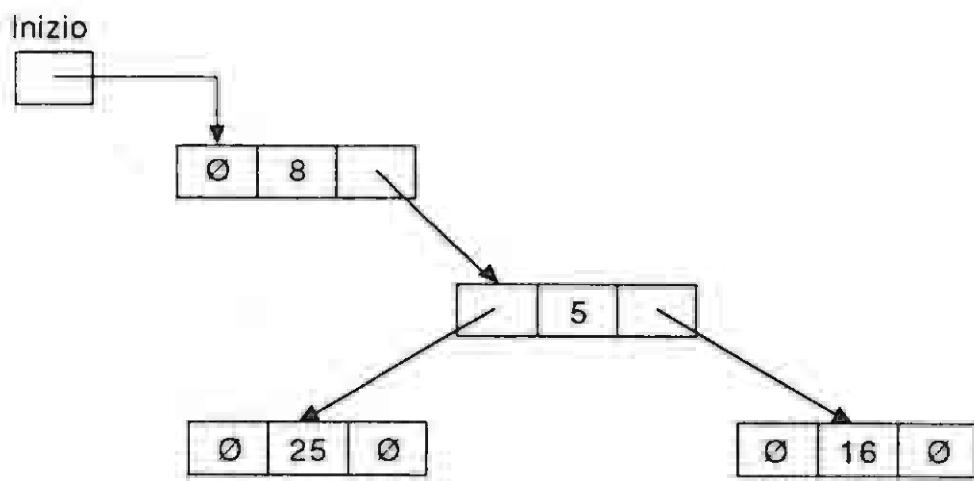


Fig. 60 - Rappresentazione mediante record e puntatori

Una possibile dichiarazione di tipi in Pascal per la rappresentazione di un albero binario mediante record e puntatori è:

```
type tipo_nodo = ...;
punt_albero = ^record_albero;
record_albero= record
    info: tipo_nodo;
    sin, des: punt_albero
end;
```

Basandoci su tale dichiarazione, presentiamo in fig. 61 la realizzazione dell'operazione primitiva *sinistro*.

```
function sinistro(alb: punt_albero): punt_albero;
{ realizza l'operazione sinistro su un albero binario rappresentato
mediante record e puntatori; alb è il puntatore iniziale dell'albero }
begin
    if test_albero_vuoto(alb)
        then write('operazione non applicabile')
        else sinistro := alb^.sin
end;
```

Fig. 61 - La funzione *sinistro*

Come esempio di realizzazione di una operazione non primitiva, presentiamo la visita in preordine di un albero binario. Una realizzazione semplice ed immediata di tale algoritmo è quella che fa uso della ricorsione. In fig. 62 mostriamo la procedura ricorsiva per la visita in preordine di un albero binario, effettuata per stampare il valore dei suoi nodi.

Maggiori difficoltà presenta la versione iterativa di questo algoritmo. Il criterio che in genere si adotta per realizzarla è analogo alla visita iterativa di una lista semplice: si utilizza una variabile puntatore il cui valore indichi il nodo che di volta in volta viene analizzato (nodo corrente). La difficoltà nasce dal fatto che, seguendo i puntatori secondo l'ordine stabilito dalla visita in preordine, si deve tenere traccia dei nodi via via analizzati, al fine di riprendere la visita dopo aver raggiunto una foglia. Per questo scopo, possiamo memorizzare i puntatori dei sottoalberi destri (non vuoti) dei nodi che via via vengono analizzati, in modo che la visita possa riprendere correttamente una volta raggiunta una foglia. Considerando l'ordine con cui i nodi vengono analizzati

```

procedure preordine(alb: punt_albero);
  { effettua la visita in predordine di un albero binario per stampare i valori
    dei nodi; alb è il puntatore iniziale dell'albero }

begin
  if not( test_albero_vuoto(alb) )
  then begin
    { analizza la radice }
    write(alb^.info);
    { visita in preordine il sottoalbero sinistro }
    preordine(sinistro(alb));
    { visita in preordine il sottoalbero destro }
    preordine(destro(alb))
  end
end;

```

Fig. 62 - La procedura ricorsiva *preordine*

nella visita in preordine, è ovvio che la struttura più adeguata per memorizzare tali puntatori è la pila: infatti, il nodo da cui occorre riprendere la visita una volta che non si può accedere ad un sottoalbero sinistro, è quello il cui puntatore è stato memorizzato per ultimo. La pila è usata in modo che, di volta in volta, l'elemento affiorante rappresenti il puntatore del prossimo nodo da visitare: inizialmente si inserisce nella pila la radice; si entra poi in un ciclo in cui si estrae un elemento dalla pila, si visita il corrispondente nodo, e si inserisce nella pila prima il puntatore destro (se esiste) e poi il puntatore sinistro (se esiste). La corrispondente procedura compare in fig. 63.

Per ottenere la versione definitiva della procedura è necessario completare la dichiarazione di tipi, variabili, procedure e funzioni necessarie alla gestione della pila. Nel fare questo si dovrà anche scegliere la rappresentazione della pila.

Come tutte le rappresentazioni collegate, questo metodo consente agevoli operazioni di inserimento ed eliminazione di elementi (in questo caso nodi dell'albero, o, più in generale, sottoalberi) ed inoltre, quando è realizzata con record e puntatori, permette una occupazione di memoria proporzionale al numero di nodi, pur richiedendo spazio di memoria aggiuntivo per la memorizzazione dei puntatori.

L'operazione di accesso ai figli di un dato nodo può essere effettuata molto efficientemente, utilizzando in modo opportuno i riferimenti. Al contrario, la rappresentazione collegata che abbiamo descritto non consente di effettuare

```

procedure preordine(alb: punt_albero);
  { effettua la visita in predordine di un albero binario per stampare i valori
  dei nodi; alb è e il puntatore iniziale dell'albero }

type tipo_elementi_pila = punt_albero;
      tipo_pila =.....;

var pila: tipo_pila;

function test_pila_vuota(p: tipo_pila): boolean;
.......

procedure push (var p: tipo_pila; e: tipo_elementi_pila );
.......

procedure pop (var p: tipo_pila; var e: tipo_elementi_pila );

.......

procedure assegna_vuota (var p: tipo_pila);
.......

begin
  { si assegna a pila il valore pari a pila_vuota }
  assegna_vuota(pila);
  { se l'albero non è vuoto, si memorizza nella pila il puntatore alla radice,
  in modo che essa sia il primo nodo ad essere visitato }
  if alb <> nil
  then push(pila, alb);
  while not(test_pila_vuota(pila) )
  do begin
    { l'elemento affiorante della pila è il puntatore alla radice del
    prossimo sottoalbero da visitare, cioè è il puntatore al prossimo
    nodo dell'albero da visitare }
    pop(pila, alb);
    write(alb^.info);
    { si inseriscono nella pila i puntatori al figlio destro (prima) e al
    figlio sinistro (dopo), solo se sono diversi da nil; l'ordine è dovuto
    al fatto che il figlio sinistro è il prossimo da analizzare, e quindi
    deve essere l'ultimo ad essere inserito nella pila }
    if alb^.des <> nil
    then push(pila, alb^.des);
    if alb^.sin <> nil
    then push(pila, alb^.sin)
  end
end;

```

Fig. 63 - La procedura iterativa preordine

agevolmente l'operazione di accesso al padre di un nodo. Se questa operazione deve essere effettuata spesso, è consigliabile modificare la rappresentazione prevedendo, per ogni nodo, un ulteriore puntatore al padre. Su questo aspetto torneremo più avanti, quando parleremo di rappresentazioni tramate.

Illustriamo, infine, una procedura che consente di costruire in memoria la rappresentazione di un albero a partire da un insieme di dati di ingresso. Tenendo presente la corrispondenza che abbiamo stabilito tra alberi binari e liste, decidiamo che l'albero binario venga fornito in ingresso mediante la rappresentazione parentetica della corrispondente lista (tale rappresentazione viene semplicemente detta rappresentazione parentetica dell'albero). La procedura dovrà leggere ed interpretare correttamente i simboli in ingresso e costruire in memoria la rappresentazione dell'albero con record e puntatori.

Assumiamo che le informazioni associate ai nodi dell'albero siano di tipo carattere. Dopo aver letto da ingresso il primo carattere (la parentesi aperta), la procedura legge un ulteriore dato da ingresso. Se tale carattere è la parentesi chiusa, allora viene costruito un albero vuoto, semplicemente ponendo a *nil* il puntatore iniziale; altrimenti viene creato un nuovo nodo memorizzando l'informazione associata alla radice, e, ricorsivamente, viene costruito il sottoalbero sinistro ed il sottoalbero destro. La procedura è mostrata in fig. 64. Essa assume che la rappresentazione parentetica in ingresso sia corretta.

6.4. Gli alberi binari di ricerca

Come ultimo aspetto relativo agli alberi binari, parliamo in questo paragrafo degli alberi binari di ricerca, che sono usati quando si devono memorizzare grosse quantità di dati su cui viene spesso eseguita l'operazione di ricerca di un elemento.

Definizione. Un albero binario di ricerca è un albero binario in cui ogni nodo N ha la seguente proprietà (si assuma che sui valori associati ai nodi dell'albero sia definito un ordinamento): tutti i nodi del sottoalbero sinistro di N hanno un valore minore o uguale di quello di N e tutti i nodi del sottoalbero destro hanno un valore maggiore di quello del nodo N .

L'albero di fig. 65, ad esempio, è un albero binario di ricerca.

L'operazione di ricerca in tali alberi può essere realizzata mediante questo semplice algoritmo:

```

procedure costruisci(var alb: punt_albero);
  { costruisce la rappresentazione con record e puntatori di un albero
    binario di cui si legge da ingresso la rappresentazione parentetica; il
    puntatore iniziale dell'albero viene restituito nel parametro alb }
  var ch: char;
  begin
    { viene letta la parentesi aperta iniziale }
    read(ch);
    read(ch);
    if ch = '('
      then { l'albero da costruire è vuoto }
        alb := nil
      else begin
        { si crea il nodo corrispondente alla radice }
        new(alb);
        alb^.info := ch;
        { si costruisce il sottoalbero sinistro }
        costruisci(alb^.sin);
        { si costruisce il sottoalbero destro }
        costruisci(alb^.des);
        { viene letta la parentesi chiusa finale }
        read(ch)
      end
    end;

```

Fig. 64 - La procedura *costruisci*

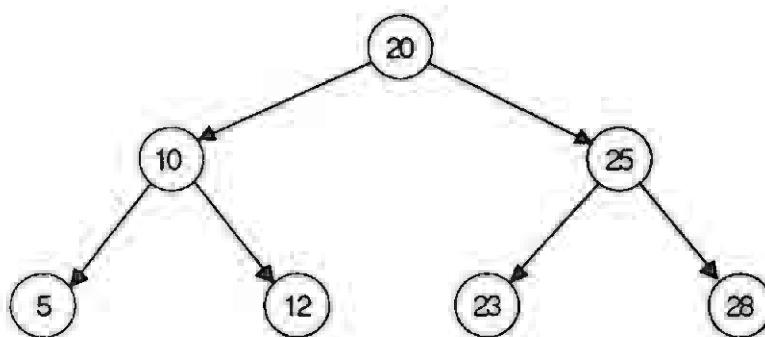


Fig. 65 - Un albero binario di ricerca

cerca il valore x nell'albero T:

```
se test_albero_vuoto(T)
  allora restituisci false
  altrimenti se x = radice(T)
    allora restituisci true
    altrimenti se x < radice(T)
      allora cerca il valore x
      nell'albero sinistro(T)
      altrimenti cerca il valore x
      nell'albero destro(T)
```

Se confrontiamo questo metodo di ricerca con una ricerca in un albero binario qualunque, ottenuta secondo un qualunque algoritmo di visita, si può apprezzare la maggiore efficienza consentita dall'albero binario di ricerca: infatti, il numero di confronti necessari per stabilire se l'elemento è o no presente è al massimo pari alla profondità dell'albero più uno. Se, quindi, si riesce a mantenere basso il valore della profondità, si può fare in modo che ad ogni passo della ricerca si eliminino la metà degli elementi, con evidente guadagno di efficienza. Riprenderemo più approfonditamente queste considerazioni nel cap. 6, quando parleremo della complessità degli algoritmi.

6.5. Gli alberi N-ari

Gli alberi che abbiamo trattato nelle precedenti sezioni sono soggetti al vincolo che ogni loro nodo ha al massimo due figli. Un albero non soggetto a tale vincolo viene detto albero N-ario, o, semplicemente, albero.

Definizione. Un albero è un grafo orientato che o è vuoto oppure ha le seguenti caratteristiche:

1. esiste un nodo R, detto radice, senza predecessori, con $N(N \geq 0)$ successori A₁, A₂, ..., A_N;
2. tutti gli altri nodi sono ripartiti in n sottoalberi mutuamente disgiunti T₁, T₂, ..., T_N, che hanno rispettivamente A₁, A₂, ..., A_N come radice.

Nella definizione abbiamo assunto che l'albero sia ordinato, cioè abbiamo assunto che sui figli di ogni nodo (e, quindi, anche sui sottoalberi) sia definito un ordinamento (A₁ viene prima di A₂, che viene prima di A₃, e così via).

Molte considerazioni sugli alberi sono dirette generalizzazioni di quelle su

alberi binari.

Riguardo ai metodi di visita, i più comuni per gli alberi N-ari sono due: la visita in preordine e la visita in postordine.

I corrispondenti algoritmi possono essere formulati nel modo seguente:

visita in preordine l'albero T

{siano T_1, T_2, \dots, T_n i sottoalberi di T ; si assuma T non vuoto}

analizza la radice di T ;

Per ogni sottoalbero T_i

visita in preordine l'albero T_i

visita in postordine l'albero T

{siano T_1, T_2, \dots, T_n i sottoalberi di T ; si assuma T non vuoto}

Per ogni sottoalbero T_i

visita in postordine l'albero T_i ;

analizza la radice di T

In fig. 66 compare un esempio di albero, in cui le informazioni associate ai nodi sono di tipo intero. Per questo albero, la visita in preordine ha l'effetto di visitare i nodi secondo questa sequenza:

5 4 8 9 12 11 3 15 6 21 20

Mentre la visita in postordine ha l'effetto di visitare i nodi secondo questa sequenza:

8 12 11 3 9 4 15 21 20 6 5

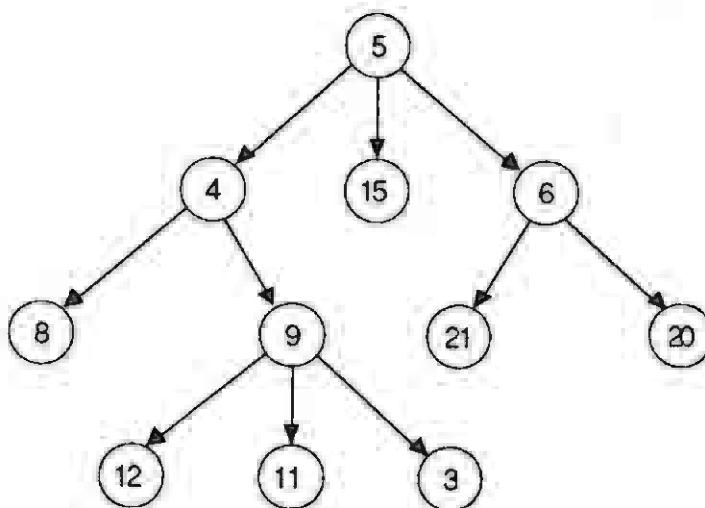


Fig. 66 - Un albero N-ario

Analizziamo ora i metodi per la rappresentazione degli alberi.

6.5.1. La rappresentazione mediante lista

Un albero N-ario può essere rappresentato mediante una lista secondo queste regole:

- se l'albero è vuoto, la lista che lo rappresenta è vuota;
- altrimenti, l'albero è composto da una radice e da k sottoalberi T₁, T₂, ..., T_k, e la lista è formata da k+1 elementi: il primo rappresenta la radice, mentre gli altri sono liste che rappresentano rispettivamente gli alberi T₁, T₂, ..., T_k ($k \geq 0$).

La lista che rappresenta un albero è, in generale, rappresentata mediante una struttura collegata. Diversamente dal caso degli alberi binari, non è possibile associare ad ogni elemento un numero di puntatori pari al massimo dei figli, poiché tale numero non è in generale determinabile a priori. La rappresentazione, allora, è leggermente più complessa: la radice dell'albero viene memorizzata nel primo elemento della lista, che contiene il riferimento ad una lista di elementi, uno per ogni sottoalbero. Ciascuno di questi elementi contiene, a sua volta, il riferimento iniziale alla lista che rappresenta il corrispondente sottoalbero. Mostriamo, in fig. 67, la rappresentazione diagrammatica della lista corrispondente all'albero di fig. 66.

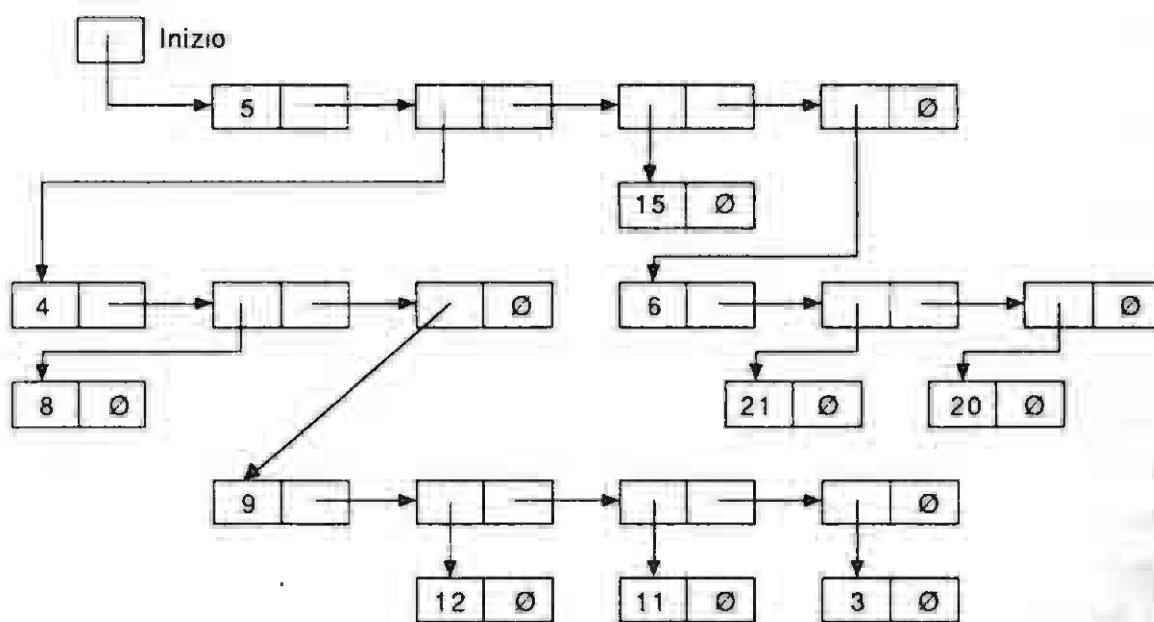


Fig. 67 - La lista corrispondente all'albero di fig. 66

Quando la lista viene realizzata con record e puntatori, ogni elemento della lista è un record ed ogni riferimento è un puntatore. In questo caso, o si definisce un tipo **record** con parte variante, oppure si definisce un tipo **record** con tre campi, uno per la parte informazione (cioè il dato associato al nodo dell'albero), e due per i puntatori. In quest'ultimo caso, per ogni record sarà sempre significativo uno dei campi puntatori, mentre o il campo informazione o l'altro campo puntatore non verrà utilizzato. Se è necessario distinguere i due casi suddetti, si può aggiungere un opportuno campo ad ogni record che indichi, con il suo valore, quale dei campi è significativo. In Pascal, una possibile dichiarazione per realizzare questa rappresentazione è:

```
type punt = ^nodo_alb;
nodo_alb = record
    info: tipo_elemento;
    figlio, next: punt
end;
```

tipo_elemento è il tipo dei valori associati ai nodi dell'albero; *figlio* è il campo riservato al puntatore al sottoalbero, mentre *next* è il campo per il puntatore al successivo elemento della lista (cioè al record che contiene il puntatore al successivo sottoalbero).

Come esempio di realizzazione di una operazione, presentiamo in fig. 68 una procedura ricorsiva per la visita in postordine (eseguita per stampare i valori associati ai nodi) di un albero, assumendo note le dichiarazioni di tipo scritte sopra.

La rappresentazione descritta presenta i vantaggi delle rappresentazioni collegate: ad esempio, eventuali modifiche sull'albero si effettuano agevolmente, modificando opportunamente i riferimenti. Quando poi si usano record e puntatori, lo spazio di memoria utilizzato è proporzionale alla dimensione dell'albero.

Riguardo alla efficienza delle operazioni, valgono considerazioni analoghe a quelle fatte per la rappresentazione collegata di alberi binari: l'operazione di accesso ai figli di un nodo è eseguibile in modo efficiente, mentre l'accesso al padre di un nodo presenta notevole difficoltà.

Anche per gli alberi n-ari sorge l'esigenza di utilizzare la rappresentazione parentetica, che è esattamente la rappresentazione parentetica della lista che rappresenta l'albero. Essa può essere descritta mediante le seguenti regole sintattiche:

```

procedure postordine (p: punt);
  { visita in postordine un albero rappresentato mediante lista; p è il
    puntatore alla radice dell'albero }

var q: punt;
begin
  { si lascia che p punti alla radice }
  q := p;
  { visita ricorsivamente tutti i sottalberi }
  while q.next <> nil
  do begin
    { ci sono ancora sottalberi da visitare }
    q := q.next;
    postordine(q.figlio)
  end;
  { visita la radice }
  write(p.info)
end;

```

Fig. 68 - La procedura ricorsiva *postordine*

$\langle \text{albero} \rangle ::= () / \langle \text{albero_non_vuoto} \rangle$
 $\langle \text{albero_non_vuoto} \rangle ::= ("(" \langle \text{nodo} \rangle \{ \langle \text{sottoalbero} \rangle \} ")")$
 $\langle \text{sottoalbero} \rangle ::= \langle \text{albero_non_vuoto} \rangle$

Il significato di tale rappresentazione è il seguente:

1. la stringa () denota l'albero vuoto;
2. se N è un simbolo che denota un nodo, e S1, S2, ..., Sn denotano, secondo la rappresentazione parentetica, gli alberi non vuoti A1, A2, ..., An, allora la stringa:

$$(N S_1 S_2 \dots S_n)$$

denota l'albero che ha N come radice e A1, A2, ..., An (in questo ordine) come sottalberi.

Esempio. La rappresentazione parentetica dell'albero di fig. 66 è:

$$(5 (4 (8) (9 (12) (11) (3)))) (15) (6 (21) (20)))$$

6.5.2. Altre rappresentazioni

Un albero N-ario può anche essere rappresentato mediante un albero binario, secondo questa regola:

- per ogni nodo N_n dell'albero N-ario ne esiste uno corrispondente N_b dell'albero binario;
- al primo figlio (quello più a sinistra) di N_n corrisponde il figlio sinistro di N_b , mentre al fratello immediatamente a destra di N_n corrisponde il figlio destro di N_b .

In fig. 69 compare la rappresentazione mediante albero binario dell'albero di fig. 66.

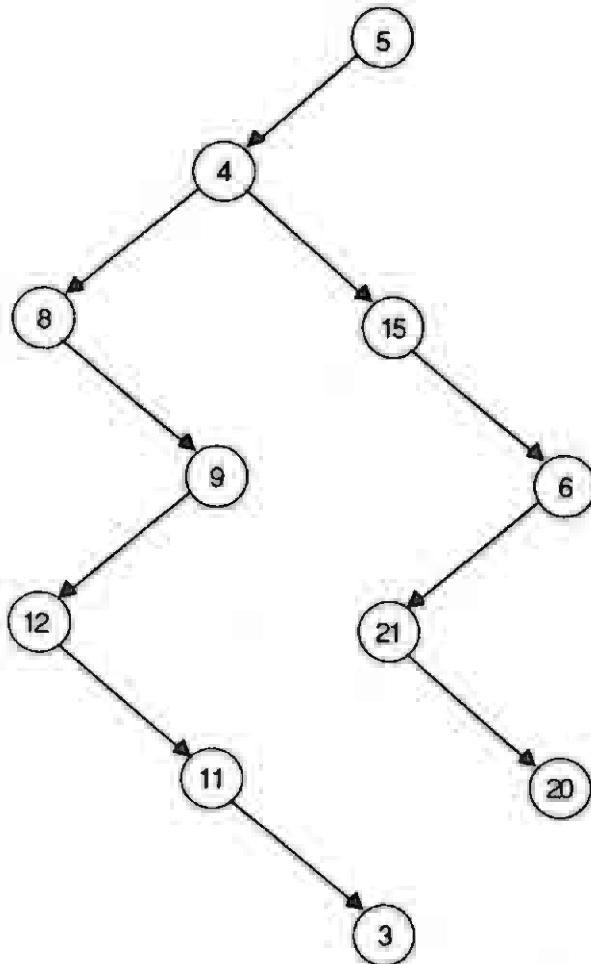


Fig. 69 - Rappresentazione di albero mediante albero binario

Si noti che la sequenza dei nodi visitati in preordine dell'albero binario corrisponde alla sequenza in preordine dell'albero N-ario, mentre le due sequenze in postordine non coincidono.

La rappresentazione mediante albero binario ha il pregio di richiedere un numero di record inferiori rispetto alla rappresentazione mediante lista: infatti, il numero di record è esattamente pari al numero di nodi dell'albero N-ario.

Come esempio di operazione eseguita su questa rappresentazione, consideriamo la visita in postordine. Prima di presentare la procedura ricorsiva che la realizza, definiamo il tipo puntatore ed il tipo record per i nodi dell'albero. Il record ha tre campi: il campo informazione, e due puntatori, quello al primo figlio del nodo e quello al fratello.

```

type punt = ^record_nodo;
record_nodo = record
    info: integer;
    primo_figlio, fratello: punt
end;

procedure postordine(p: punt);
    { esegue la visita in postordine di un albero N-ario rappresentato
      mediante albero binario; l'albero binario è realizzato con record e
      puntatori; p è il puntatore alla radice }
var q: punt;
begin
    q := p^.primo_figlio;
    while q <> nil
        do begin
            { visita il prossimo sottoalbero di p }
            postordine(q);
            q := q^.fratello
        end;
        { analizza la radice }
        write(p^.info)
    end;

```

Ulteriori rappresentazioni per gli alberi sono le cosiddette *rappresentazioni tramate*. Esse sono rappresentazioni collegate in cui, allo scopo di agevolare alcune operazioni, si fa uso di puntatori ausiliari, detti di trama. La più semplice rappresentazione tramata è quella che associa ad ogni nodo il puntatore al padre. Abbiamo più volte sottolineato che nella rappresentazione di alberi mediante liste, l'operazione di accesso al padre di un nodo è complessa: essa richiede, nel caso peggiore, l'analisi di tutti i nodi dell'albero. Al contrario, associando ad ogni nodo il puntatore al padre, si può effettuare questa operazione in modo semplice ed efficiente. In fig. 70 compare questa rappresentazione per l'albero di fig. 66; i puntatori di trama sono quelli tratteggiati.

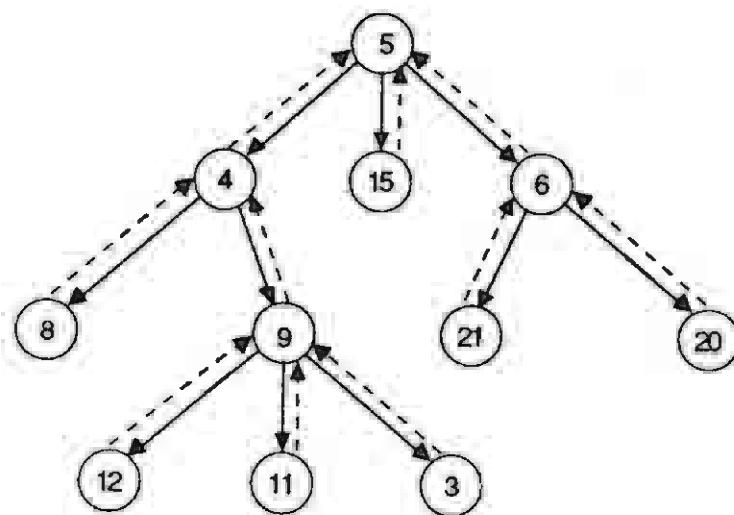


Fig. 70 - Rappresentazione tramata di un albero

Un'altra rappresentazione tramata è quella in cui i puntatori di trama servono a facilitare la visita dell'albero. Ad esempio, per la visita in preordine, si può fare in modo che il puntatore di trama associato al nodo N punti al nodo che segue N nella visita in preordine. In fig. 71 compare la rappresentazione tramata per la visita in preordine dell'albero di fig. 66.

Tutte le rappresentazioni tramate si rivelano particolarmente utili per alcune operazioni, ma quando l'albero viene modificato, esse richiedono operazioni di aggiornamento dei puntatori di trama.

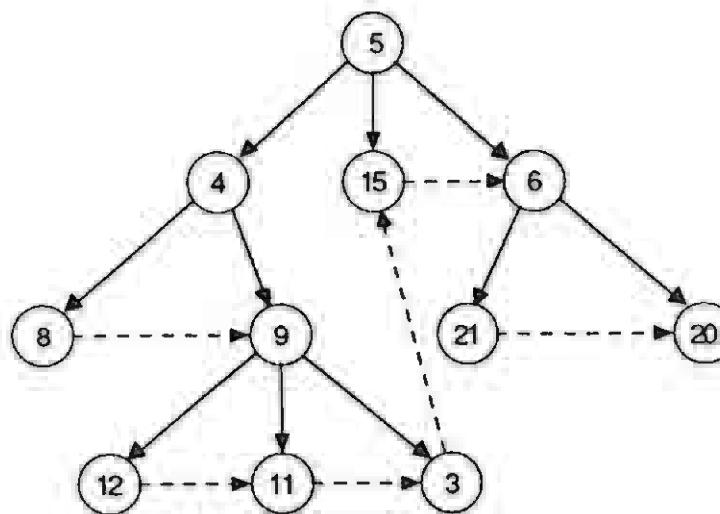


Fig. 71 - Rappresentazione tramata per la visita in postordine

7. I GRAFI

I grafi sono strutture di dati che rappresentano relazioni binarie su un insieme di elementi. Abbiamo già dato la definizione di grafo nel par. 3.1 dell'appendice.

dice, illustrando anche diversi concetti relativi ai grafi.

In questo paragrafo analizzeremo i metodi per la rappresentazione di grafi e presenteremo alcuni algoritmi fondamentali su queste strutture di dati.

Come per gli alberi, anche per i grafi sorge l'esigenza di definire algoritmi che consentono l'analisi di tutti i nodi, secondo una prefissata sequenza. Due sono i più comuni metodi per visitare un grafo, la visita in profondità e la visita in ampiezza.

La visita in profondità (in inglese *depth first search*) è l'analogia della visita in preordine di un albero: si analizza il nodo di partenza I , e per ogni successore S di I , si visita, con lo stesso metodo, il grafo partendo da S . Al contrario degli alberi, in un grafo è possibile tornare, seguendo gli archi, ad un nodo già analizzato, in particolare se esistono cicli nel grafo. Per evitare che l'algoritmo di visita entri in un ciclo infinito, è necessario controllare che il nodo di volta in volta considerato non sia già stato analizzato. L'algoritmo di visita in profondità può essere così formulato:

visita in profondità il grafo G partendo dal nodo I :

analizza I e marcalo "visitato";

finché ci sono successori di I non ancora visitati (sia S il prossimo)

visita in profondità il grafo G partendo dal nodo S

Esempio. In fig. 72 compare un grafo i cui nodi sono etichettati con valori interi. Eseguendo la visita in profondità partendo dal nodo 1, un possibile ordine di visita dei nodi è il seguente:

1 3 4 6 5 7 2

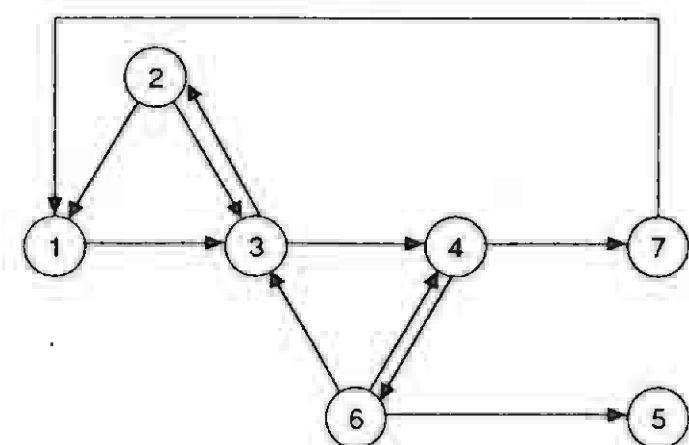


Fig. 72 - Un grafo

La visita in ampiezza (in inglese *breadth first search*) prevede, invece, che,

partendo da un nodo I , si visitino tutti i suoi successori, poi tutti i successori dei successori e così via. L'algoritmo corrispondente fa uso di una coda, e può essere formulato nel modo seguente:

visita il grafo G partendo dal nodo I :
inserisci I nella coda;
finché la coda non è vuota
esegui
 estrai il primo elemento J dalla coda;
 analizza J e marcalo "visitato";
 metti nella coda tutti i successori di J che non sono
 marcati
fine

Esempio. Eseguendo la visita in ampiezza del grafo di fig. 72 partendo dal nodo 1, un possibile ordine di visita dei nodi è il seguente:

1 3 4 2 6 7 5

7.1. Rappresentazioni di grafi

Il primo tipo di rappresentazione che analizziamo fa uso di una matrice, detta matrice delle adiacenze. La matrice delle adiacenze corrispondente al generico grafo G ha tante righe e colonne quanti sono i nodi di G . I nodi del grafo sono in corrispondenza biunivoca con i valori degli indici di riga e di colonna; quando parliamo del "nodo i " ci riferiamo al nodo che corrisponde al valore i di riga e colonna della matrice. Gli elementi della matrice sono di tipo booleano ed il loro valore è stabilito da questa regola: l'elemento nella riga i e nella colonna j della matrice (elemento $\langle i,j \rangle$) è pari a *true* se nel grafo G c'è un arco dal nodo i al nodo j , *false* altrimenti.

Esempio. La matrice delle adiacenze che rappresenta il grafo di fig. 72 è mostrata in fig. 73 dove 0 rappresenta *false* e 1 *true*.

Se non è possibile porre in corrispondenza diretta gli indici della matrice con i valori associati ai nodi del grafo, si fa uso di un vettore in cui si memorizza l'etichetta di ogni nodo ed il corrispondente valore dell'indice della matrice delle adiacenze. Chiameremo tale vettore, il *vettore dei nodi*.

	1	2	3	4	5	6	7
1	0	0	1	0	0	0	0
2	1	0	1	0	0	0	0
3	0	1	0	1	0	0	0
4	0	0	0	0	0	1	1
5	0	0	0	0	0	0	0
6	0	0	1	1	1	0	0
7	1	0	0	0	0	0	0

Fig. 73 - Matrice delle adiacenze

Esempio. In fig. 74 compare un grafo etichettato nei nodi con valori di tipo stringa. Nella stessa figura è mostrato un possibile vettore dei nodi per tale grafo.

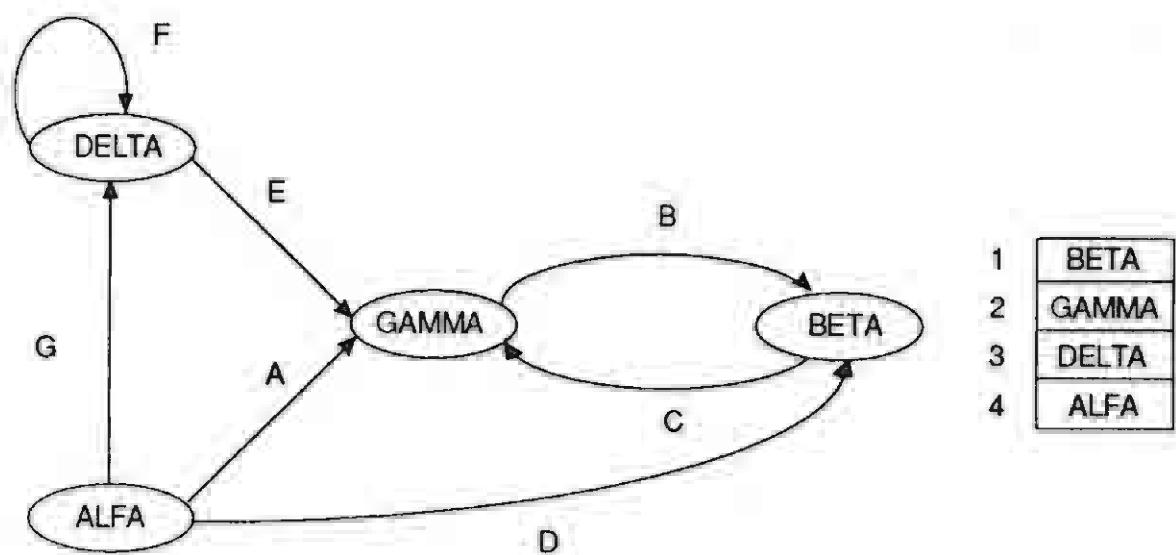


Fig. 74 - Un grafo etichettato ed il corrispondente vettore dei nodi

Se il grafo è etichettato negli archi, i valori degli elementi della matrice delle adiacenze non sono booleani, ma del tipo delle etichette degli archi. In questi casi si utilizza un valore speciale in posizione $\langle i,j \rangle$ per indicare che nel grafo non c'è l'arco dal nodo i al nodo j.

Esempio. Il grafo di fig. 74 è etichettato negli archi con valori di tipo carattere. In fig. 75 compare la matrice delle adiacenze corrispondente a tale grafo. Il valore 'Z' nella matrice indica che nel grafo non c'è l'arco corrispondente.

La rappresentazione mediante matrice delle adiacenze ha il pregio di essere molto semplice e di consentire un accesso diretto all'informazione relativa

		1	2	3	4
1	BETA	Z	C	Z	Z
2	GAMMA	B	Z	Z	Z
3	DELTA	Z	E	F	Z
4	ALFA	D	A	G	Z

Fig. 75 - Matrice delle adiacenze per un grafo etichettato

all'arco che collega due nodi. In particolare, per la verifica se un nodo è successore o predecessore di un altro nodo, si utilizza il meccanismo di accesso diretto della matrice.

Due sono i difetti principali di tale rappresentazione: da una parte essa impone un limite massimo per i nodi del grafo (determinato dal numero di righe e colonne della matrice), e dall'altra essa richiede un'occupazione di memoria proporzionale al numero massimo di archi del grafo, pari a $N \times N$, dove N è il numero di nodi.

Si noti che l'operazione di accesso ai successori di un nodo (ad esempio il nodo i) richiede l'analisi della i -esima riga, e quindi richiede l'accesso ad N elementi della matrice. Analogamente, l'operazione di accesso ai predecessori di un nodo (ad esempio il nodo i) richiede l'analisi della i -esima colonna. Come esempio di realizzazione di una operazione su un grafo rappresentato mediante matrice delle adiacenze, presentiamo una funzione che verifica se tra due nodi di un grafo esiste un semicammino (per la definizione di semicammino, si veda l'appendice). La verifica dell'esistenza di un semicammino viene realizzata mediante una variante della visita in profondità dal nodo h : per ogni nodo N si considerano, oltre ai successori di N , anche i suoi predecessori.

La matrice delle adiacenze consente di eseguire la verifica in modo particolarmente efficiente, perchè permette un facile accesso ai predecessori di un nodo. Come in tutti gli algoritmi di visita di un grafo, faremo uso di un vettore di marche associate ai nodi (se si utilizza il vettore di nodi le marche si possono inserire direttamente in questo vettore), che servono per verificare se un nodo è già stato analizzato. La definizione di tipi per la rappresentazione del grafo è la seguente:

```

type tipo_grafo = record
    { N è una costante che rappresenta il numero massimo
      di nodi del grafo }
    marche: array [1..N] of boolean;
    mat_adiacenze: array [1..N,1..N] of boolean
end;

```

La funzione è mostrata in fig. 76.

Abbiamo assunto che, inizialmente, il valore di tutte le componenti del vettore *marche* sia *false*. All'interno della funzione *sempcammino* è stata definita la funzione *esiste*, che effettua ricorsivamente la visita del grafo.

Una variante della rappresentazione di un grafo mediante matrice delle adiacenze consiste nel rappresentare tale matrice in modo compatto, diminuendo, così, lo spazio di memoria richiesto per il grafo. Si può ad esempio utilizzare una rappresentazione compatta realizzata con un vettore di record, i cui elementi sono in corrispondenza biunivoca con gli archi del grafo. Il generico record corrispondente all'arco $\langle i,j \rangle$ contiene due campi, uno per il nodo i di partenza e l'altro per il nodo j di arrivo. Si veda in fig. 77 la rappresentazione compatta della matrice delle adiacenze di fig. 73.

Si noti che in questa rappresentazione l'operazione di accesso ai successori e ai predecessori di un nodo non è agevole come nel caso della rappresentazione non compatta della matrice delle adiacenze.

Un secondo metodo di rappresentazione consiste nell'associare, ad ogni nodo i del grafo una lista semplice, realizzata mediante rappresentazione collegata, in ogni elemento della quale si memorizza il riferimento ad uno dei successori di i . Tale rappresentazione è nota sotto il nome di *liste di successori* e comprende:

1. il vettore dei nodi, in cui, oltre a memorizzare le eventuali etichette dei nodi, si memorizza il riferimento iniziale di una lista associata ad ogni nodo;
2. una lista semplice per ogni nodo del grafo, detta lista dei successori; la lista associata al generico nodo I contiene tanti elementi quanti sono i successori di I ; ciascun elemento è il riferimento ad uno dei successori (cioè l'indice dell'elemento del vettore dei nodi corrispondente al successore).

Esempio. La rappresentazione del grafo di fig. 72 in termini di liste di successori è mostrata in fig. 78.

```

function semicammino( g: tipo_grafo; num_nodi: integer;
h,k: integer): boolean;
{ verifica se nel grafo g esiste un semicammino tra h e k; num_nodi è il
 numero di nodi del grafo g; si assume che il valore di tutte le componenti
 del vettore marche sia false }

function esiste(i: integer): boolean;
{ funzione ricorsiva che verifica se esiste un semicammino tra i e k
 nel grafo g }
var temp: boolean;
begin
  if i=k
    then esiste := true
  else if g.marche[i]
    then esiste := false
  else begin
    g.marche[i] := true;
    temp := false;
    { si continua la visita a partire dai successori }
    j := 1;
    while (j < num_nodi) and (not temp)
      do begin
        if g.mat_adiacenze[i,j]
          then temp := esiste(j);
          j := j+1
        end;
        { si continua la visita a partire dai predecessori }
        j := 1;
        while (j < num_nodi) and (not temp)
          do begin
            if g.mat_adiacenze[j,i]
              then temp := esiste(j);
              j := j+1
            end;
          esiste := temp
        end
      end;
    { inizio istruzioni di semicammino }
    begin
      semicammino := esiste(h)
    end;

```

Fig. 76 · La funzione *semicammino*

nodo di partenza *nodo di arrivo*

	1	3
1	1	
2	2	1
3	2	3
4	3	2
5	3	4
6	4	6
7	4	7
8	6	3
9	6	4
10	6	5
11	7	1

Fig. 77 - Rappresentazione compatta della matrice di fig. 73

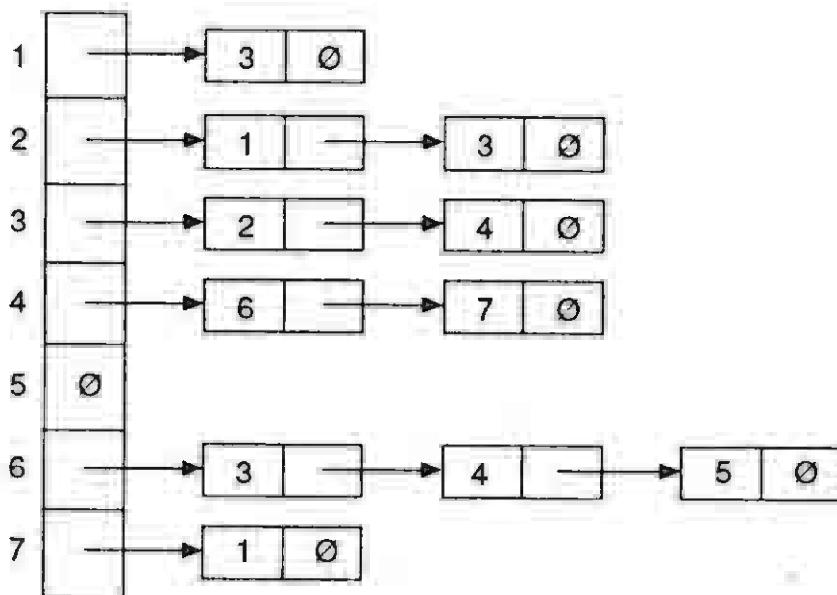


Fig. 78 - Liste di successori

Si noti che gli elementi della lista dei successori del nodo 1 sono in corrispondenza con gli archi uscenti dal nodo. Quindi, eventuali etichette degli archi si possono memorizzare negli elementi della liste dei successori.

Esempio. In fig. 79 compare la rappresentazione mediante liste di successori del grafo di fig. 74.

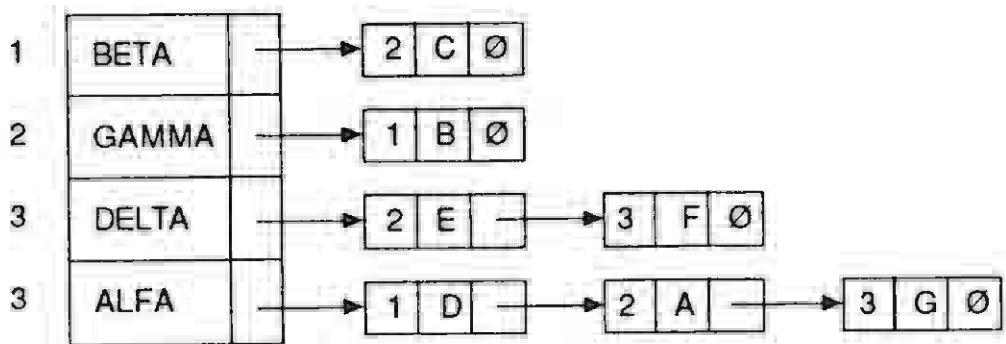


Fig. 79 - Liste di successori di un grafo etichettato negli archi

Realizziamo ora la visita in ampiezza di un grafo rappresentato mediante liste di successori (si veda la fig. 80).

Assumiamo nota la seguente definizione di tipi:

```

type punt = ^record_archi;
record_archi = record
    succ: integer;
    next: punt
end;
tipo_grafo = array [1..N] of record
    marca: boolean;
    successori: punt
end;

```

Rispetto all'occupazione di memoria, la rappresentazione con liste di successori è migliore di quella con matrice delle adiacenze: se N è il numero dei nodi del grafo e M è il numero degli archi, lo spazio di memoria per le liste di successori è proporzionale a N + M.

Nella rappresentazione mediante liste di successori, la verifica dell'esistenza di un arco dal nodo i al nodo j richiede la scansione della lista dei successori di i, ed è quindi meno efficiente che nel caso della matrice delle adiacenze. Lo stesso vale per l'operazione di accesso ai predecessori di un nodo: per trovare tutti i predecessori di un nodo i è necessario scandire, per ogni nodo j del grafo, la relativa lista dei successori e verificare se tra essi compaia l'arco entrante nel nodo i.

Al contrario, l'operazione di accesso a tutti i successori di un nodo può essere effettuata più efficientemente, poiché richiede un numero di confronti pari al numero di archi uscenti dal nodo.

In tutte le rappresentazioni che abbiamo sinora descritto, le informazioni sui

```

procedure ampiezza(g: tipo_grafo; i: integer);
  { visita in ampiezza il grafo g, rappresentato mediante liste di successori,
    a partire dal nodo i }
var j: integer;

function test_coda_vuota ...

function primo ...

procedure assegna_vuota ...
  { assegna il valore coda_vuota }

procedure in_coda...
procedure out_coda ...

begin
  assegna_vuota(q);
  in_coda(q, i);
  while not(test_coda_vuota(q))
  do begin
    { si estraе un elemento dalla coda }
    j := primo(q);
    out_coda(q);
    g[j].marca := true;
    write(j);
    p := g[j].successori;
    { si considerano tutti i suoi successori }
    while p <> nil
    do begin
      if g/p.succ].marca=false
      then in_coda(q, g/p.succ]);
      p := p.next
    end
  end
end;

```

Fig. 80 - La procedura *ampiezza*

nodi del grafo vengono memorizzate in un vettore. Da ciò deriva che aggiunte ed eliminazioni di nodi non vengono effettuate agevolmente, ed il numero dei nodi del grafo non può crescere oltre le dimensioni del vettore.

Per risolvere questi problemi, si può definire una variante della rappresenta-

zione mediante liste di successori, in cui si utilizza una lista per memorizzare le informazioni associate ai nodi, ottenendo quella che viene chiamata la rappresentazione mediante lista doppia. Si veda, in fig. 81, la rappresentazione mediante lista doppia di un grafo.

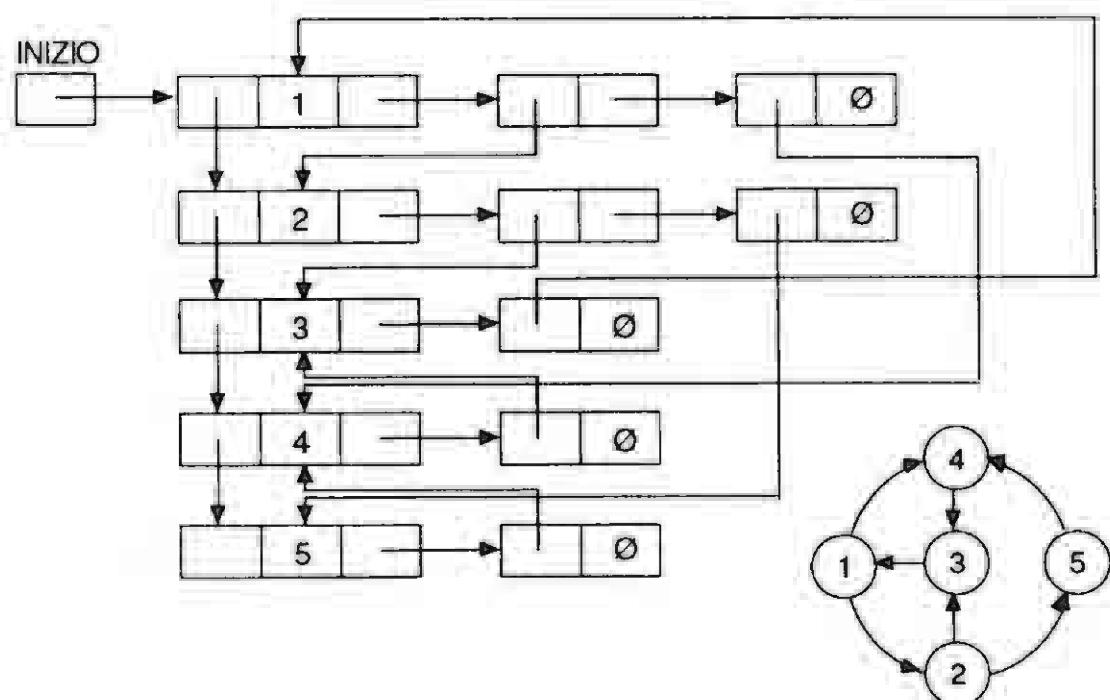


Fig. 81 - Lista doppia

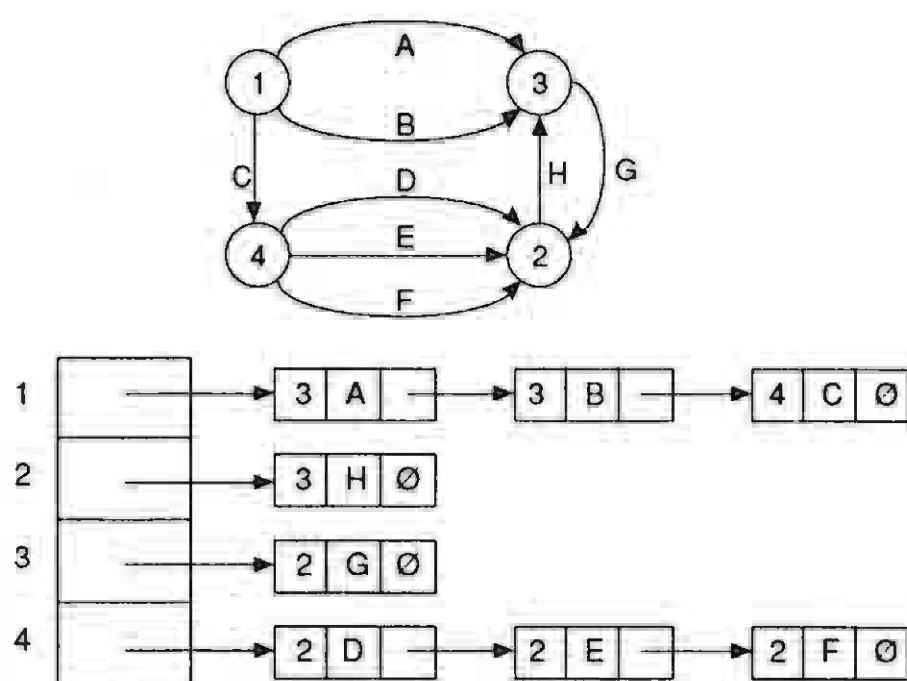


Fig. 82 - Un multigrafo e la sua rappresentazione

La lista doppia consente di effettuare agevolmente le operazioni di aggiunta ed eliminazione dei nodi del grafo. Tuttavia, l'accesso ad un nodo può risultare più complesso: mentre nel caso della rappresentazione mediante liste di successori (e anche nel caso della matrice delle adiacenze) si poteva sfruttare l'accesso diretto agli elementi del vettore dei nodi, ora l'accesso richiede la scansione della lista dei nodi.

Concludiamo questa sezione con alcune considerazioni sulla rappresentazione dei multigrafi. Ricordiamo che un multigrafo è un grafo in cui possono esistere più archi che collegano due nodi. Per i multigrafi si possono utilizzare le rappresentazioni che abbiamo descritto per i grafi, opportunamente adattate. Ad esempio, nel caso di liste di successori, ogni lista rappresenterà un multiinsieme di archi, invece che un insieme. Come per i grafi, eventuali etichette degli archi vengono memorizzate negli elementi delle liste dei successori.

Esempio. In fig. 82 compare un multigrafo e la corrispondente rappresentazione mediante liste di successori.

7.2. Il problema del percorso più breve in un grafo

Supponiamo di avere un grafo etichettato negli archi con valori interi positivi. I nodi possono, ad esempio, rappresentare le città di una nazione, e gli archi strade che le collegano. Delle strade sono note le lunghezze, rappresentate dalle etichette degli archi. Il problema che affrontiamo in questa sezione è quello di trovare la lunghezza del percorso più breve tra due città A e B . Un percorso tra due città corrisponde nel grafo ad un cammino tra due nodi, e la lunghezza del percorso corrisponde alla somma delle etichette degli archi che formano tale cammino (peso del cammino). Il problema si risolve allora calcolando la lunghezza del cammino con peso minimo (detto anche cammino minimo) tra tutti quelli da A a B .

L'algoritmo che utilizziamo è stato proposto da Dijkstra e si basa sull'idea di calcolare, in ordine crescente, la lunghezza dei cammini minimi da A a tutti i nodi del grafo. Indichiamo con S l'insieme dei nodi di cui, ad un certo punto dell'algoritmo, si è già calcolato la lunghezza del cammino minimo da A . Utilizziamo un vettore $dist$ con tante componenti quanti sono i nodi del grafo, in modo che $dist[j]$ rappresenti la lunghezza del cammino minimo tra quelli che vanno da A a j passando solo (a parte j) per nodi contenuti in S . Osserviamo che se il prossimo cammino minimo C da generare è quello da A al nodo W , allora tutti i nodi di tale cammino (tranne W) sono contenuti in S . Infatti, se un nodo

```

procedure cammini_minimi (g: tipo_grafo; A: integer;
                         var dist: array_distanze;)
    { dato un grafo g ed un nodo A, calcola la lunghezza del cammino minimo
      da A ad ogni nodo i di g, memorizzandolo in dist[i] }

var S: tipo_insieme;
     num, i: integer; W: integer;
begin
    { assegna a S il valore insieme_vuoto }
    assegna_vuoto(S);
    for i := 1 to N
        do { assegna a dist[i] l'etichetta dell'arco  $\langle A, i \rangle$  }
            dist[i] := etichetta(A,i);
        num := 0; { num è il numero di nodi di cui si è già calcolato il cammino
                      minimo }
    while num < N
        do begin
            { calcola il nodo W non in S tale che dist[W] è minore di tutti i
              valori dist[K] per K non in S }
            W := calcola_minimo;
            { inserisci W in S }
            inserisci(S,W);
            num := num + 1;
            { aggiorna dist, assegnando a dist[Z], per ogni Z non in S, il
              minimo tra dist[Z] e dist[W]+t, dove t è l'etichetta dell'arco
               $\langle W, Z \rangle$  }
            aggiorna_distanze(W)
        end
    end;

```

Fig. 83 - La procedura *cammini_minimi*

K di *C* non appartenesse a *S*, vi sarebbe un cammino da *A* ad un nodo non contenuto in *S* (*K*, appunto) di lunghezza minore di quella di *C*, contraddicendo l'ipotesi che il prossimo cammino minimo da generare sia proprio *C*. Si noti che la lunghezza di *C* ed il nodo *W* sono facilmente individuabili: è sufficiente, infatti, calcolare il valore minimo di *dist[j]*, per *j* non contenuto in *S*.

Una volta individuato *W*, lo si può senz'altro inserire in *S*, e si può procedere all'aggiornamento di *dist*, per i nodi che non appartengono a *S*: il valore di *dist*

per questi nodi può infatti cambiare dopo che W è stato inserito in S . In particolare, se per un certo nodo Z connesso a W da un arco $\langle W, Z \rangle$ con etichetta E , la somma $dist[W] + E$ è minore di $dist[Z]$, allora a $dist[Z]$ deve essere assegnato il nuovo valore $dist[W] + E$. Infatti, in questo caso (e solo in questo) il cammino minimo tra quelli che vanno da A a Z passando solo per nodi in S , è cambiato, essendo ora composto dal cammino minimo da A a W e dall'arco $\langle W, Z \rangle$.

Possiamo sintetizzare quanto detto nell'algoritmo di fig. 83, in cui si assume che i nodi del grafo siano etichettati con valori da 1 a N . Si assume anche che per ogni nodo i , ci sia l'arco $\langle i, i \rangle$ etichettato con 0, e che tra due nodi non connessi vi sia un arco fittizio etichettato con un valore maggiore di tutte le altre etichette (nell'algoritmo, useremo $maxint$).

L'algoritmo calcola in $dist[i]$ la lunghezza del cammino minimo da A ad ogni nodo i del grafo: se siamo interessati al cammino minimo da A a B , il valore cercato è quindi memorizzato in $dist[B]$. Si noti che l'algoritmo non consente di ricostruire la sequenza di nodi che forma il cammino minimo; per ottenerla, si dovrebbe utilizzare un'apposita struttura di dati da aggiornare di volta in volta insieme al vettore $dist$.

Per completare l'algoritmo, occorre adesso scegliere il metodo di rappresentazione sia per il grafo g che per l'insieme S , e scrivere le procedure e le funzioni attivate dalla procedura *cammini_minimi*.

Per il grafo decidiamo la rappresentazione mediante liste di successori, mentre per l'insieme scegliamo la rappresentazione mediante vettore caratteristico. Le corrispondenti definizioni di tipi sono:

```

type tipo_insieme = array[1..N] of boolean;
  punt = ^record_archi;
  record_archi = record
    succ: integer;
    etic: integer;
    next: punt
  end;
  tipo_grafo= array [1..N] of punt;

```

Le procedure e le funzioni attivate dalla procedura *cammini_minimi* sono mostrate nelle figg. 84, 85, 86, 87 e 88.

```
procedure assegna_vuoto (var S: tipo_insieme);  
{ assegna all'insieme S, rappresentato mediante vettore caratteristico,  
il valore insieme_vuoto }  
var i:integer;  
begin  
    for i := 1 to N  
    do S[i] := false  
end;
```

Fig. 84 - La procedura *assegna_vuoto*

```
procedure inserisci (var S: tipo_insieme; e: integer);  
{ inserisce nell'insieme S, rappresentato mediante vettore caratteristico,  
il valore e }  
begin  
    S[e] := true  
end;
```

Fig. 85 - La procedura *inserisci*

```
function etichetta(E,F: integer): integer;  
{ calcola l'etichetta dell'arco da E a F nel grafo g }  
var p: punt;  
    trovato: boolean;  
begin  
    p := g[E];  
    trovato := false;  
    while p <> nil and trovato  
    do if p^.succ = F  
        then trovato := true  
        else p := p^.next;  
    if trovato  
    then etichetta := p^.etic  
    else etichetta := maxint  
end;
```

Fig. 86 - La funzione *etichetta*

```

function calcola_minimo: integer;
  { calcola il nodo W tale che dist[W] è minore di tutti i valori dist [K] tale
    che K non è contenuto nell'insieme S }
var minimo, i: integer;
begin
  i := 1;
  while S[i]
    do i := i + 1;
    minimo := i;
    for i := i+1 to N
      do if (dist[i] < dist[minimo]) and (not S[i])
        then minimo := i;
    calcola_minimo := minimo
end;

```

Fig. 87 - La funzione *calcola_minimo*

```

procedure aggiorna_distanze(W: integer);
  { aggiorna dist, assegnando a dist[Z], per ogni Z non in S, il minimo tra
    dist[Z] e dist[W]+t, dove t è l'etichetta dell'arco <W,Z> }
var i, t: integer;
begin
  for i := 1 to N
    do begin
      t := etichetta(W,i);
      if not(S[i])
        then if dist[W] + t < dist[i]
          then dist[i] := dist[W] + t
    end
  end;

```

Fig. 88 - La procedura *aggiorna_distanze*

8. LE TAVOLE

La tavola è un tipo astratto di dati che serve a rappresentare insiemi di coppie <chiave, attributi>. Ogni coppia rappresenta un insieme di dati che si riferiscono

no ad un'unica entità logica (una persona, un articolo merceologico, ecc.). La chiave è un dato che identifica l'entità logica, mentre gli attributi rappresentano le sue proprietà. Ad esempio, se si vogliono rappresentare le informazioni relative ai corsi che si tengono in una università, si potrà usare una tavola, in cui ogni elemento sarà formato dal codice del corso, la facoltà in cui il corso è insegnato ed il docente titolare del corso stesso. Poiché corsi diversi hanno codici diversi, il codice sarà la chiave della tavola, mentre gli altri dati saranno gli attributi.

Le operazioni tipiche sulle tavole sono:

inserisci: *tavola* \times *chiave* \times *attributi* \rightarrow *tavola*

cancella: *tavola* \times *chiave* \rightarrow *tavola*

esiste: *tavola* \times *chiave* \rightarrow boolean

ricerca: *tavola* \times *chiave* \rightarrow *attributi*

inserisci è l'operazione che serve ad inserire un nuovo elemento nella tavola: *inserisci* (T, C, A) fornisce come risultato la tavola che si ottiene aggiungendo a T l'elemento di chiave C e attributi A .

cancella serve ad eliminare un elemento dalla tavola. *cancella* (T, C) fornisce come risultato la tavola ottenuta da T eliminando l'elemento di chiave C (si ricordi che una chiave individua un unico elemento della tavola).

L'operazione *esiste*, applicata ad una tavola T ed una chiave C , restituisce *true* se l'elemento di chiave C è presente in T , *false* altrimenti.

L'operazione *ricerca*, applicata ad una tavola T ed una chiave C fornisce come risultato gli attributi dell'elemento identificato da C . Questa è considerata l'operazione basilare sulle tavole, e l'adeguatezza di una rappresentazione viene spesso misurata rispetto all'efficienza di questa operazione.

Si noti la stretta analogia tra la definizione di tavola e la definizione di insieme data nel par. 4: è immediato verificare che un insieme può essere definito come una tavola i cui elementi non hanno attributi.

Nel seguito di questo paragrafo esamineremo i principali metodi per la rappresentazione delle tavole. Poiché ogni elemento rappresenta un'aggregazione di dati (chiave e attributi), è naturale pensare all'uso di record per memorizzare i vari elementi della tavola: sia la chiave che gli attributi saranno rappresentati mediante un campo del record (o più campi se, a loro volta, sono composti da più dati). In genere si aggiunge un ulteriore campo ad ogni record per indicare se esso è libero, oppure se nei suoi campi sono memorizzate informazioni significative.

Nei paragrafi successivi assumeremo questo tipo di rappresentazione e ci

occuperemo dei metodi per rappresentare l'insieme degli elementi che compongono la tavola. Distingueremo quattro classi di metodi: le rappresentazioni sequenziali, collegate, ad albero, e con funzioni di accesso.

Il problema della rappresentazione delle tavole presenta una caratteristica peculiare rispetto alle altre strutture di dati. Poiché le tavole sono in genere usate per memorizzare grandi quantità di dati, per esse si utilizzano spesso tecniche di memorizzazione su memorie di massa, in modo che le informazioni presenti siano accessibili da più programmi o da più esecuzioni dello stesso programma. Per queste ragioni, nei paragrafi successivi faremo riferimento a varie tecniche di memorizzazione delle tavole su memorie di massa. Tutti i linguaggi di programmazione consentono di utilizzare i cosiddetti archivi (*file* in inglese), secondo modalità dipendenti dal linguaggio. Un archivio rappresenta un insieme di record memorizzati su memoria di massa. Esistono diversi tipi di archivi, caratterizzati essenzialmente dal modo in cui si può accedere alle loro componenti. Nei prossimi paragrafi torneremo su questo aspetto.

8.1. Rappresentazioni sequenziali

Analizziamo due metodi di rappresentazione di tavole che prevedono l'utilizzo di strutture sequenziali: la rappresentazione mediante array e quella mediante file sequenziali.

Nella prima, la tavola è rappresentata mediante un array di record, in cui ogni componente rappresenta un elemento della tavola.

L'accesso ad un elemento conoscendo la sua chiave può essere realizzato mediante la cosiddetta *ricerca esaustiva*, cioè la scansione sequenziale dell'array fino a che o si trova l'elemento cercato oppure si sono scanditi tutti gli elementi. Ovviamente, nel caso peggiore, tutto l'array deve essere analizzato: se N è il numero di elementi nella tavola, la ricerca di un elemento richiede N confronti. Nel caso medio, il numero di confronti sarà $N/2$.

Riguardo all'occupazione di memoria, si hanno le usuali limitazioni e gli usuali problemi delle rappresentazioni mediante array: è necessario imporre un limite massimo per le dimensioni dell'array e lo spazio di memoria utilizzato è fisso.

La seconda rappresentazione sequenziale fa uso di file sequenziali. Un file sequenziale è una struttura di dati che rappresenta una sequenza di record memorizzati in posizioni contigue su memoria di massa. La caratteristica principale di tali archivi è che l'accesso alle loro componenti avviene in

```
procedure ricerca (var F: file; elem: tipo_elem );
{ ricerca elem nel file F }
var trovato: boolean;
begin
  reset(F);
  trovato := false;
  while (not EOF(F)) and (not trovato)
    do if F^=elem
      then trovato := true
      else get(F)
  end;
```

Fig. 89 - La procedura *ricerca*

maniera rigidamente sequenziale: per accedere all'i-esimo elemento occorre scandire tutti gli (i-1) elementi memorizzati in posizione precedente all'i-esimo. Non esistono, in generale, limitazioni sul numero di elementi che si possono memorizzare nell'archivio.

Nel Pascal standard, ad esempio, il tipo *file* appresenta archivi sequenziali. Un file può essere utilizzato in due modalità: in lettura, quando si vuole accedere alle sue componenti senza modificarne il valore, o in scrittura quando si vuole aggiungere componenti, oppure modificarne i valori. Quando il file è utilizzato in lettura si hanno a disposizione due operatori fondamentali: *reset* e *get*. Il primo consente di posizionarsi sulla prima componente del file; il secondo consente di passare da una componente alla successiva. In scrittura, gli operatori a disposizione sono *rewrite* e *put*. Il primo serve ad azzerare il file, cioè ad eliminare tutte le sue componenti; il secondo consente di aggiungere una componente al file: dopo l'applicazione di questo operatore, la componente inserita sarà l'ultima del file. Sia in lettura che in scrittura si può utilizzare l'operatore booleano *EOF*, che consente di verificare se nell'analisi del file si è giunti alla sua ultima componente.

Per i file sequenziali una schema di ricerca di un elemento può essere quello mostrato in fig. 89.

Si noti che la procedura *ricerca* realizza un algoritmo di ricerca esaustiva. Quindi, nel caso peggiore, richiede un numero di confronti pari al numero delle componenti del file.

Una variante interessante della rappresentazione sequenziale si ottiene memorizzando gli elementi in modo ordinato nella tavola (ad esempio in ordine non decrescente rispetto al valore della chiave). Questa organizzazione consen-

te algoritmi di ricerca più efficienti: ad esempio, la ricerca esaustiva può arrestarsi appena si incontra un elemento con chiave maggiore di quella che si sta cercando. Questo accorgimento, pur consentendo un miglior comportamento dell'algoritmo in molti casi, non permette di migliorare l'efficienza nel caso peggiore: se l'elemento cercato ha una chiave maggiore di tutti gli N elementi presenti nella tavola, la ricerca esaustiva richiede sempre N confronti.

Per abbassare il numero di confronti anche nel caso peggiore, occorre adottare un diverso metodo di ricerca, la cosiddetta *ricerca binaria*. L'idea che sta alla base della ricerca binaria è la seguente: si accede all'elemento mediano della tavola (cioè, se la tavola ha N elementi memorizzati rispettivamente nelle posizioni dalla prima alla N-esima, si accede a quello memorizzato in posizione $(N/2)$ -esima) e lo si confronta col valore che si sta cercando. Se l'elemento è quello cercato, allora la ricerca si conclude con successo; se l'elemento non è quello cercato, si distinguono due casi. Se l'elemento mediano ha una chiave maggiore di quella che si sta cercando, allora la ricerca prosegue, con lo stesso metodo, nella parte di tavola formata dagli elementi in posizioni precedenti rispetto all'elemento mediano. Se, al contrario, l'elemento mediano ha una chiave minore di quella cercata, la ricerca prosegue, con lo stesso metodo, nella parte di tavola formata dagli elementi memorizzati nelle posizioni successive all'elemento mediano. È chiaro che ad ogni passo si eliminano dalla ricerca metà degli elementi che formano la tavola. Da ciò segue che, nel caso peggiore, si eseguono $\log N$ (la base del logaritmo è 2) confronti. Si veda il par. 2.1.2 del cap. 5 per una ulteriore discussione sulla ricerca binaria, e il par. 3 del cap. 6 per l'analisi della complessità.

La ricerca binaria può essere realizzata solo se il metodo di rappresentazione della tavola consente l'accesso diretto all'elemento che occupa una data posizione. La realizzazione mediante array, ad esempio, è adatta per questo tipo di ricerca, mentre la realizzazione mediante file sequenziale non la consente.

A questo proposito è bene osservare che esistono tipi di file che consentono l'accesso diretto alle componenti. Anche in alcune implementazioni del Pascal si può accedere ai file in modo diretto, utilizzando l'informazione relativa alla posizione (cioè il numero d'ordine) del record cercato. In queste implementazioni è disponibile un operatore del tipo: *seek (F,I)*, dove F è un file e I è un intero, che consente l'accesso all'elemento I-esimo del file F. Utilizzando l'accesso diretto, e conoscendo il numero di elementi memorizzati nel file, si può eseguire la ricerca binaria anche su una tavola realizzata mediante file.

Si osservi, infine, che la necessità di mantenere la tavola ordinata richiede una maggiore complessità delle operazioni di inserimento e cancellazione. Rife-

rendoci, ad esempio, alla realizzazione mediante array, l'inserimento di un elemento può richiedere lo spostamento di tutti quelli con chiave maggiore. Analoghe considerazioni valgono per l'operazione di eliminazione.

8.2. Rappresentazioni collegate e rappresentazioni ad albero

Un primo metodo di rappresentazione collegata di una tavola prevede l'uso di una lista semplice, in modo che ogni elemento della lista corrisponda ad un elemento della tavola. La ricerca di un elemento viene realizzata mediante la ricerca esaustiva: la lista semplice viene scandita, confrontando ogni volta la chiave dell'elemento analizzato con la chiave dell'elemento cercato.

Analogamente a quanto osservato per la rappresentazione sequenziale, se gli elementi della tavola sono memorizzati in ordine non decrescente rispetto al valore della chiave, la ricerca può concludersi appena si trova un elemento con chiave maggiore o uguale a quella cercata.

Riguardo all'accesso di elementi, quindi, la rappresentazione collegata non presenta vantaggi rispetto alla rappresentazione sequenziale. Al contrario, le operazioni di inserimento e cancellazione, possono essere eseguite molto più efficientemente (si veda il par. 3.3 dedicato alla rappresentazione collegata di una lista).

Si osservi anche che l'occupazione di memoria richiesta per la memorizzazione della tavola aumenta rispetto alla rappresentazione sequenziale: è necessario, infatti, riservare memoria per i puntatori (o, comunque, per i riferimenti agli elementi).

Una tavola può essere anche realizzata mediante un albero. Ogni nodo dell'albero corrisponde ad un elemento della tavola e l'accesso ad un elemento avviene secondo un prefissato algoritmo di visita. Molto vantaggiosa risulta la realizzazione mediante albero binario di ricerca. Questa rappresentazione consente un veloce algoritmo di ricerca (si veda il par. 4 del cap. 6), che, tuttavia, è efficiente solo se l'albero è bilanciato (cioè se tutti i cammini dalla radice alle foglie hanno circa la stessa lunghezza). La gestione di una tavola mediante albero richiede quindi che le operazioni di inserimento e cancellazione vengano effettuate mantenendo il bilanciamento dell'albero, ottenuto con opportune ristrutturazioni dell'albero stesso.

8.3. Rappresentazioni con funzioni di accesso

Nelle rappresentazioni con funzioni di accesso gli elementi della tavola sono memorizzati senza un ordine prefissato e l'accesso al singolo elemento viene eseguito utilizzando una funzione, detta appunto *funzione di accesso*, che, data la chiave dell'elemento cercato, restituisce un indirizzo di memoria.

Indicando con *indirizzo* l'insieme delle posizioni di memoria in cui sono memorizzati gli elementi della tavola, si ha:

$$\text{funzione_di_accesso}: \text{chiave} \rightarrow \text{indirizzo}$$

Distingueremo nel seguito due casi: il primo corrisponde alla situazione in cui la funzione sia una corrispondenza biunivoca, cioè faccia corrispondere indirizzi diversi a chiavi diverse. Il secondo si riferisce alla situazione in cui la funzione può calcolare lo stesso indirizzo per due o più chiavi diverse.

Nel caso in cui la funzione di accesso sia una corrispondenza biunivoca tra chiavi ed indirizzi, essa viene detta funzione di accesso diretto. Si noti che in questo caso, per realizzare la tavola, si deve disporre di una posizione di memoria per ogni elemento.

Come semplice esempio di questa rappresentazione, possiamo considerare una tavola realizzata mediante array in cui sia definita una corrispondenza biunivoca tra chiavi e indici dell'array: data una chiave, la funzione di accesso calcola l'indice della posizione del vettore riservata all'elemento corrispondente al valore della chiave.

Un altro esempio può essere la realizzazione mediante file, se a tale file è possibile accedere mediante i numeri d'ordine dei record, e se è definita una funzione che stabilisce la corrispondenza tra chiavi e numeri d'ordine dei record.

In questa rappresentazione l'operazione di accesso è molto efficiente, poiché non dipende dal numero di elementi presenti nella tavola. Molto efficienti sono anche le operazioni di inserzione e cancellazione, poiché ogni potenziale elemento della tavola ha una posizione di memoria ad esso riservata. Quest'ultima osservazione ci chiarisce quale sia il principale difetto della rappresentazione con accesso diretto: essa richiede che la struttura in cui si memorizzano gli elementi della tavola consenta di memorizzare tanti elementi quanti sono i possibili valori della chiave. Consideriamo, ad esempio, una tavola per la memorizzazione dei dati relativi agli studenti iscritti ad una facoltà. Supponiamo che tali studenti siano al massimo 3.000, e che la chiave sia il cognome degli studenti. Assumendo che ogni cognome non sia più lungo di 15 caratteri, il

numero di chiavi possibili è 2^6 elevato alla 15. È evidente che, con queste ipotesi, il metodo con funzione di accesso diretto è impraticabile.

Per queste ragioni, questo metodo di rappresentazione viene utilizzato quando il numero di elementi presenti di media nella tavola è circa uguale al numero di chiavi possibili. In queste condizioni, infatti, il metodo è particolarmente efficiente rispetto sia all'occupazione di memoria, che alle operazioni di accesso e di aggiornamento della tavola.

Per ovviare all'inconveniente appena citato è necessario predisporre per la tavola una zona di memoria la cui dimensione dipende dal numero di elementi mediamente presenti nella tavola. La funzione di accesso sarà, in questo caso, definita in modo che a chiavi diverse possa corrispondere lo stesso indirizzo. Se K_1 e K_2 sono due chiavi diverse, e fun è la funzione di accesso, sarà, perciò, possibile che:

$$fun(K_1) = fun(K_2)$$

Una tale situazione è detta *collisione* tra K_1 e K_2 . I problemi che sorgono in questo tipo di rappresentazione sono due:

1. determinare una funzione di accesso che riduca al massimo la possibilità di collisione;
2. individuare un metodo di scansione della tavola (o per cercare un elemento o per individuare una posizione libera dove inserire un nuovo elemento) qualora si verifichi una collisione.

Riguardo al primo punto, è evidente che l'obiettivo da raggiungere è progettare una funzione che distribuisca gli indirizzi calcolati in modo più uniforme possibile su tutto l'insieme di indirizzi disponibili. Per questo scopo è necessario che il valore restituito dalla funzione dipenda da tutta la chiave, e non solo da sue parti. Funzioni di questo tipo vengono realizzate mediante opportuni calcoli effettuati sui caratteri che formano la chiave, e vengono dette *funzioni hash* (dall'inglese *to hash*, cioè tritare o rimescolare). Due classici esempi di funzioni hash sono:

1. data la chiave K , si considera la sua rappresentazione binaria e si estraggono M bit al centro (supponendo che M bit siano sufficienti per rappresentare gli indirizzi di tutte le posizioni di memoria riservate agli elementi della tavola);
2. data la chiave K , si calcola il resto della divisione di k (rappresentazione binaria di K) per il numero massimo (previsto) N di elementi della tavola. Si noti che è opportuno che N non sia una potenza di 2: infatti, se $N=(2$ elevato a $L)$ è sufficiente che due chiavi coincidano negli ultimi L bit perché

siano coinvolti in una collisione. In questo caso, il valore restituito dalla funzione dipenderebbe solo dagli ultimi L bit della chiave, contraddicendo i criteri fondamentali per la realizzazione di funzioni *hash*.

Riguardo al secondo punto, diversi sono i metodi di scansione della tavola che possono essere adottati quando si verificano collisioni.

La più semplice è la *scansione lineare*. Supponiamo di voler cercare l'elemento di chiave K nella tavola T e sia *hash* la funzione utilizzata per l'accesso. Le operazioni da eseguire sono: si calcola $\text{hash}(K)=\text{ind}$; se nella tavola T, in posizione *ind*, c'è un elemento di chiave K_1 , diversa da K, allora si analizzano in sequenza le posizioni di indirizzo $\text{ind}+h$ (dove h è un prefissato numero intero positivo), poi $\text{ind}+2+h$, poi $\text{ind}+3+h$, e così via, fino a che si trova l'elemento di chiave K oppure una posizione libera della tavola, oppure ancora un elemento già analizzato. Si noti che, al fine di garantire l'analisi di tutte le posizioni della tavola, si deve fare in modo che h sia un numero primo rispetto alla dimensione della zona di memoria riservata agli elementi della tavola.

Un inconveniente della scansione lineare è il fenomeno della agglomerazione, cioè il fenomeno per cui la sequenza di indirizzi scanditi per la chiave K (cioè $\text{ind}+i+h$, per valori crescenti di i) è uguale a quella per le chiavi K' , K'' , ..., tali che $\text{hash}(K')=\text{ind}+h$, $\text{hash}(K'')=\text{ind}+2*h$, e così via.

Un altro metodo di scansione prevede l'uso di *aree di trabocco*, cioè di aree di memoria destinate alla memorizzazione degli elementi che, in inserimento, hanno dato collisione. Con questo metodo, la ricerca di un elemento di chiave K nella tavola T, avviene così: si calcola $\text{hash}(K)=\text{ind}$; se in posizione *ind*, c'è un elemento di chiave K_1 , diversa da K, allora si accede all'area di trabocco associata alla posizione *ind* e si comincia una nuova ricerca dell'elemento in

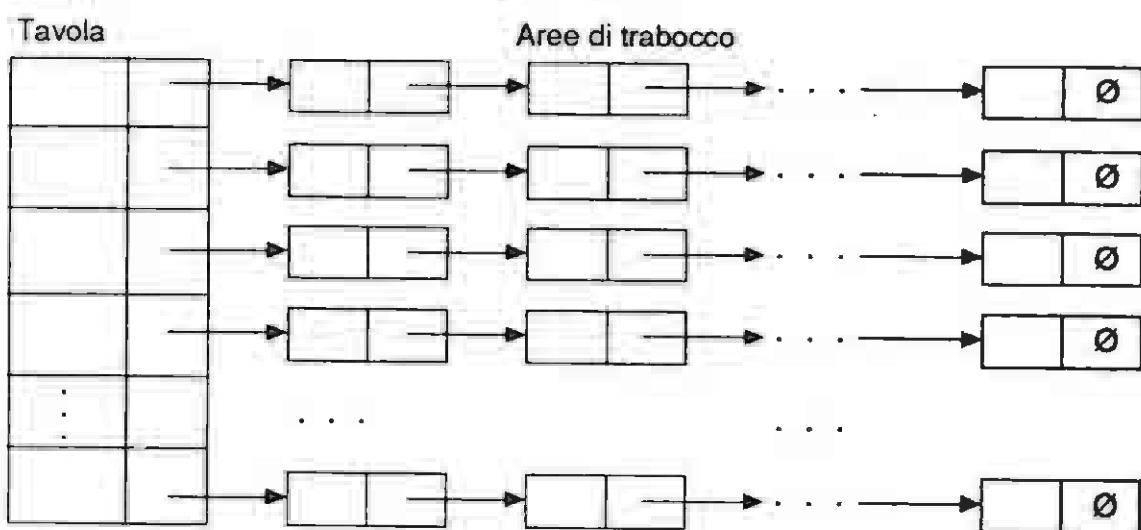


Fig. 90 - Tavola con aree di trabocco

tale area. Si può utilizzare un'unica area di trabocco per tutta la tavola oppure (e questa è la soluzione più comune) più aree di trabocco. L'area di trabocco può essere considerata una vera e propria tavola, e la ricerca di un suo elemento può avvenire con varie modalità: ricerca esaustiva, ricerca binaria, oppure con un'ulteriore funzione *hash*.

In fig. 90 mostriamo una rappresentazione di una tavola in cui le aree di trabocco (in questo caso una per ogni posizione della tavola principale) sono organizzate secondo una rappresentazione collegata. In questo caso, la ricerca nell'area di trabocco sarà una ricerca esaustiva.

Parte III

Metodologie

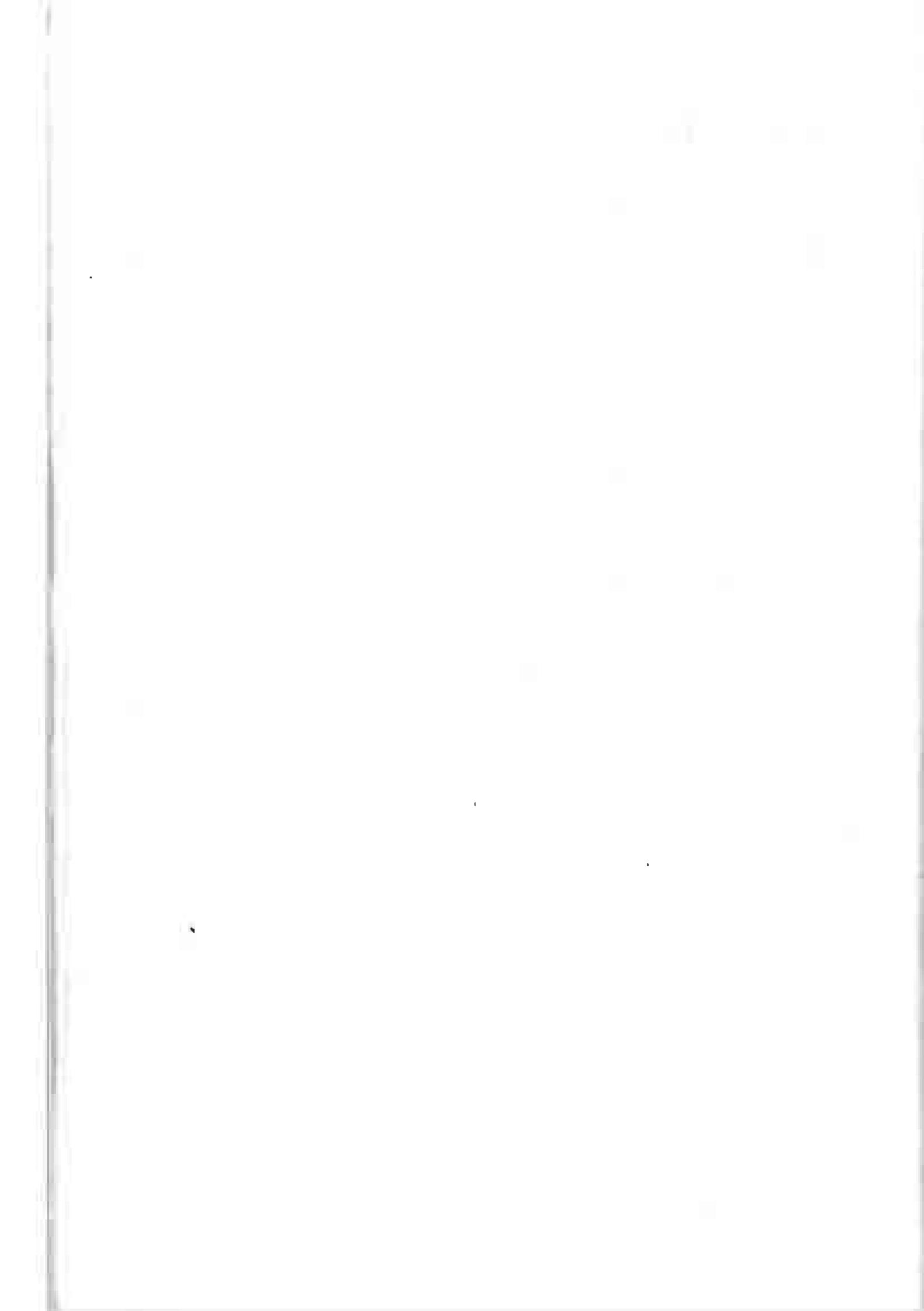
di programmazione

4. Metodologie di progetto di programmi

5. Analisi di programmi: la correttezza

6. Analisi di programmi: la complessità

7. Elementi di calcolabilità



4.

Metodologie di progetto di programmi

Nei precedenti capitoli sono stati descritti gli strumenti linguistici per esprimere l'algoritmo risolutivo di un problema per mezzo di un programma. Talvolta non è semplice utilizzare questi strumenti per progettare un programma a partire dalla specifica di un problema. Abbiamo perciò bisogno di metodi, che ci forniscano una razionalità per utilizzare i linguaggi programmativi. Lo scopo di questi metodi è duplice: semplificare la progettazione degli algoritmi risolutivi di problemi e produrre programmi di elevata qualità.

Per quanto riguarda il primo punto, esso ha un diretto impatto sul costo della produzione del software: è noto che, a fronte di una vertiginosa diminuzione dei costi dell'hardware, il costo del software per unità di istruzione è diminuito molto più lentamente. Per quanto riguarda il secondo punto, i programmi, alla stregua degli altri prodotti industriali sono caratterizzati da qualità caratteristiche del particolare scopo per cui essi sono prodotti.

Obiettivo di questo capitolo è la descrizione delle metodologie sviluppate per il raggiungimento dei due precedenti scopi. In particolare: nel par. 1 descrivremo le fasi di cui si compone quello che viene chiamato il ciclo di vita di un programma e le qualità che un programma deve avere; nel par. 2 esamineremo i principi generali che presiedono al progetto di un programma: l'uso di astrazioni e della modularità. Nel par. 3 descriveremo un insieme di metodologie che permettono il progetto di un programma mediante l'uso di astrazioni. Approfondiremo lo studio della modularità nel par. 4, in cui si introducono alcuni criteri per verificare il grado di modularità di un programma. Fino a questo punto del capitolo le tecniche di progetto sono state esaminate da un punto di vista generale, indipendente dal linguaggio programmatico scelto per la codifica. Il par. 5 descrive i criteri da utilizzare per la codifica di un programma in un linguaggio tipo Pascal, dotato di strutture di controllo con

particolari caratteristiche di strutturazione. Il par. 6 descrive un insieme di tecniche per documentare i programmi e renderli facilmente comprensibili. Infine, il par. 7 riassume gli argomenti trattati descrivendo una metodologia di natura generale per il progetto di programmi.

1. LE QUALITÀ DI UN PROGRAMMA

Possiamo distinguere due categorie di programmi: quelli realizzati in base ad esigenze estemporanee, quali la interrogazione di un archivio, e quelli che scaturiscono da necessità pianificate, e richiedono uno sforzo di progetto misurabile in un arco temporale di giorni, mesi, ed anche, in progetti molto grandi, in decine di anni uomo (un anno uomo è la misura del lavoro che una persona effettua in un anno).

I programmi del primo tipo sono sempre più diffusi, con l'avvento di linguaggi che permettono una progettazione rapida, e possono considerarsi programmi “usa-e-getta”, proprio in virtù del loro carattere estemporaneo. Questo tipo di programmi è scritto in genere da una singola persona. A questa categoria appartengono per esempio:

1. una interrogazione effettuata con un menu di comandi, in cui dobbiamo solo fornire il nome dell'archivio ed il tipo di interrogazione;
2. la produzione di un istogramma a partire da una matrice di valori;
3. la effettuazione di statistiche mediante un pacchetto già predisposto, cui bisogna solo comunicare i dati di ingresso ed il tipo di statistiche.

I programmi del secondo tipo si possono a loro volta distinguere in due categorie: programmi progettati da una singola persona in un arco di tempo limitato (giorni o settimane), e programmi progettati da gruppi di persone ognuno dei quali è in genere responsabile di un insieme di funzionalità limitato rispetto alla dimensione del programma. Per i primi si usa parlare di *programmazione in piccolo (programming in the small)* e per i secondi di *programmazione in grande (programming in the large)*. Le differenze fondamentali fra questi due tipi di programmazione sono la maggiore importanza nel secondo tipo del coordinamento e della comunicazione tra i diversi componenti il gruppo di progetto, ed il fatto che i programmi così realizzati hanno in genere una vita che va al di là della loro primitiva progettazione: una volta progettati e realizzati, essi vengono utilizzati per molto tempo, ed in genere possono venire modificati per rispondere ad esigenze che cambiano nel tempo. Da questa osservazione nasce la maggiore importanza di metodologie (in genere

sviluppate in ambito industriale) fortemente strutturate ed analitiche nello stabilire compiti, forme di cooperazione e modalità di documentazione. In questo testo, ci occuperemo soprattutto di programmazione in piccolo, anche se varie considerazioni metodologiche saranno chiaramente di natura generale.

Come avviene per altri prodotti industriali, i programmi vengono realizzati attraverso una sequenza di attività che sono tradizionali in ogni applicazione ingegneristica. Tali attività compongono il "ciclo di vita" del programma, e possono suddividersi in:

1. *specifica dei requisiti*, in cui viene definito con precisione il problema da risolvere;
2. *scelta del sistema di calcolo*, che ha come scopo la scelta dell'*ambiente di programmazione* (linguaggio, compilatore, configurazione dell'hardware) in cui sviluppare il programma e l'*ambiente di produzione* in cui il programma sarà stabilmente utilizzato;
3. *progetto*, in cui vengono scelti l'algoritmo risolutivo e i tipi di dati, e, attraverso successivi passi, si giunge alla codifica nel linguaggio programmatico;
4. *test e correzione*, fase nella quale il programma viene provato, per verificare la sua rispondenza alle specifiche del problema, ed in caso negativo viene adeguatamente corretto;
5. *manutenzione*, in cui, periodicamente, quando per qualche ragione le specifiche del problema cambiano, il programma viene aggiornato per rispondere alle nuove specifiche. Altri interventi di manutenzione si verificano quando vengono saltuariamente scoperti errori nel programma, che non sono stati evidenziati nella fase 4, e che devono evidentemente essere corretti.

Il tempo (e quindi il costo) relativo delle varie fasi è mostrato in fig. 1. Vengono fornite delle fasce di costo, a causa della grande variabilità dei costi in funzione dei tipi di progetti. Per esempio, in alcuni progetti i costi di manutenzione possono anche giungere al 50-60 per cento, quando, pur rimanendo relativamente stabili le applicazioni coinvolte dal progetto, le specifiche di dettaglio cambiano con molta frequenza nel tempo. Si noti, in ogni caso, la grande rilevanza che presenta la fase di test e correzione, che può raggiungere anche la metà dell'intero costo del progetto.

Come ogni altro prodotto, i programmi devono rispondere a determinate qualità. Le qualità più importanti sono:

1. *correttezza*, intesa come aderenza alle specifiche del problema, provata in modo formale;

<i>fase del ciclo di vita</i>	<i>percentuale di tempo</i>
specificità dei requisiti	5 - 10
scelta del sistema di calcolo	2 - 5
progetto	25 - 40
test e correzione	20 - 50
manutenzione	20 - 60

Fig. 1 - Tempo relativo speso nelle varie fasi del ciclo di vita di un programma

2. *leggibilità*, intesa come capacità di far comprendere in ogni momento al progettista e/o ad altre persone le scelte fatte per quanto riguarda l'algoritmo risolutivo e la sua codifica nel linguaggio;
3. *efficienza*, intesa come capacità del programma di calcolare i risultati a partire dai dati di ingresso con limitato uso di risorse di calcolo;
4. *modularità*, intesa come grado di organizzazione interna e strutturazione del programma, che permetta di comprendere come le singole parti cooperino per lo scopo generale.

Accanto ad esse, altre qualità importanti sono:

5. *modificabilità*, intesa come facilità di adeguamento e trasformazione del programma quando cambiano le specifiche del problema;
6. *parametricità*, intesa come grado di generalità che il programma presenta nella risoluzione, più che di un solo problema, di una classe di problemi, così che la sua realizzazione e utilizzazione risulti economicamente utile;
7. *portabilità*, intesa come possibilità di eseguire il programma anche in ambienti di calcolo diversi da quello per cui è stato inizialmente progettato;
8. *robustezza*, intesa come capacità del programma di operare in maniera prevedibile, nonostante l'introduzione di dati non coerenti con le specifiche.

Ognuna delle precedenti qualità esamina il programma da un diverso angolo di visuale, e ne coglie alcune caratteristiche significative. Alcune volte, può capitare che non sia possibile garantirle tutte in eguale misura: ad esempio, intuitivamente, quanto più cerchiamo di rendere un programma parametrico, tanto più alcune volte dobbiamo sacrificare la efficienza. Quanto più vogliamo aumentare la efficienza, tanto più dovremo adottare talvolta soluzioni che ne

riducono la leggibilità.

Accanto alle precedenti qualità ne introduciamo un' altra, relativa più che al programma al suo progetto, e che possiamo chiamare *semplicità di progettazione*, intesa come possibilità di individuare semplicemente il programma a partire dal corrispondente problema.

Come abbiamo notato nella introduzione, per raggiungere le qualità descritte, è necessario disporre di metodologie di sviluppo dei programmi che aiutino ed orientino il progettista in tutto il loro ciclo di vita. Per metodologia di sviluppo intendiamo un insieme di fasi operative, strategie, verifiche che permettano di sviluppare i programmi richiesti dalle specifiche in modo economico e con elevata qualità. In questo testo concentreremo il nostro interesse sulle metodologie utilizzate nella fase di progetto. In particolare, in questo capitolo tratteremo gli aspetti riguardanti la semplicità di progettazione, la modularità e la leggibilità dei programmi. Per quanto riguarda il primo aspetto, già nel cap. 3 abbiamo trattato il problema del progetto dei dati per mezzo di livelli di astrazione; in questo capitolo riprenderemo quelle considerazioni nel contesto più generale del progetto degli algoritmi e programmi. Gli aspetti riguardanti le altre due qualità più importanti, la correttezza e la efficienza, saranno trattati nei capp. 5 e 6 rispettivamente.

2. PRINCIPI DI PROGETTO

Come abbiamo già detto, progettare programmi può essere una attività complessa, che richiede una notevole creatività, cultura e disciplina. Dato un problema, è difficile talvolta individuarne anche un solo algoritmo risolutivo; in generale può essere ancora più difficile, tra diversi algoritmi risolutivi che potrebbe avere il problema, trovare quello o quelli che hanno le qualità desiderate.

È bene premettere che il processo di progettazione di un algoritmo è sempre connesso ad una comprensione approfondita del problema e delle proprietà dei dati coinvolti. Tali proprietà sono determinanti nella scelta di un buon algoritmo risolutivo.

Per guidare il processo di progettazione dobbiamo essere in grado di dominare la vastità delle scelte da effettuare, procedendo per passi, avendo a che fare ogni volta con poche scelte fondamentali, e garantendo al programma che si va man mano formando una struttura e un ordine interno che lo renda semplice da comprendere. In sintesi, possiamo dire che i due principi fonda-

mentali di progetto sono: procedere per livelli di astrazione, e garantire al programma una elevata strutturazione e modularità.

Affrontiamo separatamente i due aspetti.

Procedere per livelli di astrazione. Una astrazione, nel contesto che ci interessa, può vedersi come un processo di generalizzazione in cui ci concentriamo sulle similarità presenti tra un insieme di oggetti, escludendo gli aspetti che li diversificano, e giungendo in questo modo alla costruzione di un nuovo oggetto che, appunto, li generalizza. Il progetto di programmi può procedere descrivendo inizialmente in maniera astratta l'algoritmo e i dati su cui essa opera, e procedendo poi ad un raffinamento di tale descrizione astratta in termini di descrizioni via via più dettagliate e precise.

I tipi di astrazioni che possiamo utilizzare nel progetto dei programmi sono tre: astrazione funzionale, astrazioni sui dati, astrazioni sul controllo. Più precisamente:

1. l'*astrazione funzionale* viene utilizzata quando un insieme di istruzioni oppure operazioni complesse su tipi di dato vengono espresse mediante una procedura o una funzione;
2. le *astrazioni sui tipi di dato* utilizzano tipi di dato astratti con le relative operazioni, ignorando gli aspetti riguardanti la rappresentazione e le manipolazioni ad esse relative;
3. le *astrazioni sul controllo* consistono nell'utilizzare comandi ad alto livello senza specificare l'esatto meccanismo di implementazione. Ad esempio, un comando del tipo:

Per tutti gli elementi dell'insieme I,
fai l'azione A

può vedersi come una struttura di controllo astratta, in cui non specifichiamo l'ordine e le modalità con cui vengono esaminati gli elementi dell'insieme.

In una strategia che proceda per livelli di astrazione, possiamo vedere il progetto del programma come una linea spezzata in un piano di progetto, in cui in ascissa rappresentiamo insieme i livelli di astrazione funzionale e sul controllo, ed in ordinata il livello di astrazione sui tipi di dato (vedi fig. 2: ogni passo del progetto dà luogo ad un nuovo raffinamento, relativo o alla struttura del programma o a qualche tipo di dati).

Per esemplificare il concetto di piano di progetto, supponiamo di voler leggere un insieme di valori interi, e di stampare, tra essi, quelli dispari. Possiamo inizialmente procedere lungo il livello di astrazione funzionale,

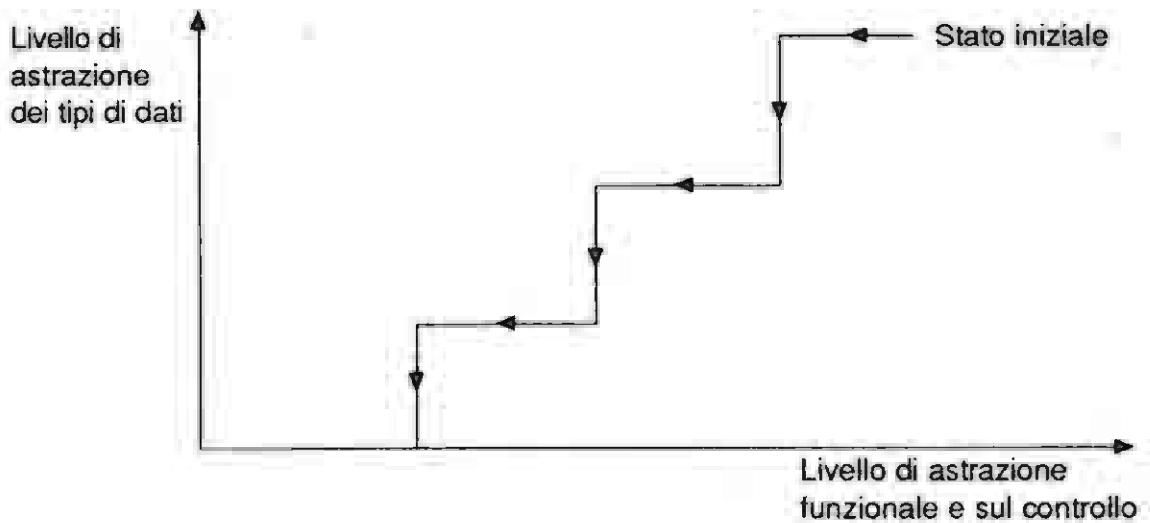


Fig. 2 - Il piano di progetto

dando luogo al programma seguente:

1. Leggi l'insieme di valori A;
2. Stampa tutti i numeri dispari di A.

Possiamo ora raffinare ulteriormente la funzionalità 2 agendo sul controllo nel modo seguente:

1. Leggi l'insieme di valori A
2. Per tutti i numeri nell'insieme A:

Se il numero è dispari, stampalo.

Effettuiamo un raffinamento sul tipo di dato. Finora abbiamo rappresentato i dati del problema per mezzo del tipo astratto *insieme*. Dobbiamo a questo punto scegliere una rappresentazione per tale tipo. Se A ha cardinalità nota, possiamo decidere di rappresentarlo per mezzo di un **array** di interi. Questo ci permette facilmente di esprimere le due funzionalità per mezzo di due cicli **for**, uno di lettura, l'altro di scansione, scelta e stampa. A questo punto la versione finale del programma può essere facilmente ottenuta.

Strutturare il programma. Il meccanismo fondamentale con cui possiamo dare una struttura ad un programma, come abbiamo visto nel cap. 2, consiste nell'uso di una **procedura** o di una **funzione**. Nel seguito usiamo il termine generale di **modulo** per indicare tale meccanismo. Le modalità con cui l'insieme delle istruzioni di un programma può essere suddiviso e strutturato in un insieme di moduli sono influenzate da vari criteri. Essi sono:

1. un primo criterio, detto di *indipendenza funzionale*, consiste nel rendere massima la indipendenza dei vari moduli di cui il programma è composto.

Quanto più ogni modulo esegue una funzione ben precisa all'interno del programma, indipendente da quella svolta da altri moduli, tanto più il programma nel suo complesso è facilmente comprensibile. L'indipendenza funzionale può a sua volta essere espressa in termini di due criteri: la massimizzazione della *coesione* delle singole funzionalità elementari svolte all'interno di un modulo e la minimizzazione dell'*accoppiamento* tra moduli, dovuto alle informazioni tra essi scambiate. L'esame di questi due criteri verrà approfondito nel par. 4;

2. il criterio della *decomposizione gerarchica* del programma in moduli che realizzano funzioni più semplici e locali ad un particolare aspetto del problema. In un programma decomposto per mezzo di un insieme di moduli che si richiamano gerarchicamente secondo una struttura ad albero, la funzione svolta da ogni modulo è più facilmente comprensibile e collocabile in relazione a quella svolta dagli altri;
3. il criterio dell'*occultamento della informazione (information hiding)* secondo il quale ogni modulo comunica con gli altri per mezzo di un insieme ristretto di parametri, "nascondendo" le modalità con cui vengono effettuate le elaborazioni al suo interno;
4. il criterio delle *astrazioni sui tipi di dato*, in base al quale particolari moduli possono vedersi come una rappresentazione di un tipo di dato astratto, e gli altri moduli interagiscono con esso solo tramite le funzioni definite sul tipo.

Garantire al programma una elevata modularità, permette il raggiungimento di altre qualità: la *leggibilità*, per le ragioni più volte esposte; la *modificabilità*, poiché è più semplice individuare in programmi modulari le funzionalità da modificare; la *correttezza*, poiché è più semplice isolare gli errori.

In base a quanto detto in precedenza, dovrebbe essere chiaro che procedere per livelli di astrazione e strutturare il programma in moduli sono due diversi punti di vista dello stesso principio generale: infatti, procedendo dall'astratto al concreto, è in generale garantita la produzione di programmi altamente strutturati; d'altra parte, per poter garantire una buona modularità al programma, dobbiamo in ogni momento avere una visione globale, che si può ottenere soltanto procedendo per raffinamenti successivi.

3. METODOLOGIE DI PROGETTO

La metodologia di progetto che si ispira direttamente al principio di astrazione introdotto nel paragrafo precedente è detta metodologia (o strategia) *top-*

down (che significa ‘dall’alto in basso’): ad essa sarà prevalentemente dedicato questo paragrafo. Parleremo anche della strategia *bottom-up* (‘dal basso verso l’alto’), e saremo quindi in grado di effettuare un confronto tra le due strategie.

La strategia top-down propone di decomporre iterativamente il problema in sottoproblemi, la cui soluzione congiunta costituisce la soluzione per il problema generale, utilizzando l’astrazione funzionale, quella sui dati e sul controllo, e proseguendo nella decomposizione fin quando ogni singolo sottoproblema ha una struttura così semplice che può direttamente essere espresso per mezzo di istruzioni del linguaggio programmatico.

Per quanto riguarda le modalità con cui procedere nell’usare le tre astrazioni, nel cap. 3 abbiamo già trattato gli aspetti riguardanti i tipi di dato. Riguardo alle astrazioni funzionale e sul controllo, nella fig. 3 sono mostrate alcune tipiche decomposizioni utilizzate nel progetto top-down.

La prima decomposizione può vedersi come una applicazione della astrazione funzionale, mentre le successive come applicazioni delle astrazioni sul controllo. Formuliamo alcuni criteri intuitivi per guidare la scelta tra le diverse decomposizioni:

1. adottiamo una *decomposizione funzionale* ogni volta che, in base ai criteri esposti nel par. 3, stabiliamo che la funzionalità espressa dalla frase risponda a qualche criterio di modularità significativo. Una importante classe di decomposizioni funzionali è quella che corrisponde alla scelta di una procedura o funzione ricorsiva, nel qual caso la procedura o funzione coincide con una già presente, generata in raffinamenti precedenti. Sceglieremo una soluzione ricorsiva ogni volta questa scaturisce dalla struttura dei dati su cui il problema opera e dalla natura del problema; in questo caso, la soluzione può esprimersi per mezzo di un passo base, che esprime la condizione iniziale dell’algoritmo, e di un passo ricorsivo espresso per mezzo di attivazione ricorsiva di procedura o funzione;
2. adottiamo la *decomposizione per sequenza* quando dalla analisi del problema ci accorgiamo che esso può essere decomposto in un insieme di attività elementari che devono essere eseguite una dopo l’altra;
3. adottiamo la *decomposizione per alternativa* quando il problema può esser decomposto in termini di una scelta tra diverse alternative;
4. adottiamo la *decomposizione per iterazione* quando si individua un raffinamento in cui una azione va eseguita più volte.

Decomposizione funzionale

<frase in linguaggio naturale> ==> *<attivazione di procedura>/<attivazione di funzione>*

Decomposizione per sequenza

<frase in linguaggio naturale> ==> **begin**

<frase>;

<frase>;

 .

<frase>

end

Decomposizione per iterazione

<frase in linguaggio naturale> ==> **while** *<condizione>*
 do *<frase>*

<frase in linguaggio naturale> ==> **repeat** *<frase>*
 until *<condizione>*

<frase in linguaggio naturale> ==> **for** *<intervallo di valori>*
 do *<frase>*

Decomposizione per alternativa

<frase in linguaggio naturale> ==> **if** *<condizione>*
 then *<frase>*
 else *<frase>*

dove *<frase>* può essere:

1. una frase in linguaggio naturale
2. un insieme di istruzioni Pascal
3. una attivazione di procedura o funzione

Fig. 3 - Decomposizioni del metodo top-down

Primo raffinamento top-down

Cerca *val* nel vettore *V*

Se è valore di un elemento, stampa un messaggio

Secondo raffinamento top-down

Per tutti gli elementi del vettore *V*:

confronta l'elemento con *val*

se il valore è uguale, ricordatelo

Se hai trovato almeno un elemento con valore uguale

allora stampa messaggio

Fig. 4 - Esempio di progetto top-down

Applichiamo la strategia top-down per un semplice problema:

Dato un valore *val* e un vettore *V*,
stampare un messaggio se *val* è valore di un elemento di *V*

In un primo raffinamento (vedi fig. 4) possiamo applicare una decomposizione per sequenza, introducendo due attività principali, la ricerca e la decisione, astraendo dalla loro realizzazione. Ad esempio, la ricerca potrebbe essere effettuata procedendo dal primo all'ultimo elemento, ovvero dall'ultimo al primo, ovvero, se sappiamo che gli elementi sono ordinati, con un metodo binario: non ci interessa a questo punto entrare in tali dettagli. Nel secondo raffinamento esprimiamo le due attività precedenti in termini di attività più elementari.

La strategia bottom-up costruisce l'algoritmo risolutivo individuando inizialmente un insieme di passi o funzionalità elementari, e componendo successivamente tali funzionalità in frammenti più grandi, fino alla individuazione dell'intero algoritmo. L'uso delle astrazioni è anche tipico della strategia bottom-up: in questo caso, esse vengono utilizzate per generare versioni più astratte di frammenti del programma a partire da versioni di dettaglio. Man mano che individuiamo un nuovo frammento di algoritmo o un nuovo tipo di dato, possiamo esprimere per mezzo di uno dei meccanismi di astrazione visti in precedenza. Nella strategia bottom-up, una visione complessiva potrà essere ottenuta soltanto alla fine del progetto, quando si ha a disposizione l'insieme

Passo 1: Primo insieme di operazioni individuate

- O1: *val* è uguale a $V[i]$?
- O2: ricordati che l'hai già trovato
- O3: hai finito di scandire il vettore V ?

Passo 2: Nuova operazione individuata

- O4: avanza nel vettore

Passo 3: Composizione delle precedenti operazioni

- O12: se O1 allora O2
- O1234: repeat O12; O4
 - until O3

Passo 4: Nuove operazioni da aggiungere

- O5: comincia a scandire il primo elemento
- O6: se l'hai trovato, stampa messaggio

Passo 5: Algoritmo finale

- O123456: O5;
- O1234;
- O6.

Fig. 5 - Esempio di progetto bottom-up

dei moduli per l'intero problema. Tipico atteggiamento di progetto che possiamo associare alla strategia bottom-up è anche quello di riutilizzare programmi già scritti per altri problemi, adattandoli alle specifiche del nuovo problema.

Una possibile strategia bottom-up per il problema precedente è mostrata in fig. 5. Qui individuiamo dapprima un insieme di operazioni elementari (passo 1) che pensiamo utili per la soluzione del problema. Tentando di comporre le operazioni O1, O2, O3 in una operazione di ciclo, ci accorgiamo che manca una operazione di avanzamento nel vettore (passo 2, operazione O4). Possiamo ora riunire le operazioni in una operazione più astratta di ricerca (passo 3, operazione intermedia O12, e operazione finale O1234). Ai fini della soluzione

dell'intero problema, mancano ancora una operazione di inizializzazione, ed una di decisione (passo 4). A questo punto possiamo costruire l'algoritmo finale (passo 5).

A partire dalle considerazioni ed esempi precedenti possiamo dare alcuni criteri di confronto delle due strategie.

1. *Semplicità di progettazione* - La strategia top-down permette di concentrare l'attenzione, in ogni passo del progetto, sugli aspetti che sono significativi in quel momento, e rimandare a passi successivi gli aspetti di maggior dettaglio. Come conseguenza, risponde in maniera ideale alla tipica esigenza di analizzare in ogni momento un insieme limitato di aspetti di un problema. I programmi scritti con il metodo top-down sono facili da comprendere, perché l'insieme dei raffinamenti è anche una importante documentazione di progetto. La strategia bottom-up può rivelarsi talvolta utile quando, in virtù della complessità del problema, si voglia cominciare a fissare qualche aspetto dell'algoritmo, così da semplificare l'analisi successiva;
2. *Modularità* - La modularità è più facilmente raggiungibile con la strategia top-down, che procede per raffinamenti di funzionalità ad alto livello in termini di gruppi di funzionalità più elementari. Con la strategia bottom-up, una volta costruito l'intero programma, si dovrà in generale analizzarlo nuovamente per rivedere la divisione in moduli rispetto a quella ottenuta inizialmente a seguito di scelte locali;
3. *Modificabilità* - È più facile da raggiungere con la strategia top-down, perché in questo caso si potrà ripercorrere le scelte di progetto e sarà perciò più semplice, a fronte di un progetto condotto in modo strutturato, individuare quali scelte debbano essere modificate ed in che modo;
4. *Efficienza* - Quando si usa il metodo top-down, è possibile avere fin dall'inizio del progetto una valutazione generale sulla efficienza dell'algoritmo globale; se si è intrapresa una soluzione inefficiente, occorre subito riconsiderare le scelte effettuate nei raffinamenti precedenti, perché sarà difficile successivamente fare ristrutturazioni che migliorino la situazione. Tornando all'esempio di fig. 2, la strategia per cui prima si analizza l'intero vettore e poi si prende la decisione potrebbe essere sostituita da quella in cui, se si trova il valore si abbandona subito la ricerca. Nel metodo bottom-up abbiamo un immediato riscontro relativamente alla efficienza delle singole funzionalità, ma possiamo fornire una valutazione sull'intero algoritmo solo alla fine del progetto;
5. *Influenza degli errori* - Un errore commesso nella strategia top-down, se non

individuato, si riproduce amplificato per tutto il progetto, con effetti tanto più critici quanto prima è stato commesso. Un errore nella strategia bottom-up è inizialmente locale al modulo in cui si verifica, poi il suo effetto si propaga mano a mano che il modulo viene composto con altri moduli.

Possiamo senz'altro affermare che la strategia top-down è in ogni caso da privilegiare rispetto a quella bottom-up. Quando non si riesca a trovare raffinamenti significativi del problema, si può procedere ad attività bottom-up, che permettano temporaneamente di risolvere sotto problemi parziali. Anche in questo caso, però, prima di procedere è opportuno produrre un nuovo progetto, composto di sole attività top-down.

4. CRITERI DI MODULARIZZAZIONE

In questo paragrafo esaminiamo alcuni criteri per verificare il grado di modularità di un programma. Affrontiamo inizialmente il criterio che nel par. 2 abbiamo chiamato coesione. Parleremo successivamente dell'accoppiamento tra moduli.

La *coesione* è una misura del tipo di relazione che sussiste tra elementi, o funzioni, svolte nel modulo. Più forte è questa relazione tra un insieme di funzioni, più è opportuno vederle organizzate come un singolo modulo, e perciò realizzarle con una specifica procedura o funzione. La coesione può essere usata sia per valutare a posteriori la bontà di una particolare struttura scelta per il programma sia come criterio di progetto. È possibile definire una scala di qualità, puramente indicativa tra diversi tipi di coesione. La coesione può essere:

1. *logica*, quando al modulo vengono demandate tutte le operazioni di uno stesso tipo, o con uno stesso scopo, una delle quali viene esplicitamente selezionata dal modulo chiamante. Ad esempio, è tipico nei programmi di una certa dimensione concentrare in un'unica procedura tutte le stampe di messaggi di controllo o di errore;
2. *procedurale*, quando i moduli corrispondono alle funzioni elaborate dalle diverse strutture di controllo: cioè, il criterio per decidere quali sono i moduli è stabilito dalla struttura gerarchica del controllo. In fig. 6 mostriamo un esempio di scelta dei moduli tramite il criterio della coesione procedurale;
3. *di comunicazione*, quando il modulo esegue in ordine sequenziale diverse funzioni che fanno riferimento ad uno stesso insieme di dati: ad esempio, i dati calcolati dalla prima funzione sono i dati di ingresso alla seconda, e così

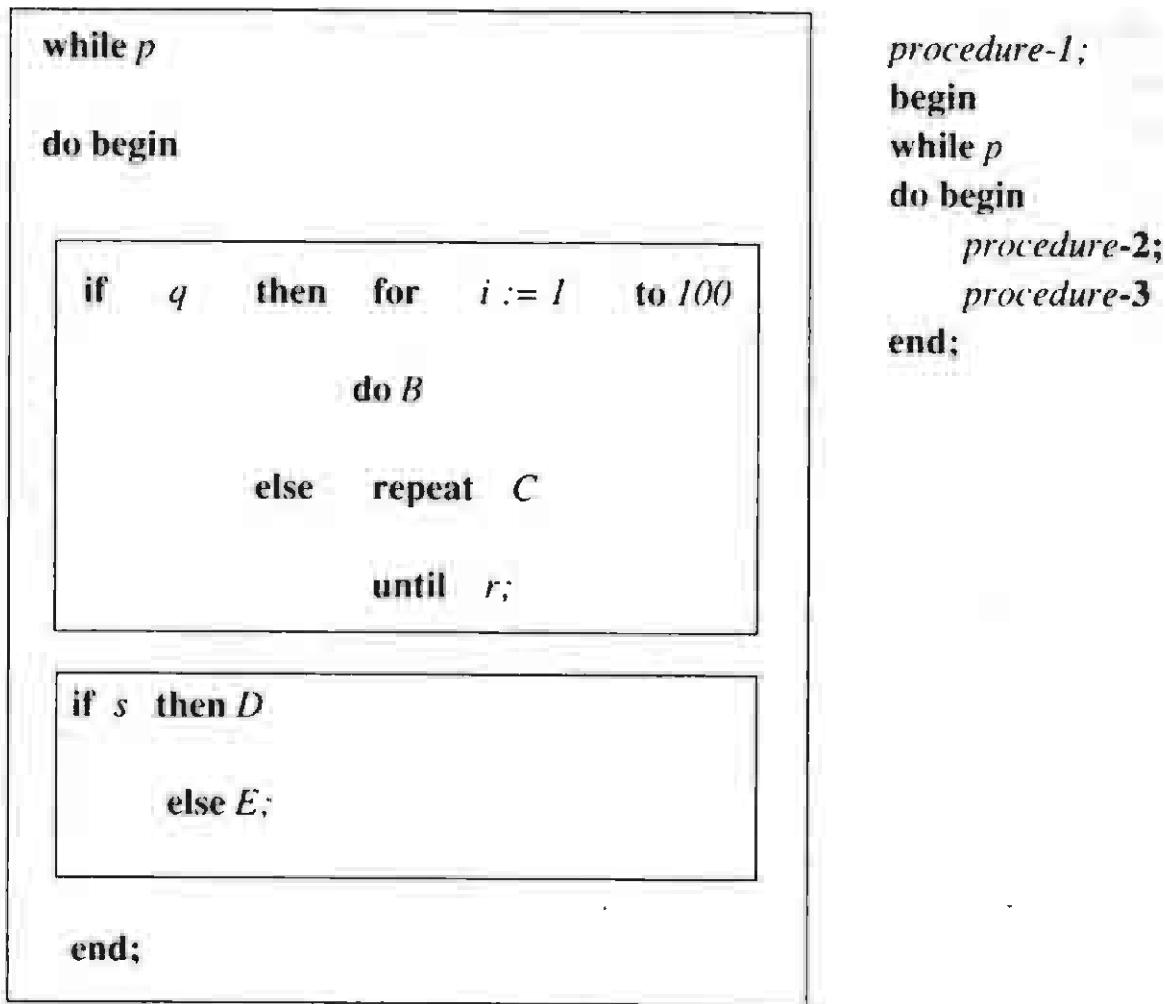


Fig. 6 - Esempio di coesione procedurale

via. Ad esempio, se nell'ambito di un problema dobbiamo acquisire dati da terminale, riprodurli sullo schermo e memorizzarli prima di procedere alla effettiva elaborazione, il modulo definito come "riproduci il dato sullo schermo e memorizzalo" ha una coesione di comunicazione tra le due funzionalità in esso definite. Questo tipo di coesione può considerarsi più forte delle precedenti perché le singole funzioni del modulo sono eseguite in sequenza temporale ed operano tutte sugli stessi dati:

4. funzionale, che può considerarsi il più alto grado di coesione, e si realizza quando tutte le istruzioni del modulo concorrono ad una stessa funzionalità facilmente identificabile e definibile mediante una espressione del linguaggio naturale sintetica ed espressiva. Ad esempio i moduli "ordina i numeri di matricola", "cerca la prenotazione nell'archivio", "calcola gli stipendi" sono esempi di moduli con elevata coesione funzionale.

Accanto ai precedenti tipi di coesione, vogliamo accennare alla coesione *informativa*, che caratterizza i moduli corrispondenti a tipi di dato astratti. In

questo caso le singole funzionalità corrispondono alle diverse operazioni definite per il tipo astratto, che operano in comune sugli stessi dati, quelli definiti dal tipo.

Se la coesione caratterizza la relazione tra funzioni all'interno di un modulo, l'accoppiamento è una misura delle relazioni che esistono tra moduli diversi. Queste relazioni sono determinate dalle modalità con cui vengono scambiate le informazioni tra moduli. Il più intuitivo tipo di accoppiamento tra moduli si ha quando i due moduli utilizzano uno stesso insieme di variabili globali. In questa situazione i moduli fanno riferimento agli stessi identificatori. Ogni ridenominazione o ristrutturazione delle variabili globali provoca in generale una ristrutturazione di tutti i moduli che fanno riferimento a tali variabili. Inoltre, l'uso di variabili globali rende complessa la riutilizzazione di un modulo nello stesso programma o in programmi diversi. Con questo tipo di accoppiamento si riduce inoltre la leggibilità dei programmi, perché occorre fare riferimento all'intera gerarchia delle procedure per comprendere il significato delle variabili utilizzate in un modulo. Infine, sono possibili accessi indesiderati a variabili globali da parte di moduli che in base alle specifiche non dovrebbero operare su tali variabili.

Un modo per evitare questo tipo di accoppiamento, detto *accoppiamento per dati comuni*, è quello di passare esplicitamente per mezzo di parametri tutti i dati comunicati tra moduli. Questo è il criterio che si dovrebbe adottare in un programma modulare. Nella sua applicazione occorre notare che nella scelta dei parametri i moduli si devono passare solo le informazioni strettamente indispensabili. Ad esempio, consideriamo un modulo il cui compito è di controllare, nell'ambito di un record che contiene varie informazioni relative agli impiegati di una azienda, la correttezza dei campi *matricola* ed *età*. In questo caso il modulo chiamante può passare l'intero record, ovvero i soli due campi; in questo ultimo caso, l'accoppiamento tra i moduli è chiaramente più ridotto.

Si noti che l'applicazione rigida di questo criterio può ridurre la leggibilità del programma. Supponiamo ad esempio che in un programma il modulo principale richiami un numero elevato di moduli, e passi a ciascuno, accanto a dati specifici, un insieme comune di dati. In questo caso, può convenire mantenere globali le variabili comuni a tutti i moduli, in modo da ridurre i parametri passati, e aumentare perciò la leggibilità complessiva del programma.

Accenniamo ad un secondo tipo di accoppiamento, detto di controllo, che si ha quando tra i dati passati da un modulo ad un altro vi sono dati di controllo,

cioè dati che non vengono modificati dal modulo subordinato, ma determinano l'ordine di esecuzione delle istruzioni.

Esempio. Sia dato un programma che legge un insieme di record di vario tipo (ad esempio record che descrivono informazioni anagrafiche, fiscali, ecc.), effettua su di essi dei controlli di integrità dipendenti dal tipo (ad esempio, per record anagrafici verifica che il cognome sia formato da tutte lettere, l'età sia un numero compreso tra 0 e 120, ecc.), e, in caso di presenza di errori, effettua una stampa.

Il precedente programma può essere strutturato in termini di un insieme di procedure, mostrato in fig. 7.

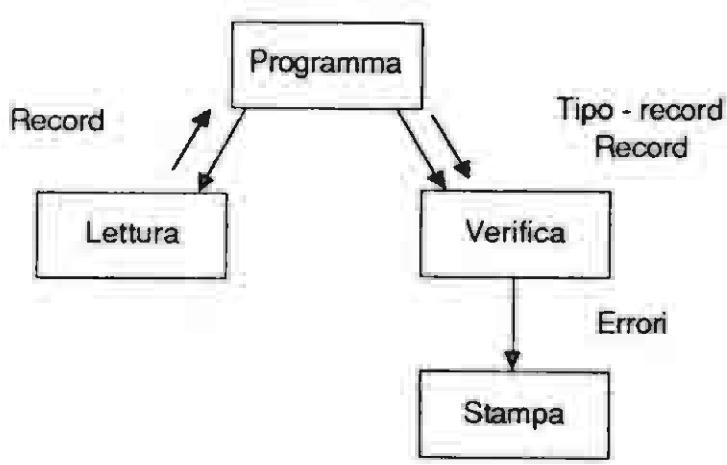


Fig. 7 - Moduli con accoppiamento di controllo

La figura mostra la struttura gerarchica dei moduli ed i dati scambiati; tale struttura verrà anche chiamata nel seguito *albero dei moduli* o *albero delle procedure*. In questo caso c'è un accoppiamento di controllo tra il programma principale ed il modulo *Verifica*, dovuto al passaggio del parametro *Tipo-record*, che determina il tipo di operazioni effettuate nel modulo *Verifica*. Il disaccoppiamento può avvenire (vedi fig. 8) spostando nella procedura *Verifica* il blocco di istruzioni che individua il tipo di controllo, e delegando al *Programma* la eventuale chiamata della procedura *Stampa*.

(5. LA PROGRAMMAZIONE STRUTTURATA

Nei precedenti paragrafi abbiamo esaminato tecniche generali per organizzare il progetto di un programma complesso. Tali tecniche sono indipendenti dal linguaggio programmatico scelto. Il loro utilizzo porta alla costruzione di

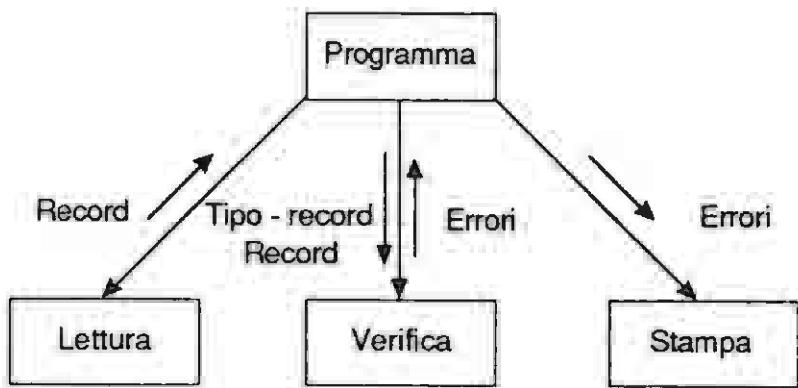


Fig. 8 - Possibile disaccoppiamento tra i moduli

un algoritmo in cui sono state effettuate le scelte fondamentali relativamente alla struttura delle procedure, dei tipi di dati, e delle principali strutture di controllo. Nella fase finale del progetto il prodotto della fase precedente viene codificato nel linguaggio programmatico. Questa attività dipende in generale dal linguaggio scelto; essa è tanto più facilitata quanto più il linguaggio programmatico presenta caratteristiche che rendano naturale la traduzione delle scelte fatte in termini di strutture del linguaggio. Se, ad esempio, si è scelto un algoritmo ricorsivo, sarà immediata la sua traduzione in un linguaggio che permetta la chiamata ricorsiva di procedure e funzioni, mentre dovremo procedere ad un passo di traduzione se il linguaggio scelto non la permette.

In questo paragrafo approfondiamo le modalità con cui viene tradotta la struttura del controllo in termini delle istruzioni del linguaggio, con riferimento al Pascal. Introdurremo in particolare la disciplina di programmazione che prende il nome di programmazione strutturata; essa ha lo scopo di semplificare la struttura del programma, disciplinando le modalità di composizione delle istruzioni e delle procedure, in modo da ottenere programmi la cui struttura, dal punto di vista del controllo, sia facilmente individuabile e comprensibile.

I fondamentali meccanismi di strutturazione del controllo utilizzati negli algoritmi, e che abbiamo già considerato come meccanismi di decomposizione nel par. 2 possono ricondursi a tre:

1. la struttura di sequenza, che permette di comporre istruzioni ed eseguirle una di seguito all'altra;
2. la struttura condizionale, che permette di eseguire una tra due (o più) istruzioni a seconda del valore di una espressione booleana;
3. la struttura di iterazione, che permette di eseguire ripetutamente una istruzione, sotto il controllo di una espressione booleana.

```
loop  
<istruzione1>  
if <condizione> then exit  
<istruzione2>  
endloop
```

Fig. 9 - La struttura *loop exit endloop*

Al di là della naturalezza di tali strutture di controllo, un aspetto che le rende particolarmente efficaci per generare programmi fortemente strutturati, è quello di avere soltanto un punto di ingresso (alla struttura), ed un punto di uscita. I raffinamenti mostrati nella fig. 3 portano in effetti ad utilizzare solo istruzioni ad un ingresso ed una uscita. Questa proprietà le rende semplici da comprendere, ed è inoltre rispettata da tutte le strutture di controllo che si possono costruire per composizione da esse.

Il linguaggio Pascal mette a disposizione un ricco insieme di istruzioni che hanno un solo ingresso ed una sola uscita: la istruzione composta, **if then else**, **case**, **while**, **for**, **repeat**. Accanto alle strutture di controllo del linguaggio Pascal, altre ne sono state proposte che permettono di eseguire funzionalità più complesse, continuando a mantenere la semplice struttura ad un ingresso ed una uscita. Una di queste è la *loop ... exit ... endloop* (vedi fig. 9). La sua esecuzione provoca iterativamente la esecuzione della *<istruzione1>*, il calcolo della *<condizione>*, e l'esecuzione della *<istruzione2>*, fin quando *<condizione>* non risulta vera: a questo punto il ciclo viene abbandonato, e si passa ad eseguire la istruzione successiva. Dunque, la istruzione permette una uscita alla fine del ciclo da un suo punto intermedio.

Un altro meccanismo utilizzato nei linguaggi per modificare il flusso del controllo è costituito dalle istruzioni di salto incondizionato, come la istruzione **goto**. Con la istruzione **goto** il controllo può saltare da un punto ad un altro del programma in modo completamente arbitrario. L'uso della istruzione **goto** riduce notevolmente la leggibilità e la modularità dei programmi per le seguenti ragioni:

1. fa perdere la organizzazione gerarchica delle strutture di controllo che hanno i programmi strutturati;

2. non esprime al suo interno il motivo che determina la alterazione del normale flusso del controllo. Fa perdere perciò leggibilità al programma, perché la comprensione del suo significato richiede una analisi del contesto in cui la istruzione è eseguita.

Possiamo a questo punto essere più precisi nella descrizione dei potenziali vantaggi della programmazione strutturata:

1. dà luogo ad algoritmi più leggibili, in quanto è più facile, in programmi strutturati, individuare i moduli corrispondenti alle varie funzionalità di cui si compone l'algoritmo;
2. rende possibile un progetto per raffinamenti successivi di tipo top-down, in cui ogni nuovo raffinamento è completamente indipendente dai precedenti e dai successivi, semplificando in tal modo le scelte di progetto; una ulteriore semplificazione è dovuta al fatto che i raffinamenti ammessi sono in numero limitato (sono quelli che corrispondono alle strutture di controllo ammesse);
3. come conseguenza dei due punti precedenti, semplifica la individuazione degli errori e la loro correzione, poiché, essendo più semplice individuare nel programma le varie funzionalità eseguite, è anche più facile scoprire quale (o quali) tra esse siano state progettate scorrettamente, e risalire così alla causa dell'errore;
4. semplifica la manutenzione del programma, per motivazioni analoghe a quelle espresse nel punto 3.

Talvolta, disciplinarsi a non usare la istruzione **goto** può richiedere la introduzione nel programma di ulteriori strutture programmatiche. Si veda a tale proposito il seguente esempio.

Esempio. Consideriamo l'algoritmo di fig. 10.a espresso nel linguaggio dei grafi di flusso. Questo algoritmo, che è facilmente esprimibile con la istruzione di *loop* (vedi fig. 10.b) non può essere direttamente espresso per mezzo delle istruzioni iterative nel linguaggio Pascal (**while**, **for**, **repeat**): questo perché l'uscita dal ciclo avviene al verificarsi di una condizione che non si trova né all'inizio né alla fine del ciclo, ma in un punto intermedio. Per realizzare l'algoritmo con le strutture di controllo note dobbiamo trasformarlo, introducendo una nuova variabile booleana, il cui scopo è di rappresentare la condizione di uscita (vedi il programma Pascal in fig. 10.b). Quando l'insieme delle variabili booleane è limitato, come peraltro avviene nella grande maggioranza dei casi, il loro uso costituisce un semplice mezzo per evitare i salti incondizionati.

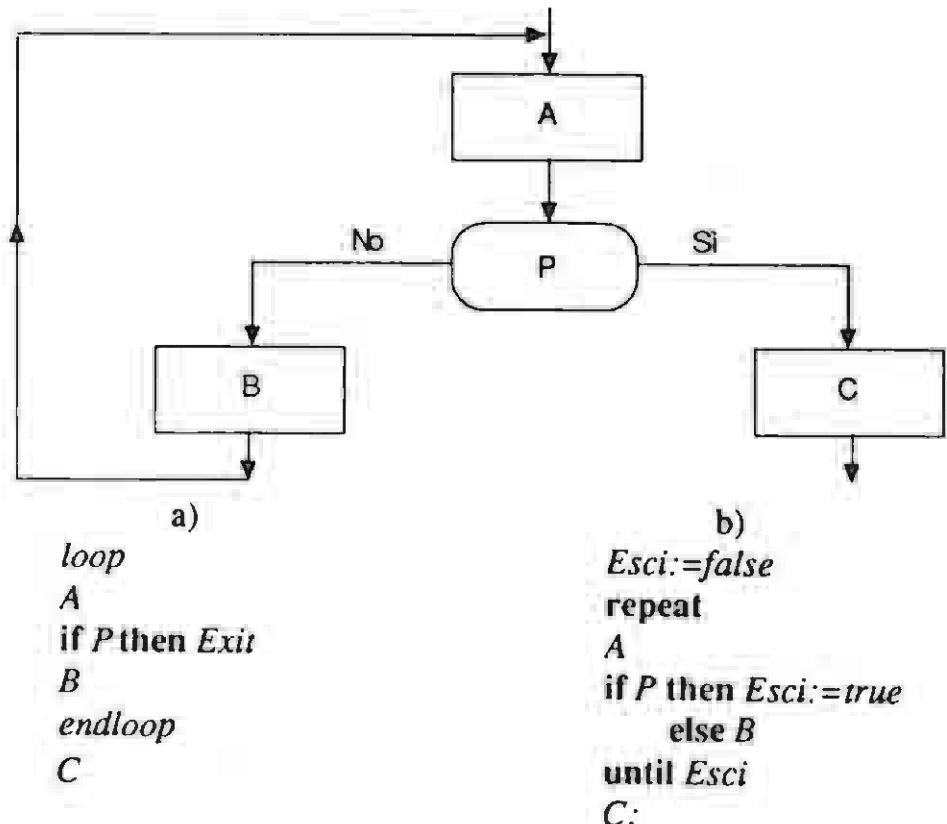


Fig. 10 - Uso di una variabile di terminazione ciclo per realizzare un programma strutturato

Esempio. Come ulteriore esempio, vediamo in fig. 11 due realizzazioni dell'algoritmo di ricerca di un valore in un vettore: la prima utilizza una istruzione **while**, la seconda una istruzione **for** e una istruzione **go to**.

Prima di concludere, è rilevante da un punto di vista teorico chiedersi se l'utilizzo di sole strutture di controllo ad un ingresso ed una uscita non limiti il potere espressivo di un linguaggio. Per esprimere in modo più formale il precedente problema è necessario definire *l'equivalenza di programmi*. Diremo che due programmi sono *fortemente equivalenti* se, sottoposti agli stessi dati di ingresso, o non terminano o terminano entrambi producendo gli stessi dati di uscita. Diremo anche che *un insieme di strutture di controllo è completo* in un linguaggio se per ogni programma nel linguaggio ne esiste uno *fortemente equivalente* che usa solo quelle strutture di controllo.

Il teorema di Böhm e Jacopini, concepito inizialmente per il linguaggio dei grafi di flusso, può essere ridefinito nel linguaggio Pascal come segue:

Le strutture di composizione, if-then-else e while costituiscono un insieme completo.

Come corollario del teorema, sono anche insiemi completi quelli in cui si

Cerca un valore in un vettore e stampa un messaggio

con la struttura while

```
i := 1;  
while i <= n and v[i] <> valore do i := i+1;  
if v[i] = valore  
then writeln [i]  
else writeln ('non trovato')
```

con la istruzione go to

```
for i := 1 to n do  
  if v[i] = valore  
  then begin  
    writeln ('trovato');  
    go to l  
  end;  
writeln ('non trovato');  
l:
```

Fig. 11 - Lo stesso algoritmo espresso con varie strutture di controllo

sostituisca alla istruzione **while** la **repeat** ed alla istruzione **if** la **case**. Infatti l'istruzione **case** può essere espressa per mezzo di un insieme di istruzioni **if** in cascata, mentre la istruzione **while** può essere espressa per mezzo della istruzione **repeat** con la trasformazione di fig. 12 (viene anche riportata la trasformazione dalla **repeat** alla **while**).

Il teorema, importante dal punto di vista teorico, non fornisce peraltro una indicazione per affrontare l'aspetto trattato in precedenza, e cioè come generare direttamente programmi strutturati.

6. LA DOCUMENTAZIONE

In questo paragrafo ci soffermiamo sulla qualità che abbiamo chiamato leggibilità, e sulle tecniche a disposizione del progettista per ottenerla. Comprendere facilmente i programmi è molto utile, per diverse ragioni:

1. permette di proseguire più facilmente il progetto ogni volta che lo si sia

repeat		A;
<i>A</i>	può realizzarsi con	while not <i>p</i>
until <i>p</i> :		do <i>A</i> ;
while <i>p</i>		if <i>p</i>
do <i>A</i> :	può realizzarsi con	then repeat
		<i>A</i>
		until not <i>p</i> ;

Fig. 12 - Simulazione di una repeat con una while e viceversa

- interrotto (e questo accade molto spesso, perché la progettazione di un programma di medie dimensioni può richiedere settimane e mesi);
2. permette di comunicare ad altri in maniera chiara le scelte adottate nel programma;
 3. permette di capire in maniera più semplice quale problema il programma risolva e quindi capire più facilmente se il programma sia o meno corretto, e dove si trovino gli errori;
 4. permette di modificare più facilmente il programma, ogni volta che nel tempo cambino le specifiche del problema.

Le tecniche a disposizione del progettista per produrre programmi leggibili sono molto varie; introdurremo nel seguito le principali.

1. Scegliere algoritmi autodocumentanti

Un problema ha in generale molti algoritmi risolutivi; se ne abbiamo individuati diversi, conviene scegliere quello (o quelli) la cui strategia risolutiva esprime nella maniera più semplice le richieste del problema.

Esempio. Si voglia costruire una matrice quadrata come quella di fig. 13. La matrice ha tutti gli elementi uguali a zero, eccetto che sulla diagonale principale, dove hanno valore uno; essa è chiamata diagonale.

La fig. 14 mostra quattro programmi ognuno dei quali risolve il problema posto.

Le strategie seguite dai programmi sono molto diverse; si noti, tuttavia, che, mentre per i primi tre una breve analisi del programma permette di ricostruire

1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1

Fig. 13 - Matrice diagonale

la funzione calcolata, il programma 4 è meno immediatamente comprensibile, poiché sfrutta una proprietà delle operazioni su interi per discriminare tra gli elementi della diagonale principale ed elementi esterni alla diagonale; in questo caso si richiede a chi legge il programma una conoscenza specifica di questa proprietà, il che ne rende più complicata la comprensione. Per quanto riguarda i primi tre algoritmi, è difficile dire quale sia il più naturale: probabilmente la strategia più facilmente comprensibile è quella dell'algoritmo 1 che possiamo riassumere in linguaggio naturale come segue:

Per ogni elemento della matrice
se è sulla diagonale,
allora assegnagli valore 1
altrimenti assegnagli valore 0

2. Evidenziare la struttura del programma

Chiarire come il programma sia organizzato è fondamentale per comprenderne il significato. Nei paragrafi precedenti abbiamo mostrato come la modularità e l'uso di opportune strutture di controllo sono primi meccanismi in tal senso. Altre tecniche sono le seguenti.

a. *Indentare il programma.* Indentare il programma significa mettere in evidenza nel testo, mediante opportuni incolonnamenti, i gruppi di istruzioni che svolgono una funzione comune, in modo da evidenziare tale funzione rispetto al resto del programma.

Programma 1

```
for i := 1 to n do
    for j := 1 to n do
        if i = j then mat [i,j] := 1
        else mat [i,j] := 0
```

Programma 2

```
for i := 1 to n do
    for j := 1 to n do
        begin
            mat [i,j] := 0;
            if i = j then mat [i,j] := 1
        end
```

Programma 3

```
for i := 1 to n do
    for j := 1 to n do
        mat [i,j] := 0;
for i := 1 to n do
    mat [i,i] := 1
```

Programma 4

```
for i := 1 to n do
    for j := 1 to n do
        mat [i,j] := (i div j) * (j div i)
```

Fig. 14 - Esempi di programmi equivalenti

Esempio. La fig. 15 mostra un programma non indentato e lo stesso programma con l'indentazione. Lo scopo del programma (già visto nella introduzione) è leggere tre valori che rappresentano le lunghezze dei lati di un triangolo e determinare il corrispondente tipo di triangolo. La indentazione permette di individuare le singole funzioni svolte.

Programma non indentato

```
if a = b then
  .
  if b = c
    then writeln ('equilatero')
  else writeln ('isoscele')
  else
    if b = c
      then writeln ('isoscele')
    else
      if a = c
        then writeln ('isocele')
      else writeln ('scaleno')
```

Programma indentato

```
if a = b
  then
    if b = c
      then writeln ('equilatero')
    else writeln ('isoscele')
  else
    if b = c
      then writeln ('isoscele')
    else
      if a = c
        then writeln ('isocele')
      else writeln ('scaleno')
```

Fig. 15 - Programma non indentato e corrispondente programma indentato

b. Documentare per livelli di astrazione. Se il progetto è stato effettuato in modo top-down, l' insieme delle successive versioni del programma (anche chiamate *piani di raffinamento*) è un ottimo strumento di documentazione. Accanto ad esso possiamo produrre l'albero delle procedure, che esprime in maniera grafica i legami gerarchici tra i moduli del programma. Qualora il programma sia stato ottenuto mediante un utilizzo misto di raffinamenti top-

down e di attività bottom-up, conviene in genere produrre un nuovo progetto, questa volta tutto composto di raffinamenti top-down.

c. *Commentare il programma*. Le frasi in linguaggio naturale prodotte nel corso del progetto per descrivere l'algoritmo a vari livelli di raffinamento possono comparire nella versione definitiva sotto forma di commenti. Anche se l'algoritmo è stato progettato utilizzando direttamente il linguaggio programmatico a disposizione, è conveniente in ogni caso inserire nel programma commenti. In particolare, è utile:

1. usare un commento all'inizio del programma, che descriva il problema risolto;
2. usare commenti nella sezione di dichiarazione delle variabili, che spieghino per ogni variabile il ruolo che essa svolge nel programma: essi sono detti *commenti sui dati*;
3. usare un commento *prima* di ogni frammento di programma che realizza una funzionalità significativa. Questi commenti hanno lo scopo di spiegare *cosa* il frammento di programma calcola, indipendentemente da *come* lo calcola: essi sono talvolta chiamati *commenti motivazione*;
4. usare commenti *dopo* ogni frammento di programma che realizza una funzionalità significativa. Questi commenti, talvolta chiamati *commenti asserzione*, hanno lo scopo di chiarire le relazioni logiche che sussistono tra le variabili in quel punto del programma. Sono utili perché la derivazione delle relazioni tra le variabili obbliga ad una comprensione profonda del significato del programma; essi possono anche essere utilizzati per provare qualitativamente la correttezza del programma.

Mostriamo in fig. 16 un semplice programma in cui sono stati utilizzati tutti i precedenti tipi di commenti (con le dizioni *CD*, *CM* e *CA* intendiamo rispettivamente commenti sui dati, commenti motivazione e commenti asserzione).

d. *Usare nomi di variabili autoesplicativi*. Per spiegare il ruolo delle variabili, oltre che usare commenti, possiamo scegliere nomi mnemonici. Ad esempio, ad una variabile che denota il massimo di un insieme di valori conviene dare un nome come *max*, o *massimo*, piuttosto che *xyx*.

A conclusione di questo paragrafo mostriamo con un esempio come si possa organizzare la documentazione di un programma a partire da un programma totalmente non documentato. Questo è un caso limite, perché in generale la documentazione va prodotta nel corso stesso del progetto.

```

program cerca_massimo;
{ Lo scopo del programma è cercare il massimo tra un insieme di numeri }

var
massimo { CD: esprime durante il programma il massimo provvisorio, ed
al termine, il massimo dell'insieme }: integer;
n { CD: numero dei valori da esaminare }: integer;
numero { CD: generico numero nell'insieme }: integer;
i { CD: variabile di controllo del ciclo }: integer;

begin
{ CM: legge il numero dei valori da leggere, ed il primo di tali valori, che
è il massimo provvisorio }
readln (n);
readln (numero);
massimo := numero;
{ CM: legge i valori nell'insieme, e seleziona il massimo }
for i := 1 to n - 1 do
begin
readln (numero);
{ CM: ogni volta che trova un valore maggiore del massimo provvisorio, lo aggiorna }
if numero > massimo
then massimo := numero
{ CA: massimo rappresenta il massimo tra i numeri letti fino a questo
punto }
end;
{ CA: massimo rappresenta il massimo nell'insieme }
writeln (massimo)
end.

```

Fig. 16 - Esempio di programma commentato

Esempio. Si veda il programma di fig. 17. Il programma ha lo scopo di leggere una sequenza di 50 numeri che possiamo interpretare come un insieme di età e deve stampare tutte le età non incluse nell'insieme che siano comprese tra la minima e la massima. Il programma documentato è mostrato nelle tre figure seguenti. La documentazione è composta da:

```
program poco_documentato;
var m1, m2, m11, m21, i, e: integer; k: array [1 .. 100] of
boolean; begin for i := 1 to 100 do k [i] := false; for i := 1 to
50 do begin readln (e); k [e]:= true end; i := 1; while i <= 100
and k [e] = false do i := i + 1; m1 := i; i := 100; while i >= 1
and k [e] = false do i := i - 1; m2 := i; m11 := m1 + 1; m21 := 
m2 - 1; for i := m11 to m21 do if k [i] = false then writeln (i)
end.
```

Fig. 17 - Programma mal documentato

```
program cerca_eta;
```

{ Il programma legge un insieme che rappresenta le età di un gruppo di persone, che si assume siano comprese tra 1 e 100, e stampa tutte le età comprese tra la minima e la massima non presenti nell'insieme. Se ad esempio l'insieme fosse composto delle età 5, 7, 2, si dovrebbe stampare 3, 4, 6 }

begin

1. Leggi l'insieme delle età
2. Trova le età minime e massime nell'insieme
3. Trova le età tra la minima e la massima non comprese nell'insieme

end.

Fig. 18 - Prima versione top-down del problema

1. uno sviluppo top down del programma (fig. 18);
2. il programma (fig. 19);
3. l'albero delle procedure (fig. 20).

Si noti che si sono anche effettuate alcune piccole ristrutturazioni, semplificando alcune condizioni booleane ed eliminando le due variabili M11 e M21, il cui unico scopo era di rappresentare i limiti inferiore e superiore della ricerca dei valori da stampare, valori che possono essere rappresentati anche per mezzo delle espressioni *minimo + 1* e *massimo - 1*.

```

program cerca_eta;
{ Il programma legge un insieme che rappresenta le età di un gruppo di persone, che si assume siano comprese tra 1 e 100, e stampa tutte le età comprese tra la minima e la massima non presenti nel vettore. Se ad esempio il vettore fosse composto delle età 5, 7, 2, si dovrebbe stampare 3,4,6 }

type array_eta: array [1 .. 100] of boolean;
var minimo { minima età nel vettore },
    massimo { massima età nel vettore },
    i { indice di ciclo }: integer;
    presenza_eta { vettore che rappresenta le età; nel vettore l'età è assunta come indice; il generico elemento ha valore true se la corrispondente età è compresa nel vettore, false altrimenti }: array_eta;

procedure leggi_eta {var presenza_eta: array_eta};
var eta: integer;
{ prepara il vettore presenza_eta }
begin
for i := 1 to 100 do presenza_eta [i] := false;
for i := 1 to 50
do begin
    readln (eta);
    presenza_eta [eta] := true
    end;
end;

procedure trova_min_max_eta (var presenza_eta: array_eta;
                                var minimo, massimo: integer );
var i { indice di ciclo }: integer;
begin
{cerca il minimo, scandendo il vettore a partire dal primo elemento }
    i := 1;
    while i <= 100 and (not presenza_eta [i] )
    do i := i + 1;
    minimo := i;
{cerca il massimo, scandendo il vettore a partire dall'ultimo elemento }
    i := 100;

```

```

while i >= 1 and (not presenza_eta [i] )
  do i := i - 1;
  massimo := i
end;
procedure trova_eta (var presenza_eta: array_eta;
                      minimo, massimo: integer);
{ scandisce il vettore presenza_eta e seleziona i valori compresi tra il
minimo ed il massimo che non sono comprese nel vettore presenza_eta }
var i { indice di ciclo }: integer;
begin
  for i := minimo + 1 to massimo - 1 do
    if not presenza_eta [i] then writeln (i)
end;
{ corpo del programma }
begin
  { leggi le età }
  leggi_eta (presenza_eta);
  { trova la età minima e massima }
  trova_min_max_eta (presenza_eta , minimo, massimo);
  { a questo punto minimo e massimo rappresentano la minima e
massima età }
  { trova le età desiderate }
  trova_eta (presenza_eta, minimo, massimo)
end.

```

Fig. 19 - Programma suddiviso in moduli e commentato

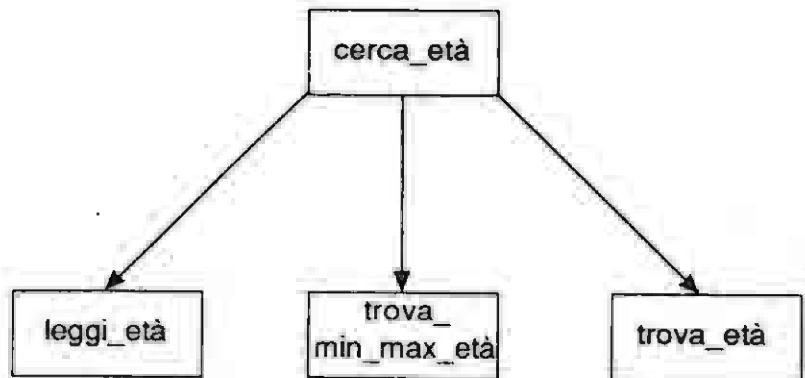


Fig. 20 - Albero delle procedure

7. UNA METODOLOGIA DI PROGETTO

A conclusione di questo capitolo, riassumiamo le considerazioni effettuate mostrando sinteticamente la struttura di una metodologia di progetto (vedi fig. 21).

Come si vede, la metodologia si compone di quattro fasi: nel seguito le analizzeremo separatamente.

7.1. Analisi dei requisiti

La fase di analisi dei requisiti ha lo scopo di chiarire eventuali ambiguità ed incompletezze in essi presenti. È frequente che la versione iniziale dei requisiti sia imprecisa. È fondamentale che tali imprecisioni siano eliminate il più presto possibile, per evitare che esse vadano ad incidere sulle fasi successive, poiché, intuitivamente, il costo della correzione è sempre maggiore man mano che si avanza nel progetto.

Alcune regole semplici per scoprire le ambiguità di una specifica sono le seguenti:

- a. verificare se esse coprono tutte le possibili situazioni coinvolte dal problema;
- b. verificare che ogni aspetto delle specifiche possa avere una unica interpretazione nel contesto della applicazione che esse descrivono;
- c. verificare che non vi siano inconsistenze;
- d. fare alcuni esempi della trasformazione desiderata tra le informazioni in ingresso e le informazioni in uscita, così da sincerarsi su alcuni casi reali della adeguatezza delle specifiche.

Questo primo passo in generale darà luogo ad una nuova versione delle specifiche, su cui ci si baserà nei passi successivi.

7.2. Raffinamento iterativo dell'algoritmo

Il passo di raffinamento iterativo produce, a partire dal problema, una prima decomposizione dell'algoritmo, e genera successivi raffinamenti, sempre più vicini alla sua formulazione nel linguaggio di programmazione. I raffinamenti possono riguardare la struttura dell'algoritmo, ovvero i tipi di dati coinvolti. Consideriamo i due aspetti separatamente.

a. Raffinamenti sull'algoritmo. Come abbiamo visto, le strategie tipiche a disposizione del progettista sono la strategia top-down e la bottom-up; esse sono descritte nei passi 2.2.1 e 2.2.2 di fig. 21.

Nella fig. 3 abbiamo anche introdotto i principali tipi di decomposizioni che il progettista può adottare per raffinare la descrizione dell'algoritmo. Abbiamo visto che in tali decomposizioni, oltre ad utilizzare procedure o funzioni, possiamo utilizzare:

1. una frase in linguaggio naturale;
2. un insieme di istruzioni Pascal.

La scelta cadrà su una frase in linguaggio naturale ogni qualvolta si individua una funzionalità che non si è ancora in grado di esprimere per mezzo del linguaggio Pascal e che non si ritiene ancora di realizzare mediante attivazione di procedura o funzione. La seconda scelta corrisponde al caso in cui si è ormai individuata la sequenza di istruzioni che corrisponde al frammento di programma.

b. Raffinamenti sui tipi di dato. Per i tipi di dato sono adottati i metodi visti nel cap. 3, individuando inizialmente la struttura astratta che risponde in maniera più naturale alle caratteristiche del problema, e successivamente scegliendo tra le realizzazioni possibili quella più conveniente dal punto di vista della efficienza e della comprensibilità del programma che ne risulta. La conoscenza delle operazioni sui tipi previste dal problema e la loro frequenza fornisce elementi importanti per affrontare la scelta.

7.3. Analisi dell'algoritmo

In questa fase si verifica periodicamente se le scelte fatte per l'algoritmo e i tipi di dato sono rispondenti alle qualità desiderate. I criteri di verifica illustrati in fig. 21 riguardano:

- a la documentazione: tutte le frasi in linguaggio naturale prodotte durante i passi intermedi potranno essere utilizzate nelle versioni successive dell'algoritmo; così, se a un dato raffinamento una frase in linguaggio naturale viene sostituita da una procedura, la frase sarà trasformata in un commento da inserire prima della attivazione della procedura;
- b scelta dei frammenti di programma da realizzare per mezzo di procedure o funzioni; un criterio approssimato suggerisce di scegliere frammenti che corrispondono a funzionalità individuabili in modo evidente e descrivibili

Passo 1 { *Analisi dei requisiti* }

1. Accertati che le specifiche del problema siano chiare, non ambigue, complete.

Passo 2 { *Raffinamento iterativo dell' algoritmo* }

- 2.1 Scegli un primo raffinamento dell'algoritmo, utilizzando specifiche astratte per i tipi di dati.
- 2.2 Finché c'è qualche parte del problema non ancora espressa per mezzo del linguaggio programmatico:

Passo 2.2.1 { *Passo top-down* }

2.2.1 Raffina la descrizione dell'algoritmo e dei tipi, utilizzando:

- per l'algoritmo, le strutture di controllo del linguaggio ovvero attivazioni di procedure o funzioni (la scelta va fatta in base ai criteri di modularità espressi nel par. 3);
- per i tipi, rappresentazioni delle specifiche astratte per mezzo di nuovi tipi astratti o per mezzo dei tipi di dati del linguaggio.

Passo 2.2.2 { *Passo bottom-up* }

2.2.2 Se non riesci a procedere top-down

2.2.a Individua funzionalità elementari, realizzandole per mezzo di procedure o funzioni; rappresenta i tipi di dati per mezzo di specifiche astratte;

2.2.b Utilizza programmi scritti in precedenza, eventualmente adattandoli alle specifiche del problema.

Passo 2.3 { *Passo di integrazione* }

- 2.3 Integra i moduli individuati con il passo top-down ed il passo bottom-up, finché tutte le specifiche del problema non sono state espresse nell'algoritmo risolutivo.

Passo 3 { *Passo di analisi* }

3. Ad ogni nuova versione:

- Documenta l'algoritmo utilizzando come commenti le precedenti versioni in linguaggio naturale;
- Modularizza il programma, individuandone frammenti che è opportuno esprimere per mezzo di procedure;
- Verifica la correttezza della versione attuale dell'algoritmo, sia mediante una sua esecuzione simbolica che mediante test;
- Verifica l'efficienza dell'algoritmo.

Passo 4 { *Passo di ristrutturazione* }

4. Qualora le analisi effettuate nel passo 3 consiglino una modifica, procedi ad una ristrutturazione dell'algoritmo e della relativa documentazione.
-

Fig. 21 - Una metodologia di progetto di programmi

per mezzo di frasi sintetiche, che rispondano perciò al criterio della coesione funzionale. Nella scelta dei parametri si dovrà verificare anche il grado di accoppiamento;

- c. verifica di correttezza: si rimanda al cap. 5 per lo studio delle metodologie relative a questo aspetto;
- d. efficienza: essa potrà essere analizzata fin dai primi livelli di raffinamento, sia pure qualitativamente, così da essere in grado di comprendere fin dall'inizio se la strada intrapresa è conveniente ed accettabile, o non conviene pensare a qualche altra soluzione. Anche qui si rimanda al cap. 6 per lo studio delle metodologie.

7.4. Ristrutturazione

La attività di analisi può aver portato, specialmente per quanto riguarda le verifiche di correttezza ed efficienza, a decidere di ristrutturare l'algoritmo. Se dell'algoritmo esistono varie versioni, a diverso livello di raffinamento, verrà ogni volta risalire nei vari livelli fin quando non si individua la prima versione da modificare; a questo punto occorre modificare la versione, e poi procedere propagando la ristrutturazione ai livelli successivi; questo metodo, che è certamente più dispendioso, ha il grande vantaggio di mantenere tutta la documentazione coerente con le scelte finali di progetto.

5.

Analisi di programmi: la correttezza

- In questo capitolo si approfondiscono i concetti relativi alla prova (in inglese *test*) e alla correzione di un programma.

Come è stato osservato nel cap. 4, il costo di questa fase del progetto è molto alto: circa la metà del tempo totale richiesto per la scrittura di un programma è dovuto alla prova e alla correzione. Nonostante l'alto costo, i risultati che si ottengono non sono completamente soddisfacenti; infatti, nel caso di grossi programmi, la scoperta di nuovi errori è un processo quasi senza fine: alcuni errori vengono scoperti quando il programma è stato utilizzato con successo per mesi o, addirittura, per anni.

Nel primo paragrafo vengono date alcune definizioni introduttive, si analizzano i diversi tipi di errore e si esemplifica cosa si intende per programma corretto. Nei par. 2 e 3 vengono rispettivamente presentate le problematiche relative alla dimostrazione formale della correttezza e alla prova di un programma. Nel par. 4 si considerano, invece i problemi relativi alla prova di prodotti software complessi costituiti da più moduli. Infine, nel paragrafo finale, si esamina il problema della correzione di un programma.

I. CORRETTEZZA ED ERRORI DI UN PROGRAMMA

In questo paragrafo viene innanzitutto data una definizione di programma corretto; successivamente vengono presentati alcuni degli errori più comuni classificandoli sulla base delle loro cause.

1.1. Correttezza di un programma

Come abbiamo visto nel cap. 1 la semantica assiomatica di un programma S

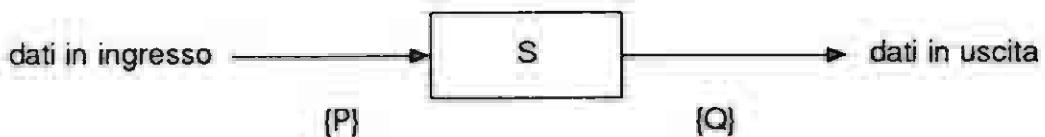


Fig. 1 - Rappresentazione grafica dell'esecuzione di un programma

è data da due predicati $\{P\}$ e $\{Q\}$ che rappresentano la precondizione e la postcondizione del programma. La fig. 1 visualizza la risoluzione di un generico problema A tramite un programma S: il programma riceve in ingresso un insieme di dati (l'*input*) e fornisce in uscita un insieme di valori (l'*output*). Affinché il programma S possa essere utilizzato correttamente i dati di ingresso devono soddisfare un insieme di vincoli. Il predicato P che è vero se e solo se l'*input* soddisfa questi vincoli rappresenta la precondizione che deve essere soddisfatta per poter utilizzare il programma S. Osserviamo inoltre che l'*output* deve soddisfare delle condizioni che dipendono dal problema A e dai particolari valori di ingresso. Il predicato Q che è vero se e solo se l'*output* soddisfa questi vincoli rappresenta la postcondizione che deve essere soddisfatta affinché il programma S risolva correttamente il problema A. Pertanto, se si vuole utilizzare il programma S per risolvere il problema A, è necessario che i dati di ingresso e di uscita soddisfino, rispettivamente, i predicati P e Q. L'espressione $\{P\} S \{Q\}$ rappresenta la semantica assiomatica o la *specifica* del programma S; essa è soddisfatta se vale la seguente proprietà: per ogni insieme di dati che soddisfano la precondizione P, se il programma S termina quando viene eseguito su tali dati, allora i dati di uscita prodotti dal programma soddisfano la postcondizione Q.

Esempio. Si deve calcolare il massimo comun divisore r di due numeri x e y interi e non negativi.

Il massimo comun divisore di due numeri non negativi è dato dal più grande numero intero che li divide entrambi. Pertanto il massimo comun divisore è definito se i due numeri x e y sono interi e non entrambi nulli. In questo caso il predicato P è

(x e y sono interi non negativi) and ((x > 0) or (y > 0))

Il massimo comun divisore r di x e y deve soddisfare il seguente predicato Q
 (r divide x) and (r divide y) and

(ogni numero intero che divide sia x che y divide anche r)

Siamo a questo punto in grado di dare la definizione di programma corretto.

Definizione. Un programma S è *parzialmente corretto* rispetto alla precondizione P e alla postcondizione Q, se la specifica $\{P\} S \{Q\}$ è soddisfatta. Un programma S è *corretto* rispetto alla precondizione P e alla postcondizione Q, se è parzialmente corretto e se esso termina ogni volta che viene eseguito su dati di ingresso che soddisfano la precondizione.

In base alla definizione precedente la dimostrazione della correttezza di un programma consiste dei seguenti passi:

1. si determina la precondizione P e la postcondizione Q;
2. si verifica se la specifica è soddisfatta;
3. si stabilisce se il programma termina per ogni possibile insieme di dati di ingresso che soddisfa la precondizione.

Nel par. 2 vedremo come sia possibile dimostrare in modo rigoroso la correttezza del programma. Purtroppo le prove di correttezza ottenute in questo modo risultano molto lunghe e possono facilmente contenere errori; per questa ragione le prove formali di correttezza sono effettuate raramente, soprattutto nel caso di programmi lunghi.

Nei paragrafi successivi considereremo un approccio più pragmatico al problema della correttezza. Nel seguito per indicare questo metodo di tipo sperimentale si userà anche l'espressione *test* di un programma. Il test di un programma consiste dei seguenti passi:

1. si determina un insieme di dati di input I;
2. si esegue il programma con i dati I;
3. si verificano i risultati ottenuti:
 - 3.1. se l'esecuzione del programma non termina in un lasso di tempo ragionevole o se i risultati ottenuti dalla esecuzione non sono quelli attesi, allora il programma non è corretto;
 - 3.2. se non sono stati rilevati errori, bisogna stabilire se è necessario effettuare altre prove. In questo caso si ritorna al passo 1, altrimenti il test termina.

Si noti che, generalmente, il test di un programma non permette di stabilirne la correttezza, a meno che non vengano provate tutte le possibili scelte di dati di ingresso. Infatti, se eseguiamo il programma con un sottoinsieme dei possibili input non possiamo escludere che esista un insieme di dati, che non è stato provato, e che evidenzia un errore.

Chiaramente, la prova di un programma con tutti i possibili dati di ingresso è impossibile nella maggioranza dei casi. Ad esempio, l'insieme dei dati di ingresso di un programma che calcola il prodotto di due interi compresi

tra -100000000 e +100000000 è composto da circa 40 milioni di miliardi di elementi. Se ciascuna prova richiede un microsecondo (0.000001 secondi) allora sono necessari 40 miliardi di secondi, cioè circa 40000 giorni.

Possiamo pertanto concludere che, anche se generalmente il test di un programma non permette di dimostrarne la correttezza, esso è lo strumento più usato nella pratica della programmazione. Infatti, se effettuiamo molte prove e non rileviamo errori, possiamo ragionevolmente supporre che la probabilità che il programma contenga errori sia molto piccola. Nei paragrafi successivi vedremo come scegliere l'insieme dei dati di test in modo tale da aumentare la probabilità che il programma sia corretto.

1.2. Classificazione degli errori

Per poter provare la correttezza di un programma, è importante anzitutto rendersi conto dei tipi di errore che possono essere commessi nel suo progetto. Esistono molti possibili errori che possono essere così classificati: errori nell'uso del linguaggio, logici e di codifica.

Gli errori nell'uso del linguaggio possono essere lessicali, sintattici e semantici. Gli errori lessicali si verificano quando si usano stringhe di caratteri non appartenenti al linguaggio di programmazione usato. Ad esempio, la stringa alfanumerica 1AX non è ammessa nel linguaggio Pascal perché inizia con un carattere numerico.

Gli errori sintattici sono dovuti ad un uso scorretto della sintassi del linguaggio. Se si utilizza il Pascal possibili errori sintattici sono quelli in cui non si pone un punto e virgola per separare due istruzioni, oppure si scrive

x := a := b

Gli errori sintattici sono individuati durante la compilazione del programma; generalmente viene anche stampato un messaggio che spiega il tipo di errore rilevato.

Gli errori semantici sono legati ad un uso scorretto delle regole che stabiliscono la semantica del linguaggio. Alcuni degli errori semantici sono individuati dal compilatore, e per questo sono detti di semantica statica, altri possono essere rilevati durante l'esecuzione del programma, e per questo sono detti di semantica dinamica.

Nel caso del linguaggio Pascal errori di semantica statica si verificano ogniqualvolta viene usata una variabile che non è stata dichiarata nella apposita

sezione di dichiarazione, oppure quando i parametri che compaiono in una istruzione di attivazione di procedura non corrispondono nel numero o nel tipo ai parametri specificati nella procedura stessa.

Gli errori che vengono rilevati a tempo di esecuzione dipendono dai valori che le variabili del programma assumono durante l'esecuzione.

Consideriamo, ad esempio, il seguente frammento di programma Pascal:

```
program ....  
....  
var v : array [1..10] of integer;  
    i : integer;  
....  
....  
readln (i);  
v[i] := 1;  
....
```

La correttezza della istruzione di assegnazione dipende dal particolare valore assegnato a *i*. Ad esempio, se a *i* viene assegnato il valore 11, allora si ha un errore semantico perché non esiste la componente di indice 11 dell'array *v*.

Un altro possibile errore di questo tipo si verifica quando si vuole accedere all'elemento puntato da una variabile puntatore, mentre questa è pari a *nil*.

Quando viene rilevato un errore semantico l'esecuzione del programma viene interrotta dopo che è stato stampato un messaggio diagnostico.

La tipologia degli errori che si commettono nell'uso del linguaggio dipende dal linguaggio utilizzato. Pertanto, per ridurre il tempo di correzione è auspicabile che il linguaggio sia definito in modo tale da aumentare il numero di errori rilevati durante la compilazione, riducendo in tal modo i tempi di correzione.

Illustriamo questo concetto facendo riferimento alla dichiarazione dei tipi di dato. Alcuni linguaggi, come ad esempio il Pascal, richiedono di dichiarare esplicitamente ogni variabile con il tipo di dato corrispondente. Per altri linguaggi questo non è obbligatorio. Ad esempio, nel linguaggio Fortran può mancare la dichiarazione di una variabile. In questo caso si utilizza un meccanismo di dichiarazione implicita che assegna alla variabile un tipo di dato a seconda del primo carattere dell'identificatore corrispondente.

Ad esempio, in mancanza di una esplicita dichiarazione, tutte le variabili il cui identificatore inizia con la lettera I sono considerate variabili intere. Questo

meccanismo può sembrare a prima vista più vantaggioso per il programmatore perché permette di risparmiare tempo nella scrittura del programma. Questo piccolo risparmio di tempo può essere però molto pericoloso.

Supponiamo, infatti, di avere un programma che utilizza, fra le altre, la variabile *IDFJ* a cui ad un certo punto del programma viene assegnato il valore 0. Nel linguaggio Fortran questo si scrive

$$IDFJ = 0$$

Supponiamo anche che, per un errore di battitura, accada di scrivere

$$IDGJ = 0$$

Poiché non tutte le variabili devono essere dichiarate esplicitamente, l'istruzione *IDGJ = 0* viene interpretata come l'assegnazione del valore zero alla variabile *IDGJ*. In questo caso la rilevazione dell'errore non è automatica durante la compilazione, ma è lasciata al programmatore.

È facile vedere che, se si utilizza un linguaggio di programmazione che richiede di dichiarare esplicitamente tutte le variabili, l'errore precedente viene immediatamente rilevato dal compilatore (a meno che non abbiamo dichiarato anche *IDGJ* come variabile intera!).

Per quanto riguarda gli errori logici essi sono, in genere, dovuti ad una scelta sbagliata dell'algoritmo di soluzione o delle strutture di dati. Gli errori di codifica, infine, sono molte volte dovuti a sviste o disattenzione e si verificano, ad esempio, quando si sbaglia l'inizializzazione di una variabile o la condizione di terminazione di un ciclo.

La frequenza di questi ultimi due tipi di errori può essere notevolmente ridotta utilizzando metodi sistematici di programmazione.

Esempio. Consideriamo il problema di stabilire il tipo di un triangolo, già visto nel capitolo introduttivo. Il programma di fig. 2 legge tre numeri che rappresentano le lunghezze dei lati di un triangolo e stampa il messaggio '*equilatero*', '*isoscele*', '*scaleno*', a seconda che i tre numeri rappresentino le lunghezze di un triangolo equilatero, isoscele o scaleno.

Il programma contiene due errori. Innanzitutto nella codifica della parte *then* della seconda istruzione *if* si incrementa erroneamente la variabile *uguali* di 2 (invece che di 1). Inoltre è facile vedere che nel caso che il triangolo sia scaleno vengono stampati ambedue i messaggi '*scaleno*' e '*equilatero*'. La correzione di questo errore richiede di eliminare la parte *else* dell'ultima istruzione *if* e di inserire alla fine la seguente istruzione *if*:

```
if uguali=3 then writeln ('equilatero')
```

In base alla classificazione precedente possiamo dire che il primo errore è un errore di codifica, mentre il secondo è un errore logico. Osserviamo infine che, se avessimo definito la variabile *uguali* come variabile di tipo booleano, allora avremmo avuto un errore di semantica (statica).

```
program triangolo_errato;
var primo, secondo, terzo, uguali: integer;
begin
  readln (primo, secondo, terzo);
  uguali := 0;
  if primo=secondo then uguali := uguali + 1;
  if primo=terzo then uguali := uguali + 2;
  if secondo=terzo then uguali := uguali + 1;
  if uguali = 0 then writeln ('scaleno');
  if uguali = 1 then writeln ('isoscele')
    else writeln ('equilatero')
end.
```

Fig. 2 - Programma errato per il problema del tipo di triangolo

2. PROVE FORMALI DI CORRETTEZZA

I metodi di prova formale della correttezza di un programma utilizzano una definizione formale della semantica del linguaggio.

Come abbiamo già accennato le prove formali di correttezza possono essere lunghe e complesse. Pertanto, in questo paragrafo ci limiteremo a esemplificare un metodo di prova che utilizza una semantica intuitiva del linguaggio. Nel seguito considereremo prima il caso di programmi che non utilizzano procedure e poi quello di programmi che ne fanno uso.

2.1. Correttezza di programmi senza procedure

Vediamo innanzitutto come l'utilizzo di strutture ad un solo ingresso e ad una sola uscita permetta di semplificare la verifica di correttezza di un

programma.

Supponiamo di avere un programma Pascal S composto di due o più istruzioni. Pertanto, la parte istruzioni di S può essere vista come formata da due istruzioni composte X e Y. In questo caso possiamo riscrivere la specifica del programma $\{P\} S \{Q\}$ nel seguente modo

$$\{P\} X ; Y \{Q\}$$

Con questa notazione si intende che il programma formato dalle due istruzioni X e Y da eseguire sequenzialmente ha P come precondizione e Q come postcondizione. Ora osserviamo che X e Y possono essere considerati a loro volta come due programmi per cui possiamo dare una specifica separata. In particolare, supponiamo di riuscire a individuare un predicato R tale che X e Y soddisfino le seguenti specifiche:

$$\{P\} X \{R\} \text{ e } \{R\} Y \{Q\}$$

In particolare $\{P\} X \{R\}$ specifica che quando i dati di input soddisfano la precondizione P allora i valori delle variabili dopo l'esecuzione del programma X soddisfano il predicato R. Analogamente, $\{R\} Y \{Q\}$ specifica che quando i dati di input di Y soddisfano il predicato R allora i dati di uscita del programma Y soddisfano la postcondizione Q.

Se si riesce a individuare R e a provare che ambedue le specifiche delle due istruzioni X e Y sono soddisfatte allora possiamo dedurre che anche la specifica del programma S è soddisfatta.

In modo analogo è possibile dimostrare che la prova di un programma di n istruzioni può essere ricondotta alla prova della correttezza di n programmi più semplici.

Illustriamo i concetti visti con due esempi.

2.1.1. Calcolo del modulo di due numeri interi

Supponiamo di scrivere un programma che legge da ingresso due numeri interi non negativi x e y e calcola r pari al valore di $x \bmod y$. Per non rendere il problema banale si supponga che le sole operazioni aritmetiche possibili siano l'addizione, la sottrazione e la moltiplicazione.

Affinché l'operazione modulo sia definita è necessario che y sia diverso da zero. Pertanto la precondizione che i dati di input devono soddisfare è la seguente:

$$P) \quad x \geq 0 \quad \text{and} \quad y > 0$$

Affinché il valore finale di r sia corretto esso deve risultare pari al resto della divisione di x per y . Formalmente questo equivale a dire che esiste un intero q che rende vero il seguente predicato Z che rappresenta la postcondizione:

$$Z) \quad x = q * y + r \quad \text{and} \quad 0 \leq r < y$$

Nella formula precedente e nel seguito del capitolo il simbolo $*$ denota l'operatore di moltiplicazione.

Consideriamo ora il programma della fig. 3.a.

Per dimostrare che la specifica del programma è soddisfatta eliminiamo le istruzioni di lettura e scrittura e consideriamo le due istruzioni iniziali come un'unica istruzione composta. Pertanto otteniamo il frammento di programma di fig. 3.b composto da due istruzioni.

Nel seguito occorre distinguere fra identificatore e valore di una variabile. Questa distinzione può essere fatta in modo rigoroso utilizzando una semantica formale del linguaggio. La notazione conseguente diverrebbe molto pesante. Pertanto, ancora una volta, preferiamo utilizzare una notazione più intuitiva. In particolare nel seguito assumiamo che identificatori di variabili sono denotati da lettere minuscole, mentre i corrispondenti valori sono denotati da lettere maiuscole. Ad esempio, r denota una variabile, mentre R rappresenta il valore (intero) della variabile r .

Sia W il seguente predicato

$$W) \quad (X = Q * Y + R) \quad \text{and} \quad (0 \leq R < Y)$$

Per dimostrare che la specifica del programma è soddisfatta è sufficiente far vedere che:

1. se i valori X e Y dei dati di ingresso soddisfano il predicato P allora, dopo aver eseguito l'istruzione composta, i valori delle variabili del programma soddisfano il predicato W ;

while venga eseguita, allora le variabili del programma soddisfano il predicato Z quando l'esecuzione dell'istruzione è terminata.

Si noti che la dimostrazione di questo secondo fatto è più complessa dato che l'istruzione **while** è un'istruzione di ciclo e la sua esecuzione può richiedere l'esecuzione del corpo del ciclo più volte.

Riguardo al primo punto, è banale verificare che W è soddisfatto. Infatti dopo aver posto q pari a zero e r pari a X abbiamo

$$Q * Y + R = 0 * Y + X = X$$

```
program modulo;
{calcola il valore di r = x mod y}
var x, y, q, r : integer;
begin
  readln (x,y);
  q := 0 ;
  r := x ;
  while r > y
  do begin
    r := r - y;
    q := q + 1
  end;
  writeln (r)
end.
```

Fig. 3.a

```
begin
  q := 0 ;
  r := x
end ;

while r > y
do begin
  r := r - y;
  q := q + 1
end;
```

Fig. 3.b

Fig. 3 - Programma per il calcolo del modulo

Inoltre la precondizione $X \geq 0$ assicura che anche $R \geq 0$.

Per quanto riguarda il secondo punto, dimostreremo ora la verità della seguente affermazione:

se il predicato W è vero prima di eseguire l'istruzione del corpo del ciclo è vero anche dopo che questa istruzione è stata eseguita.

Si noti che l'affermazione precedente implica che il predicato W è vero ogni volta che la condizione di terminazione dell'istruzione **while** viene verificata. Quindi se il predicato W è vero prima di eseguire l'istruzione **while** allora deve essere vero anche quando l'esecuzione dell'istruzione **while** è terminata. In altre parole, possiamo dire che il predicato W è un *invariante del ciclo*, nel senso che la sua verità non viene modificata dall'esecuzione del corpo del ciclo.

Per dimostrare che il predicato W è un invariante del ciclo osserviamo che: se W è il predicato condizione del ciclo **while** sono ambedue veri prima di eseguire il corpo del ciclo, i valori delle variabili soddisfano la seguente condizione:

$$(X = Q * Y + R) \text{ and } (R \geq Y)$$

Il primo termine deriva dal predicato W il secondo dalla condizione dell'istruzione **while**.

Denotiamo con R', Q', X', Y' i valori delle variabili r, q, x, y dopo una esecuzione del corpo del ciclo. Abbiamo

$$R' = R - Y$$

$$Q' = Q + 1$$

$$X' = X$$

$$Y' = Y$$

Dato che prima dell'esecuzione del ciclo R non è minore di Y dopo che il corpo del ciclo è stato eseguito il valore R' deve essere maggiore o uguale a zero. Inoltre la condizione W è soddisfatta anche dai nuovi valori. Abbiamo infatti che

$$\begin{aligned} Q' * Y' + R' &= (Q + 1) * Y + (R - Y) \\ &= Q * Y + R \\ &= X \end{aligned}$$

(perché il predicato W è vero prima di eseguire il ciclo)
 $= X'$

Osserviamo ora che quando l'istruzione **while** termina il valore della variabile r soddisfa il predicato Z e, quindi, la specifica del programma è soddisfatta.

Per completare la dimostrazione della correttezza del programma bisogna ancora mostrare che esso termina per ogni valore dei dati di ingresso che soddisfano il predicato P. Chiaramente è sufficiente dimostrare che l'istruzione **while** non conduce ad un ciclo infinito. Per mostrare ciò, osserviamo che ogni

volta che il corpo dell'istruzione **while** viene eseguito il valore della variabile *r* diminuisce mentre quello di *y* rimane immutato. Quindi, essendo il valore di *y* positivo (per la precondizione P), dopo un numero finito di iterazioni la condizione di terminazione del ciclo **while** deve essere falsa.

Si noti che, se il valore di *y* è nullo o negativo, il programma non termina (infatti in questo caso i dati di ingresso non rispettano la precondizione P).

2.1.2. Ricerca binaria

Il metodo della ricerca binaria è stato presentato nel par. 8 del cap. 3. Nel seguito dimostreremo la correttezza della procedura di fig. 4 che effettua la ricerca binaria di un valore *val* in un array di *n* elementi interi ordinati in senso crescente.

La precondizione che i dati di input devono soddisfare è:

- P) (*val* e i valori memorizzati nell'array sono interi) **and**
(gli elementi dell'array sono ordinati in senso crescente).

La postcondizione Q è la seguente:

- Q) se una componente dell'array ha un valore uguale a quello della variabile *val*, allora, al termine dell'esecuzione, la variabile booleana *trovata* è vera, falsa altrimenti.

Per dimostrare che la specifica è soddisfatta dobbiamo individuare un predicato R che risulti invariante rispetto al ciclo.

Utilizziamo ancora la convenzione introdotta precedentemente per distinguere il nome dal valore di un variabile. Si noti che nel caso di array A[MED] rappresenta il valore della componente di indice MED (MED è il valore della variabile *med*).

Denotiamo inoltre con

ELEM (A[i..j])

l'insieme dei valori che si trovano nelle componenti dell'array a comprese fra l'indice *i* e l'indice *j* inclusi.

Il predicato R invariante del ciclo è il seguente:

R = R1 **and** R2

dove R1 e R2 sono così definiti:

- R1) (TROVATO = true) \rightarrow (A[MED]=VAL)

R1 è soddisfatto se, nel caso che il valore della variabile *trovato* è vero, il valore della variabile *val* è pari al valore della componente in posizione

```

procedure ric_bin (val : integer; var trovato : boolean);
{ la procedura prende come parametro di ingresso un valore intero val e
fornisce come uscita un valore booleano trovato così definito:
al termine della procedura, se val è uguale ad uno degli elementi dell'array
a allora trovato è uguale a vero, altrimenti trovato è uguale a falso
l'array a è noto alla procedura ed è così definito
var a : array [1..n] of integer }
var inf, sup, med : integer;

begin
inf := 1;
sup := n;
trovato := false;
while (sup - inf >= 0) and (trovato = false)
do begin
    {1}           med := (inf + sup) div 2;
    {2}           if a[med] = val then trovato := true;
    {3}           if a[med] < val then inf := med + 1;
    {4}           if a[med] > val then sup := med - 1
end
end.

```

Fig. 4 - Programma di ricerca binaria

med dell'array *a*.

- R2) ((TROVATO = false) **and** (VAL appartiene a ELEM (A[1..N])) \rightarrow (VAL appartiene a ELEM (A[*inf*..*sup*]))

R2 è soddisfatto se, nel caso che il valore della variabile *trovato* è falso e il valore della variabile *val* è presente nell'array, allora l'elemento cercato è presente in una componente di indice compreso fra *inf* e *sup* inclusi.

Dopo aver eseguito le tre istruzioni di assegnazione alle variabili *trovato*, *inf* e *sup* il predicato R è banalmente verificato.

Mostriamo ora che esso è effettivamente un invarianto del ciclo **while**. Questo equivale a dimostrare che, se R è vero prima dell'esecuzione del corpo del ciclo e la condizione

$$((sup - inf) \geq 0) \text{ and } (trovato = \text{false})$$

è anch'essa vera, allora R è vero dopo aver eseguito il corpo del ciclo.

Analogamente all'esempio precedente siano INF' , SUP' , $\text{TROVATO}'$, MED' i valori delle variabili inf , sup , trovato e med dopo aver eseguito il corpo del ciclo. Concentriamo ora la nostra attenzione sulle istruzioni che costituiscono il corpo del ciclo (vedi la fig. 4). Osserviamo innanzitutto che l'istruzione 1 del corpo del ciclo assegna alla variabile med un valore compreso fra i valori delle variabili inf e sup .

Distinguiamo tre casi a seconda dell'esito del confronto fra dei valori di val e di $a[\text{med}]$.

1. Se VAL è uguale a $\text{A}[\text{MED}']$ allora l'istruzione 2 assegna il valore vero a trovato . Poiché le altre istruzioni del corpo del ciclo non modificano alcuna variabile allora sia R1 che R2 sono veri dopo aver eseguito il corpo del ciclo. Abbiamo infatti:

- 1.1. R1 è soddisfatto perché il nuovo valore di trovato è vero e VAL è uguale a $\text{A}[\text{MED}']$;
- 1.2. R2 è soddisfatto banalmente perché il nuovo valore di trovato è vero.

2. Se val è maggiore di $\text{A}[\text{MED}']$ allora l'istruzione 3 pone il valore di inf pari a

$$\text{MED}' + 1$$

Si noti che la successiva istruzione del corpo del ciclo non modifica alcuna variabile. Quindi, il valore di trovato rimane uguale a falso. Pertanto, dopo aver eseguito il corpo del ciclo, sia R1 che R2 sono veri. Infatti R1 è banalmente soddisfatto perché il valore di trovato è falso.

Per quanto riguarda R2 osserviamo che il valore della variabile sup non viene modificato; abbiamo

$$\text{SUP}' = \text{SUP}$$

Quindi, per dimostrare che R2 è soddisfatto dobbiamo mostrare che

$$\begin{aligned} & (\text{VAL appartiene a ELEM}(\text{A}[1..N])) \rightarrow \\ & (\text{VAL appartiene a ELEM}(\text{A}[\text{MED}' + 1..\text{SUP}])) \end{aligned}$$

Osserviamo che se il predicato R e la condizione dell'istruzione **while** sono ambedue vere prima di eseguire il corpo del ciclo, allora i valori delle variabili prima di eseguire l'istruzione 3 soddisfano la seguente condizione:

$$\begin{aligned} & (\text{VAL appartiene a ELEM}(\text{A}[1..N])) \rightarrow \\ & (\text{VAL appartiene a ELEM}(\text{A}[\text{INF}..\text{SUP}])) \end{aligned}$$

La condizione precedente esprime il fatto che, se VAL è presente nell'array

esso è presente in una componente di indice compreso fra INF e SUP inclusi. Quindi per dimostrare la condizione R2 dopo l'esecuzione del corpo del ciclo è sufficiente dimostrare che il valore di *val* non può essere presente in una componente dell'array di indice compreso fra INF e MED.

Assumiamo per assurdo che una componente di indice J compreso fra INF e MED sia pari a *val*. Dato che gli elementi dell'array sono ordinati in senso crescente abbiamo

$$\text{VAL} = A[J] \leq A[\text{MED}']$$

Osserviamo inoltre che $A[\text{MED}']$ è minore di VAL, dato che la condizione dell'istruzione 3 è vera.

Otteniamo pertanto che $\text{VAL} < \text{VAL}$. Questa contraddizione fa cadere l'ipotesi che il predicato R2 non sia soddisfatto.

3. Se il valore cercato è minore di $A[\text{MED}]$ allora l'istruzione 4 assegna alla variabile *sup* il valore $\text{MED}' - 1$ e lascia immutata la variabile *trovato*. In questo caso si può dimostrare che il predicato R è vero in modo analogo a quanto fatto nel caso precedente.

Per quanto riguarda la terminazione del programma *ric_bin*, osserviamo che ogni volta che si esegue il corpo del ciclo o la variabile *trovato* assume il valore vero oppure la differenza

$$(\text{SUP} - \text{INF})$$

diminuisce.

Infatti se non si verifica la condizione

$$a[\text{med}] = \text{val}$$

allora o si incrementa la variabile *inf* oppure si decrementa la variabile *sup*. Pertanto, dopo un numero finito di iterazioni del ciclo la condizione

$$(\text{sup} - \text{inf}) \geq 0$$

deve essere falsa e la procedura termina.

Poiché abbiamo dimostrato che la specifica del programma è soddisfatta e che quando i dati di ingresso verificano la precondizione P la procedura termina, possiamo concludere che la procedura *ric_bin* è corretta.

Per completare l'esempio della ricerca binaria consideriamo ora il programma *ric_bin_2* ottenuto sostituendo l'istruzione 3 del corpo del ciclo della procedura

ric_bin con l'istruzione

```
if  $a[med] < val$  then  $inf := med$  ;
```

Per quanto riguarda la procedura *ric_bin_2* osserviamo che:

1. la prova che la specifica del programma *ric_bin* è soddisfatta, può essere utilizzata per dimostrare la correttezza parziale di *ric_bin_2*;
2. la prova di terminazione di *ric_bin* non può essere applicata a *ric_bin_2*, perché non è più vero che durante una iterazione del ciclo o la variabile *inf* o la variabile *sup* cambiano valore. Si noti che la procedura *ric_bin_2* non è corretta. Infatti è facile individuare insiemi di dati di ingresso che soddisfano la precondizione P per i quali il programma *ric_bin_2* non termina.

2.2. Correttezza di programmi con procedure

Vediamo ora come estendere le dimostrazioni di correttezza al caso di programmi che utilizzano procedure.

Prima di proseguire è bene ricordare che come negli esempi precedenti anche in questo non stiamo sviluppando dimostrazioni formali di correttezza nel senso normalmente dato a questo termine, perché non utilizziamo una semantica formale del concetto di attivazione di procedura e di passaggio di parametri.

2.2.1. Ordinamento di un array

Supponiamo di dover ordinare in senso crescente un **array** di n interi così definito:

```
type vett = array [1..n] of integer;  
var a : vett;
```

Sulla base delle specifiche del problema la precondizione è data dal seguente predicato P:

P) ogni componente di *a* è un intero

Esistono diversi modi di definire la postcondizione del problema che si basano su equivalenti definizioni di array ordinato. Ad esempio si può assumere una delle due seguenti postcondizioni Q1 o Q2:

Q1) *a* è ordinato se per ogni indice *i*, con $1 \leq i \leq n$ l'elemento di indice *i* è minore o uguale di ogni elemento di indice *j*, dove $(i+1) \leq j \leq n$.

Q2) *a* è ordinato se per ogni indice *i*, con $1 \leq i \leq n$, l'elemento di indice *i* è

minore o uguale dell'elemento di indice $i+1$.

Nel seguito consideriamo l'algoritmo di ordinamento per selezione, che sarà ripreso nel par. 4 del cap. 6. Nella fig. 5 presentiamo una procedura che realizza l'algoritmo di ordinamento per selezione ed utilizza due procedure, *cercamin* e *scambia*. Le due procedure fanno riferimento alla variabile globale *a* e sono così definite:

- *cercamin* (*i,j,indmin*) che pone in *indmin* l'indice della componente più piccola compresa fra *i* e *j*;
- *scambia*(*i,j*) che scambia fra loro i valori delle componenti di indice *i* e *j*.

Per dimostrare formalmente la correttezza della procedura *ordina* dobbiamo innanzitutto decidere la postcondizione da verificare. Fra le due che sono state date precedentemente conviene scegliere la prima (Q1). Infatti il modo con cui Q1 definisce un insieme ordinato appare simile alla strategia che la procedura *ordina* utilizza per effettuare l'ordinamento. In questo modo sembra ragionevole supporre che la prova di correttezza risulti più semplice.

```
procedure ordina (var a: vett);  
  var i, indmax : integer;  
  begin  
    for i := 1 to n  
    do begin  
      cercamin (i,n,indmin);  
      scambia (i,indmin)  
    end  
  end;
```

Fig. 5 - Programma per l'ordinamento di un array

• La dimostrazione di correttezza parziale di un programma che utilizza procedure può essere effettuata nel seguente modo:

1. per ciascuna procedura *x* si definisce una precondizione *P(x)* e una postcondizione *Q(x)*;
2. per ciascuna procedura *x* si verifica che la specifica $\{P(x)\} \times \{Q(x)\}$ sia soddisfatta;
3. si verifica che la specifica del programma principale sia soddisfatta.

Nel nostro esempio le specifiche delle due procedure *cercamin* e *scambia* sono le seguenti:

1. *cercamin*

dati di ingresso: a è un array di tipo *vett*; i e j sono interi;

dati di uscita: $indmin$ è una variabile intera il cui valore è compreso fra l e n ;

$P(cercamin)$: i e j sono interi compresi fra l e n ;

$Q(cercamin)$: $indmin$ è l'indice della componente di minor valore dell'array a compresa fra i e j .

2. *scambia*

dati di ingresso: a è un array di tipo *vett*; i e j sono interi;

dati di uscita: a , array di tipo *vett*;

$P(scambia)$: i e $indmin$ sono compresi fra l e n ;

$Q(scambia)$: la procedura modifica l'array a scambiando fra loro le componenti di indice i e $indmin$.

Supponiamo di aver effettuato la prova di correttezza parziale delle due procedure e verifichiamo che la specifica della procedura ordina sia soddisfatta.

Osserviamo che se la postcondizione $Q(cercamin)$ è soddisfatta allora la precondizione di *scambia* è anch'essa soddisfatta. Quindi, se i dati di ingresso soddisfano $P(cercamin)$ e se eseguiamo le due procedure *cercamin* e *scambia* sequenzialmente, allora, al termine della prima iterazione del ciclo **for**, l'array a soddisfa $Q(scambia)$.

Chiaramente $P(cercamin)$ è soddisfatto quando il valore della variabile i è uguale a 1. Quindi, al termine della prima attivazione della procedura *cercamin*, il valore della variabile $indmin$ è uguale all'indice di una delle componenti di minor valore dell'array; la successiva attivazione della procedura *scambia* fa sì che, al termine della prima esecuzione del ciclo **for**, l'array a verifica la postcondizione Q nel caso i uguale a 1. La dimostrazione per gli altri elementi è analoga.

2.2.2. Prova di programmi ricorsivi

La prova di un programma ricorsivo non può essere eseguita utilizzando il metodo del paragrafo precedente perché non è possibile stabilire a priori il numero delle attivazioni ricorsive.

Illustriamo le problematiche connesse utilizzando un nuovo esempio.

Esempio. Provare la correttezza del programma di fig. 6 per il calcolo del fattoriale di un numero.

```

function fatt(n : interononneg) : interononneg;
begin
    if n = 0
        then fatt := 1
        else fatt := fatt(n-1) * n
    end;

```

Fig. 6 - Programma per il calcolo del fattoriale

In questo caso la precondizione P e la postcondizione Q sono:

P) *n* è un intero nonnegativo;

Q) se *n* è uguale a 0 allora il valore calcolato dalla funzione *fatt* è 1, altrimenti il valore di *fatt* è pari a

$$N * (N - 1) * (N - 2) * \dots * 2 * 1.$$

La prova di correttezza di programmi ricorsivi viene, generalmente, eseguita induttivamente. In questo caso opereremo per induzione sul valore di *n*.

Passo base (il valore di *n* è uguale a 0). Esaminando il corpo del programma osserviamo che la condizione dell'istruzione if è verificata. Pertanto la funzione *fatt* termina e fornisce il valore 1 che è proprio il fattoriale di 0.

Passo induttivo. Assumiamo che la funzione *fatt* sia corretta per *n*-1 e dimostriamo la correttezza per *n*. In questo caso, il valore della funzione *fatt* è dato dalla seguente espressione

$$fatt(n-1) * n$$

Osserviamo che *n*-1 è non negativo; quindi la precondizione P è soddisfatta e l'attivazione di *fatt*(*n*-1) termina correttamente per l'ipotesi induttiva fornendo il valore

$$[(N - 1) * (N - 2) * \dots * 2 * 1]$$

questo implica che anche l'attivazione di *fatt*(*n*) termina e fornisce un valore pari a

$$[(N - 1) * (N - 2) * \dots * 2 * 1] * N$$

che, per le proprietà commutativa e associativa del prodotto fra interi, è proprio il valore del fattoriale di *n*.

Seppure questo esempio sia particolarmente semplice, esso ci consente di

sottolineare che la prova di correttezza delle procedure ricorsive si può basare in modo naturale sull'induzione. Questo vale, a maggior ragione, quando la struttura di dati su cui la procedura opera è definita induttivamente (ad esempio liste, alberi).

2.2.3. Considerazioni pratiche sulle dimostrazioni formali

A conclusione del paragrafo osserviamo che gli esempi precedenti mostrano che le prove di correttezza possono essere, anche nei casi più semplici, lunghe e complicate.

Abbiamo anche visto che per analizzare l'esecuzione di programmi che contengono cicli è necessario definire dei predicati, gli invarianti dei cicli. Si noti che non esiste un metodo generale che, dato un programma, permetta di determinare gli invarianti dei cicli; ciò rende la dimostrazione della correttezza niente affatto semplice.

Considerazioni analoghe possono essere fatte anche per la dimostrazione della terminazione. Anche in questo caso non esiste un metodo generale per stabilire se un programma si ferma. Vedremo nel cap. 7 che il problema di stabilire se un programma termina non è risolubile nel caso generale: si dice in questo caso che il problema è *indecidibile*. Evidentemente questo non implica che in alcuni casi particolari, come negli esempi precedenti, sia invece possibile dimostrare con un metodo ad hoc la terminazione del programma.

2.3. Controllo di eventuali errori dei dati di ingresso

La definizione di correttezza di un programma data all'inizio del capitolo non specifica quale deve essere il comportamento del programma quando i dati di ingresso su cui viene eseguito non soddisfano la precondizione.

Questa indeterminatezza non tiene conto del fatto che nella maggioranza delle applicazioni si richiede che il programma sia in grado di rilevare e trattare correttamente eventuali anomalie presenti nei dati di ingresso. Infatti, se questo non si verifica, può accadere che il programma termini fornendo dei risultati che di fatto non sono significativi.

Naturalmente non sempre è possibile considerare tutti i possibili errori dei dati di ingresso. È importante, però, cercare di prevedere quali possano essere gli errori più probabili in relazione alla applicazione in esame. L'importanza di

un controllo sulla correttezza dei dati di ingresso dipende dalle particolari applicazioni. In alcuni casi ci si può limitare a considerare solo alcuni dei possibili errori, mentre in altri casi la verifica dei dati di ingresso è così importante che essa costituisce la parte principale del programma.

Illustriamo queste problematiche facendo riferimento ad un esempio.

Esempio. Si consideri il problema di sommare n numeri interi.

In fig. 7 sono dati quattro frammenti di programma. Si suppone che nel programma principale siano presenti le seguenti dichiarazioni

```
var  n : integer {memorizza quanti numeri bisogna sommare};  
      s : integer {contiene la somma di n numeri};  
      i : integer;  
      x : real;
```

Se i dati di input sono corretti i quattro programmi risolvono tutti correttamente il problema posto. Invece, essi si comportano in modo diverso se i dati di input sono scorretti. Infatti osserviamo che:

1. l'esecuzione del primo programma si interrompe a causa di un errore non appena viene letto un dato che non è un numero intero;
2. il secondo programma può leggere sia numeri interi che numeri reali; include solo i numeri interi nella somma, non segnala la presenza di numeri reali e li include fra gli n numeri da leggere;
3. il terzo programma può leggere sia numeri interi che numeri reali; include solo i numeri interi nella somma, segnala la presenza di numeri reali e li include fra gli n numeri da leggere;
4. il quarto programma può leggere sia numeri interi che numeri reali; include solo i numeri interi nella somma; segnala la presenza di numeri reali e non li include fra gli n numeri da leggere.

- , Questo esempio serve per illustrare il tipo di problema, ma non esclude tutte le possibilità di errori. Infatti tutti i programmi si fermano se erroneamente viene dato in input un carattere non numerico.

La fig. 8 mostra un nuovo esempio di programma per la soluzione dello stesso problema in cui vengono verificati anche altri errori nei dati di ingresso; in particolare vengono rilevati quei casi in cui invece di numeri vengono dati in ingresso dei caratteri. Il programma legge una stringa di caratteri e se questa sequenza rappresenta un numero intero allora calcola il valore letto in ingresso; altrimenti segnala l'errore fatto e richiede nuovamente la lettura di un intero.

Somma di n numeri interi 1

```
s := 0;  
read (n);  
for j := 1 to n  
do begin  
    read(i);  
    s := s + i  
end;
```

Somma di n numeri interi 2

```
s := 0;  
read (n);  
for j := 1 to n  
do begin  
    read(x);  
    if trunc(x) = x then s := s + x  
end;
```

Somma di n numeri interi 3

```
s := 0;  
read (n);  
for j := 1 to n  
do begin  
    read(x);  
    if trunc (x) = x  
    then s := s + x  
    else writeln ('numero non intero')  
end;
```

Somma di n numeri interi 4

```
s := 0;  
read (n);  
while n > 0  
do begin  
    read(x);  
    if trunc (x) = x  
    then begin  
        s := s + x;  
        n := n - 1  
    end  
    else writeln ('numero non intero')  
end;
```

Fig. 7 - Quattro frammenti di programma per la somma di n numeri interi

```

procedure leggi (var a : integer);
{la procedura legge una sequenza di caratteri e comunica al programma
 principale il valore del numero letto}
var c : char;
    fine : boolean; {fine segnala la fine della lettura}
    errore : boolean; {errore è vera quando è stato trovato un errore nei
                       dati di ingresso ed è necessario ricominciare la lettura};
    negativo : boolean; {la variabile negativo è vera quando il numero letto
                          è negativo}

begin
    fine := false;
    repeat
        writeln ('dammi un numero intero');
        {lettura di spazi bianchi che precedono il primo carattere}
        repeat
            read(c)
        until c <> ' ';
        {lettura del segno}
        negativo := false;
        if c = '+' then read (c);
        if c = '-'
        then begin
            negativo := true;
            read (c)
        end;
        {lettura del numero}
        errore := false;
        a := 0;
        repeat
            if (c > '0') and (c < '9')
            then begin
                a := a * 10;
                a := a + ord(c) - ord('0');
                read (c);
                if c = ' ' then fine := true
            end
            else begin
                write ('numero errato'); errore := true
            end
            until errore or fine
        until fine;
        if negativo then a := -a
    
```

```

end; {fine procedura leggi}
{ frammento di programma per la somma di n numeri interi }
s := 0;
leggi (n);
for j := 1 to n
do begin
    leggi (i);
    s := s + i
end;

```

Fig. 8 - Frammento di programma finale per la somma di n numeri interi

3. TEST DI UN PROGRAMMA

Abbiamo visto nel paragrafo iniziale che il test di un programma consiste nella sua esecuzione con dati di prova e nel verificare la correttezza dei risultati ottenuti. In questo paragrafo e nel successivo verranno considerate diverse strategie per individuare i dati di test.

È importante premettere che la prova di un programma non consiste semplicemente nell'eseguire il programma una o più volte con dati scelti in modo casuale, ma è un'attività che deve essere condotta e pianificata durante tutto il progetto del programma, come si vedrà meglio nel seguito.

L'esempio seguente mostra come una scelta casuale dei dati di prova non risulta adeguata.

Esempio. Due stringhe di caratteri alfabetici di lunghezza pari a i e j , rispettivamente, con i e j minori o uguali a 20, sono state memorizzate nelle prime componenti di due **array** v e w . I due **array** sono definiti nel seguente modo:

```
var v, w : array [1..20] of 'a'..'z';
```

Si consideri il seguente frammento di programma. Esso dovrebbe assegnare alla variabile booleana *uguali* il valore vero se le due stringhe sono uguali, il valore falso altrimenti.

```

if i = j
then for k := 1 to i
    do uguali := (v[k] = w[k])
else uguali := false ;

```

La soluzione precedente è chiaramente errata perché è sufficiente che due parole della stessa lunghezza abbiano l'ultimo carattere uguale affinché la variabile *uguali* sia posta a vero. Pertanto il frammento di programma con dati di ingresso 'bianco' e 'giallo' assegnerà alla variabile *uguali* il valore vero.

Si noti che una esecuzione del programma con una scelta casuale dei dati rende molto improbabile la scoperta dell'errore. Infatti l'errore può essere rilevato solo se i dati di prova soddisfano contemporaneamente le due seguenti condizioni:

- *i* è pari a *j*;
- *v[i]* è uguale a *w[j]*.

Con una scelta completamente casuale dei dati la prima condizione è vera con probabilità pari a $1/20 = 0.05$ (le stringhe hanno lunghezze comprese fra uno e venti), mentre la seconda è vera con probabilità pari a circa $1/26 = 0.038$ (i caratteri possibili sono 26). Pertanto una scelta casuale dei dati di prova ha una probabilità pari a $(0.05 \times 0.038) = 0.0019$ di evidenziare l'errore. Se si eseguono più prove allora la probabilità di trovare l'errore aumenta ma molto lentamente; infatti, la probabilità di trovare l'errore eseguendo cento prove casuali in modo indipendente è inferiore a 0.2.

I metodi utilizzati per la prova di un programma possono essere classificati in due categorie: metodi basati sulle specifiche del programma e metodi basati sulla struttura del programma.

I due diversi approcci non sono in alternativa fra loro; è invece opportuno che ambedue siano usati durante la prova del programma. Infatti non esiste un metodo che permette di trovare facilmente tutti gli errori, ma ciascuno permette di scoprire più facilmente alcuni tipi di errore. Pertanto, se si utilizzano diversi approcci invece di uno solo aumenta la probabilità di individuare tutti gli errori.

I metodi di test basati sulle specifiche del programma classificano i possibili dati di input in base alle loro caratteristiche e ricercano le condizioni in cui il programma non rispetta le specifiche senza considerare la struttura del programma. Per questa ragione essi sono detti anche metodi a scatola nera (da cui il nome di *test a scatola nera*).

I metodi di test basati sulla struttura del programma analizzano la logica del programma per cercare di evidenziare i possibili errori. Metodi di questo tipo sono detti a *scatola trasparente*, e di essi ne verrà considerato uno in particolare: il metodo dei criteri di copertura del programma. Con questo metodo, si ricercano dati di test in modo tale da provare tutte le parti del programma con il minimo numero possibile di esecuzioni.

3.1. Metodi basati sulle specifiche

I metodi di test a scatola nera ricercano le condizioni in cui il programma non si comporta secondo le specifiche date, senza entrare nei dettagli di come l'algoritmo è stato progettato e il programma è stato scritto.

Per dimostrare la correttezza di un programma utilizzando le sue specifiche è, in teoria, necessario provare il programma con tutti i possibili dati di ingresso. Come abbiamo già visto, questo in pratica non è possibile; pertanto è necessario individuare un insieme piccolo di dati che siano rappresentativi di situazioni diverse. In particolare in questo paragrafo si considera un particolare metodo di test a scatola nera basato sulla divisione dei possibili input in classi di equivalenza.

Il primo passo del metodo comporta la suddivisione dell'insieme dei possibili dati di ingresso in sottoinsiemi detti *insiemi di equivalenza*. Ciascun sottoinsieme è costituito da dati di ingresso con caratteristiche simili rispetto alle specifiche del programma. Quindi, si esegue il programma tante volte quanti sono gli insiemi di equivalenza e scegliendo ogni volta i dati di ingresso da un diverso insieme di equivalenza.

In questo modo si assume che tutti i dati di prova di un insieme di equivalenza diano le medesime informazioni. In altre parole si ipotizza che, se una prova non evidenzia un errore, allora ogni altro insieme di dati di ingresso appartenente allo stesso insieme di equivalenza non dia errore. Chiaramente l'affermazione precedente non può essere provata formalmente se non si tiene conto della struttura del programma, ma l'esperienza ha dimostrato che essa risulta utile in molti casi pratici.

La determinazione degli insiemi di equivalenza deve tener conto di due esigenze contrapposte. Da un lato, per poter compiere un limitato numero di prove è opportuno avere pochi insiemi di equivalenza, ciascuno dei quali necessariamente contiene molti possibili dati di ingresso; dall'altro, per effettuare un test adeguato ed avere informazioni sufficienti sul comportamento del programma, è preferibile definire un gran numero di insiemi di equivalenza.

In pratica, la determinazione degli insiemi di equivalenza viene fatta individuando un insieme di istanze tipiche del problema ed un insieme di istanze particolari che si possono verificare e che rappresentano casi limite.

Esempio. Si consideri il problema dell'inserimento di un valore in una lista di interi ordinata in senso crescente. Il nuovo valore, se non è già presente nella lista, deve essere inserito in modo tale da mantenere l'ordinamento.

Un insieme di insiemi di equivalenza per questo problema è dato in fig.9.

classe 1

input: lista vuota

classe 2

input: lista con un solo elemento diverso dall'elemento che deve essere inserito

classe 3

input: lista con un solo elemento uguale a quello che deve essere inserito

classe 4

input: lista con più di un elemento; l'elemento da inserire è più piccolo di tutti gli elementi della lista

classe 5

input: lista con più di un elemento; l'elemento da inserire è più grande di tutti gli elementi della lista

classe 6

input: lista con più di un elemento; l'elemento da inserire ha un valore compreso fra il più piccolo e il più grande di tutti gli elementi della lista

Fig. 9 - Alcune classi di equivalenza per il problema della ricerca di un elemento in una lista

3.2. Metodi basati sulla struttura del programma

Esistono diversi metodi per individuare dati di test che fanno riferimento alla struttura del programma. Nel seguito, ci limiteremo a considerarne uno: quello basato sulla *copertura del programma*. Si possono distinguere diversi criteri di copertura, che soddisfano in misura maggiore o minore l'esigenza di provare il programma.

1. *Copertura delle istruzioni:* un insieme di dati di test soddisfa questo criterio quando tutte le istruzioni del programma sono state eseguite almeno una volta.

Questo criterio di copertura del programma non è completamente soddisfacente, perché non permette di verificare in modo adeguato le decisioni del programma presenti nelle istruzioni condizionali e di ciclo.

2. Copertura delle decisioni: questo criterio richiede di eseguire un numero sufficiente di prove in modo che ogni decisione del programma (relativa alle istruzioni condizionali e di ciclo) assuma almeno una volta il valore vero e almeno una volta il valore falso.

Anche questo secondo criterio non è completamente soddisfacente perché non permette di verificare in modo adeguato tutte le condizioni del programma.

Infatti per verificare il criterio di copertura delle decisioni nel caso della seguente espressione

$$(x < y) \text{ or } (z > 2) \text{ or } (z < -10)$$

è sufficiente eseguire il programma con due insiemi di dati, uno che renda vero almeno una fra le tre condizioni

$$\begin{aligned} &(x < y) \\ &(z > 2) \\ &(z < -10) \end{aligned}$$

ed uno in cui tutte e tre le condizioni precedenti siano false. Un insieme di valori per le tre variabili può essere ottenuto con due prove:

- prova 1) $x = 0, y = 0, z = 10$ (l'or delle tre condizioni è vero);
prova 2) $x = 0, y = 0, z = 0$ (l'or delle tre condizioni è falso).

Si noti che in ambedue le prove la condizione $(x < y)$ è sempre falsa.

3. Copertura delle condizioni e delle decisioni: questo criterio richiede di eseguire un numero di prove sufficiente a ottenere per ogni condizione e per ogni decisione del programma i due possibili valori di vero e falso.

Ad esempio, per verificare il criterio di copertura delle decisioni nel caso della seguente espressione

$$(x < y) \text{ or } (z > 2) \text{ or } (z < -10)$$

è necessario eseguire un numero sufficiente di prove in modo tale che ciascuna delle tre condizioni

$$\begin{aligned} &(x < y) \\ &(z > 2) \\ &(z < -10) \end{aligned}$$

assuma il valore vero e il valore falso.

4. Copertura delle condizioni, delle decisioni e degli intervalli limite: questo criterio, oltre a verificare il criterio delle condizioni e delle decisioni richiede di provare per ciascuna condizione l'intervallo di valori in corrispondenza al quale la condizione cambia valore.

Ad esempio, con riferimento alla seguente espressione

$$(x < y) \text{ or } (z > 2) \text{ or } (z < -10)$$

se assumiamo che le variabili siano di tipo intero allora dobbiamo verificare il programma quando z risulta pari a 1, 2, 3 (intervallo limite per la condizione $z > 2$), -9, -10, -11 ($z < -10$) e quelli in cui x risulti uguale a y , a $(y-1)$ o $(y+1)$.

Chiaramente ciascuno dei quattro criteri visti risulta più forte del precedente e, in generale, richiede un maggior numero di prove. Ad esempio, se un insieme di dati verifica il criterio di copertura delle decisioni e delle condizioni, allora esso soddisfa anche il criterio di copertura delle decisioni; il viceversa non è sempre vero. Considerazioni analoghe possono essere fatte per gli altri criteri.

Poiché ciascun criterio richiede un maggior numero di prove, sembrerebbe ragionevole supporre che esso permetta di individuare un numero maggiore di errori. Questo non è sempre vero; può accadere, infatti, che un insieme di dati di test che soddisfa il criterio di copertura delle decisioni non permetta di rilevare errori, mentre un insieme che non verifica il suddetto criterio individui un errore.

La prova del programma con un insieme di dati tali da garantire la copertura delle decisioni, delle condizioni e degli intervalli limite non è ancora completamente soddisfacente, poiché non garantisce la verifica della correttezza del programma. Per questo motivo si possono proporre nuovi criteri di copertura del programma, più forti dei criteri visti precedentemente. Si noti che, procedendo in questo modo, si richiede l'esecuzione di un numero sempre maggiore di prove e, quindi, la verifica di questi nuovi criteri di copertura del programma risulta sempre più costosa.

Da un punto di vista pratico viene generalmente considerato sufficiente soddisfare il criterio di copertura delle decisioni o quello di copertura delle decisioni e delle condizioni.

Vediamo infine come possiamo verificare se un insieme di dati di prova soddisfa uno dei criteri di copertura visti.

Nel caso del criterio di copertura delle istruzioni si può usare il metodo cosiddetto dei *contatori*. Esso utilizza un insieme di variabili intere, una per ogni istruzione del programma, che hanno lo scopo di contare quante volte la

corrispondente istruzione è eseguita: esse sono poste a zero all'inizio del programma. Inoltre, in corrispondenza a ciascuna istruzione originaria del programma si inserisce una nuova istruzione che incrementa di uno il contatore corrispondente. In questo modo, dopo aver effettuato un insieme di prove, i contatori rimasti a zero indicano le istruzioni che non sono state mai eseguite. Il numero delle variabili contatore può essere ridotto utilizzando una sola variabile contatore per sequenze di istruzioni in cui non compaiono istruzioni condizionali o di ciclo.

La verifica degli altri criteri di copertura può essere fatta in modo analogo.

Esempio. Si consideri il programma di fig. 10.

Per verificare il criterio della copertura di ogni istruzione nel caso del programma di fig. 10, è sufficiente eseguire il programma una sola volta con i seguenti dati tali da rendere vere entrambe le condizioni: $a = 3$, $c = 15$. In fig. 11 è dato il programma modificato inserendo le variabili contatore.

Osserviamo che la prova fatta per verificare il criterio di copertura delle istruzioni non permette di rilevare i due seguenti errori:

Errore 1: scrivere $a > 0$ al posto di $a < 0$ nel primo test;

Errore 2: scrivere $(a < 10) \text{ or } (c < 20)$ al posto di $(a < 10) \text{ and } (c < 20)$ nel secondo test.

Per verificare il criterio di copertura delle decisioni nel caso della procedura di fig. 10 è necessario ottenere i valori vero e falso per le due decisioni della procedura. Un possibile insieme di prove che soddisfa il criterio è formato da una prima prova in cui le due condizioni sono false (ad esempio $a = 0$ e $c = 20$), ed una seconda prova in cui ambedue le condizioni sono vere (ad esempio $a = 6$ e $c = 15$).

Per verificare il criterio di copertura delle decisioni e delle condizioni osserviamo innanzitutto che la seconda decisione è costituita dalla congiunzione di due condizioni.

Pertanto, per soddisfare il criterio di copertura delle condizioni e delle decisioni, è necessario effettuare almeno una prova in cui a è minore di 10, una prova in cui a è maggiore o uguale a 10, una prova in cui c è minore di 20 ed una in cui c è maggiore o uguale a 20. Inoltre, le prove effettuate devono anche garantire che la congiunzione delle due condizioni sia, almeno una volta, vera e una volta falsa. Un insieme di dati di prova che verifica la copertura delle decisioni e delle condizioni è dato di seguito.

- prova 1: $a = -10$ $c = 15$:

```
procedure uno ( a, c : integer; var b : integer);  
begin  
    if (a <> 0)  
        then b := b + c;  
    if (a < 10) and (c < 20)  
        then b := 2 * b  
end;
```

Fig. 10 - Programma per la verifica dei criteri di copertura.

```
procedure uno ( a, c : integer; var b : integer);  
    var cont1, cont2 : integer;  
    begin  
        cont1 := 0; cont2 := 0;  
        if (a <> 0)  
            then begin  
                b := b + c;  
                cont1 := cont1 + 1  
            end;  
        if (a < 10) and (c < 20)  
            then begin  
                b := 2 * b;  
                cont2 := cont2 + 1  
            end  
        writeln('i valori dei contatori cont1 e cont2 sono', cont1,cont2)  
end;
```

Fig. 11 - Programma ottenuto dal programma di fig. 10 inserendo le variabili contatore

- prova 1: $a = 5$ $c = 10$:
 - la prima decisione è vera;
 - ambedue le condizioni della seconda decisione sono vere, quindi la seconda decisione è vera.
- prova 2: $a = -10$ $c = 25$:
 - la prima decisione è vera;
 - la prima condizione della seconda decisione è vera;
 - la seconda condizione della seconda decisione è falsa, quindi la seconda

decisione è falsa.

- prova 3: $a = 20$ $c = 15$:
la prima decisione è vera;
la prima condizione della seconda decisione è falsa;
la seconda condizione della seconda decisione è vera, quindi la seconda decisione è falsa.
- prova 4: $a = 0$ $c = 15$:
la prima decisione è falsa;
ambedue le condizioni della seconda decisione sono vere, quindi la seconda decisione è vera.

Questo esempio mostra che il numero di prove da effettuare per soddisfare il criterio può essere molto alto. Per evitare di dover effettuare un numero eccessivo di prove, è importante individuare quei dati di test che permettono di soddisfare contemporaneamente più decisioni e condizioni.

Nel caso precedente è possibile verificare il criterio di copertura delle decisioni e delle condizioni per la procedura di fig. 10 con sole tre prove. Abbiamo infatti che il seguente insieme di prove verifica il criterio.

- prova 1: $a = -10$ $c = 15$:
la prima decisione è vera;
ambedue le condizioni della seconda decisione sono vere.
- prova 2: $a = 0$ $c = 25$:
la prima decisione è falsa;
la prima condizione della seconda decisione è vera;
la seconda condizione della seconda decisione è falsa.
- prova 3: $a = 20$ $c = 15$:
la prima decisione è vera;
la prima condizione della seconda decisione è falsa;
la seconda condizione della seconda decisione è vera.

Osserviamo che, per la semplicità del programma, abbiamo avuto una limitata riduzione del numero di prove da effettuare. Si noti però che, nel caso di programmi più complessi, una attenta scelta dei dati può ridurre notevolmente il numero di prove da svolgere.

4. IL TEST DEI MODULI

È evidente che la possibilità di commettere errori aumenta grandemente con

la dimensione del programma. Quando le dimensioni del programma sono molto grandi è praticamente impossibile eseguire un test completo secondo i metodi visti precedentemente. Pertanto, è indispensabile utilizzare una strategia che, con un numero limitato di prove, permetta di confidare ragionevolmente nella correttezza del programma.

Osserviamo che i tipici errori di un grosso programma sono errori nella divisione del programma in sottoprogrammi e nella comunicazione delle informazioni fra i sottoprogrammi. Per questa ragione è utile organizzare il test del programma tenendo conto della sua struttura utilizzando, in particolare, l'albero delle procedure.

Una possibile strategia di test, detta di *test incrementale*, consiste nel provare inizialmente i singoli moduli del programma. Quindi si provano alcuni moduli insieme; quando il test di questo insieme di moduli è completato si procede in modo analogo aggiungendo ai moduli già provati nuovi moduli fino a quando non si arriva a provare il programma completo. Esistono due diversi metodi per effettuare un test di tipo incrementale: *test top-down* e *test bottom-up*.

Un diverso metodo, detto di *test nonincrementale*, è quello in cui si provano inizialmente i diversi moduli e successivamente si provano insieme tutti i moduli del programma.

4.1. Test top-down

Il test di tipo top-down inizia dalla prova del programma principale. Quindi, si prosegue con la prova dei moduli che sono chiamati dal programma principale. Quando anche questi moduli sono stati provati si prosegue con il test dei moduli che sono chiamati, a loro volta, da moduli già provati, fino a quando tutto il programma non è stato provato.

Non esistono criteri particolari per scegliere l'ordine migliore di esecuzione del test; è importante però cercare di provare il prima possibile le parti critiche del programma, per avere in tal modo più tempo per la loro correzione. Per parte "critica" di un programma si intende un modulo molto complesso, o particolarmente determinante per la soluzione del problema.

Dato che il test del programma inizia quando non è stato ancora completato, per provare un modulo può essere necessario preparare versioni semplificate, dette *stub*, dei sottoprogrammi attivati dal modulo che viene provato. Ciascun stub simula il comportamento di un sottoprogramma. In alcuni casi queste versioni semplificate possono essere anche dei semplici file di dati che

rappresentano i valori di uscita del sottoprogramma.

Il test di tipo top-down è più lungo e costoso del test di tipo non incrementale, perché richiede la preparazione degli stub, ma nel caso di grossi programmi è consigliabile perché presenta diversi vantaggi.

Il vantaggio principale è di anticipare nel tempo la rilevazione degli errori. In particolare si possono rilevare gli errori commessi nello scambio di dati tra i due moduli P e Q non appena si provano i due moduli insieme, mentre nel caso di test non incrementale, questo avviene solo nella fase finale del test. Infine, la correzione è generalmente più semplice, perché gli errori vengono rilevati man mano che si aggiungono nuovi moduli al test e questo facilita la loro individuazione.

Esempio. Supponiamo di dover scrivere un programma che legge da ingresso un valore a multiplo di dieci e calcola il numero di monete da 10, 50, 100, 500 lire il cui valore totale sia pari ad a , minimizzando il numero di monete da utilizzare.

Il programma principale utilizza le seguenti procedure:

1. *lettura (var a : integer)*

la procedura legge un valore intero e lo assegna ad a verificando la correttezza dei dati di ingresso;

2. *calcola (a : integer; var dieci, cinquanta, cento, cinquecento : integer)*

la procedura prende come parametro di ingresso il valore a e fornisce in uscita il numero di monete da 10, 50, 100, 500 il cui valore totale risulta pari ad a ;

3. *stampa (dieci, cinquanta, cento, cinquecento : integer)*

la procedura stampa secondo un opportuno formato i valori delle monete che costituiscono il cambio.

La parte istruzioni del programma principale è:

```
begin  
lettura (cambio);  
calcola (cambio, dieci, cinquanta, cento, cinquecento);  
stampa (dieci, cinquanta, cento, cinquecento)  
end.
```

Il test di tipo top-down inizia con la prova del programma principale. A questo scopo è necessario preparare degli stub che simulino il comportamento delle procedure attivate dal programma. Nel caso della procedura *lettura* il modulo stub può essere una procedura che legge un intero e lo assegna alla

variabile *a*; la differenza fra il modulo stub e la procedura *lettura* del nostro programma è data dal fatto che il modulo stub non effettua la verifica della correttezza dei dati di ingresso.

Per quanto riguarda la procedura *calcola* lo stub corrispondente può essere il seguente:

```
procedure calcola (a:integer; var dieci, cinquanta, cento, cinquecento : integer);
begin
writeln('scrivi il numero di monete da dieci, cinquanta, cento, cinquecento che
corrispondono a ', a);
readln(dieci, cinquanta, cento, cinquecento)
end;
```

Infine, per quanto riguarda la procedura *stampa*, lo stub corrispondente può essere costituito da istruzioni di stampa dei valori senza dover peraltro rispettare il formato di stampa specificato.

4.2. Test bottom-up

Il test bottom-up si differenzia dal test top-down per la diversa strategia di esecuzione dei moduli. In un test bottom-up un modulo è provato dopo che tutti i moduli che sono utilizzati da lui siano scritti e provati precedentemente.

Chiaramente in questo modo l'ultimo modulo ad essere provato è il programma principale. Come nel caso del test top-down non esistono criteri particolari per scegliere il particolare ordine di esecuzione del test che rispetta la regola data.

Analogamente al test top-down, per poter effettuare il test di un modulo diverso dal programma principale è necessario preparare un file che simuli il comportamento dei moduli che attivano il modulo che viene provato. Questi file vengono detti *driver*.

Rispetto al test top-down osserviamo che il test bottom-up permette di provare ciascun modulo in modo completo (cosa che non è possibile nel test top-down). Inoltre la preparazione dei driver risulta, in genere, più semplice dell'analogia preparazione degli stub.

Il vantaggio del test top-down rispetto a quello bottom-up è essenzialmente quello di poter verificare la correttezza della suddivisione del programma in moduli prima che tutti i moduli del programma siano stati scritti. In questo

modo errori nella logica del programma principale possono essere rilevati prima che la scrittura del programma sia completata ottenendo un risparmio di tempo e di risorse.

In conclusione non esiste una strategia di test che risulta sempre migliore dell'altra: in alcuni casi conviene usare una strategia top-down, in altri una bottom-up, in altri ancora una strategia di tipo misto, in cui si alternano fasi di test top-down ad altre in cui si effettua un test bottom-up. Inoltre, la scelta del metodo di test è ovviamente influenzata dalla scelta della strategia (top-down o bottom-up) utilizzata per il progetto del programma.

5. LA CORREZIONE DEGLI ERRORI

Il primo passo della fase di correzione è individuare il punto del programma che contiene l'errore. Per indicare l'operazione di individuazione degli errori viene usata la parola inglese *debugging*, letteralmente "tirare via i bachi", ormai entrata nell'uso comune.

Purtroppo, questa operazione risulta spesso non facile perché le informazioni che si ottengono quando un errore si manifesta non sono generalmente sufficienti a permettere di individuare l'esatto punto del programma dove si trova l'errore.

Si noti che, in molti altri campi diversi dalla programmazione, l'individuazione dei guasti è più semplice. Ad esempio, se esce del fumo bianco dal radiatore di un'automobile, è molto probabile che ci sia un guasto nel sistema di raffreddamento. In questo caso esiste una correlazione fra il sintomo (il fumo nel radiatore) e la causa che lo ha generato. Questa correlazione non è generalmente così evidente nel caso della programmazione: l'errore rilevato in una certa zona di un programma può essere localizzato, potenzialmente, in una qualunque istruzione del programma stesso. L'operazione di debugging è resa ancor più difficile dalla mancanza di metodi per l'individuazione degli errori che garantiscano in ogni caso di localizzare l'errore.

Un metodo di ricerca degli errori può essere così riassunto:

1. utilizzando le informazioni disponibili sul tipo di errore si ipotizza la presenza dell'errore in una particolare zona del programma (cioè in un gruppo di istruzioni);
2. si cerca di ottenere informazioni utili sulle variabili nella zona del programma dove si pensa che sia presente l'errore. È importante cercare di evidenziare le condizioni in cui l'errore si verifica e le relazioni con i valori che le

variabili assumono;

3. sulla base delle informazioni ottenute si effettua un'ipotesi sulle cause dell'errore. Se le informazioni disponibili non sono sufficienti è necessario ottenere ulteriori informazioni, oppure formulare una nuova ipotesi sulla zona del programma in cui si trova l'errore; in ambedue questi casi si ritorna al punto 2;
4. dopo aver formulato l'ipotesi sulla causa dell'errore è importante verificare che l'ipotesi fatta sia corretta. Se si salta questo passo può accadere di "correggere" il programma in un punto in cui esso funziona perfettamente, introducendo così nuovi errori;
5. l'ultimo passo è quello della correzione effettiva dell'errore.

È importante sottolineare che l'intero processo di correzione appena descritto va condotto facendo uso dei metodi per la verifica e la prova dei programmi visti precedentemente nel capitolo. Ciò per assicurarsi che la correzione di un errore non introduca nuovi errori.

Per ottenere maggiori informazioni sul tipo di errore (vedi punto 2 precedente) si può utilizzare il cosiddetto *metodo delle stampe*.

Con questo metodo si inseriscono nel programma istruzioni di stampa che permettono di controllare il comportamento del programma in quella parte che supponiamo contenga l'errore. Quindi, si esegue nuovamente il programma e si analizzano le informazioni ottenute. Le istruzioni di stampa che vengono inserite devono permettere di controllare passo passo i valori delle variabili "sospette" nella zona critica del programma.

Ad esempio, la stampa dei valori di tutte le variabili modificate nel ciclo permette di controllare l'esecuzione del ciclo stesso. Quando il programma viene nuovamente eseguito, allora eventuali difformità fra i valori stampati e quelli previsti possono aggiungere nuove informazioni sulla localizzazione e sul tipo di errore. Se le informazioni che si ottengono non permettono di individuare l'errore, è necessario ottenerne di nuove inserendo altre istruzioni di stampa.

È importante inserire istruzioni di stampa con moderazione limitandosi a quelle che si ritiene siano utili alla individuazione dell'errore. Pertanto, per limitare il numero di valori che si ottengono, può essere opportuno eliminare alcune delle istruzioni di stampa precedentemente inserite che non sembrano significative per trovare l'errore.

Supponiamo, infatti, di richiedere la stampa dei valori di tutte le variabili del programma che contiene cicli che vengono ripetuti molte volte. L'insieme di

informazioni che si ottiene in questo modo è di dimensioni tali da rendere quasi impossibile la loro analisi.

Sfortunatamente, non esistono regole generali per determinare le variabili e i punti del programma in cui inserire le istruzioni di stampa. Questa operazione è fatta dal programmatore utilizzando la sua esperienza, il suo intuito, la sua conoscenza del programma e, infine, il comportamento del programma in corrispondenza all'errore osservato.

Esempio. Supponiamo di dover risolvere il seguente problema: leggere n numeri, sommare separatamente i positivi e i negativi e stampare le due somme

```
program conta_numeri;
var sommapos, sommaneg, i, numero, n: integer;
begin
  readln (n);
  i := 1;
  while i <= n do
    begin
      sommapos := 0;
      sommaneg := 0;
      readln (numero);
      if numero < 0
        then sommapos := sommapos + numero
      else sommaneg := sommaneg + numero;
      i := i+1
    end;
    writeln (sommapos);
    writeln (sommaneg)
end.
```

Fig. 12 - Programma errato per la somma di numeri positivi e negativi

(prima la somma dei positivi, poi quella dei negativi) e di aver prodotto il programma in fig. 12.

Osservando il programma, è facile rendersi conto che contiene almeno due errori:

1. ad ogni esecuzione del ciclo, le variabili *sommapos* e *sommaneg* vengono inizializzate al valore 0: dovrebbero al contrario essere inizializzate esternamente al ciclo, una volta per tutte;
2. l'espressione booleana utilizzata nella istruzione *if* è tale che *sommapos*

somma i numeri negativi e *sommaneg* i positivi.

Supponiamo che dopo la prova del programma abbiamo notato alcuni malfunzionamenti. Si tratta a questo punto di individuarne la causa, cioè i punti del programma affetti da errore. Un insieme di numeri che contiene almeno un numero positivo e almeno un numero negativo, come ad esempio

3 15 -5 3 -9 0 12

permette di eseguire tutte le istruzioni del programma.

Una stampa intermedia dei valori parziali delle somme dei numeri positivi e negativi, può essere utilmente inserita dopo la istruzione **if**, al fine di controllare che la somma venga effettuata correttamente numero per numero. La fig. 13 rappresenta il programma modificato, mentre la tabella di fig. 14 fornisce l'insieme dei valori che vengono stampati.

```
program conta_numeri;
var sommapos, sommaneg, i, numero, n: integer;
begin
  readln (n);
  i:= 1;
  while i <= n do
    begin
      sommapos:= 0;
      sommaneg:= 0;
      readln (numero);
      if numero < 0
        then sommapos:= sommapos + numero
        else sommaneg:= sommaneg + numero;
      writeln (i, numero, sommapos, sommaneg);
      i:= i+1
    end;
  writeln (sommapos);
  writeln (sommaneg)
end.
```

Fig. 13 - Programma ottenuto dal programma di fig. 12 con la inserzione di una stampa intermedia

L'analisi delle stampe prodotte conferma i comportamenti anomali del programma: ci permette di osservare che le variabili *sommapos* e *sommaneg* non memorizzano le somme parziali, ma solo l'ultimo valore; in più, *sommaneg* assume valori positivi mentre *sommapos* assume valori negativi. Possiamo allora correggere il programma, portando le due istruzioni di inizializzazione fuori dal ciclo (vedi fig. 15).

<i>i</i>	<i>numero</i>	<i>sommapos</i>	<i>somaneg</i>
1	3	0	3
2	15	0	15
3	-5	-5	0
4	3	0	3
5	-9	-9	0
6	0	0	0
7	12	0	12

Fig. 14 - Stampe parziali per il programma di fig. 13

```

program conta_numeri;
var sommapos, sommaneg, i, numero, n: integer;
begin
  readln (n);
  i := 1;
  sommapos := 0;
  sommaneg := 0;
  while i <= n do
    begin
      readln (numero);
      if numero < 0
        then sommapos := sommapos + numero
        else sommaneg := sommaneg + numero;
      writeln (i, numero, sommapos, sommaneg);
      i := i+1
    end;
  writeln (sommapos);
  writeln (sommaneg)
end.

```

Fig. 15 - Programma ottenuto dal programma di fig. 13 correggendo il primo errore rilevato

<i>i</i>	<i>numero</i>	<i>sommapos</i>	<i>somaneg</i>
1	3	0	3
2	15	0	18
3	-5	-5	18
4	3	-5	21
5	-9	-14	21
6	0	-14	21
7	12	-14	33

Fig. 16 - Stampe parziali per il programma di fig. 15

Se ora rieseguiamo il programma con gli stessi dati della prova precedente otteniamo la stampa di fig. 16.

I dati ottenuti mostrano che il secondo errore continua a presentarsi. Dalle nuove informazioni ottenute è ora chiara la causa dell'errore: l'inversione di ruolo tra le due variabili. Possiamo allora correggere il programma modificandone opportunamente la espressione booleana nella istruzione if; si ottiene in tal modo il programma corretto (non riproduciamo il programma, peraltro facilmente ottenibile da quello di fig. 15).

6.

Analisi di programmi: la complessità

La valutazione dell'efficienza degli algoritmi e la ricerca di algoritmi sempre più efficienti sono gli obiettivi di quella parte dell'informatica nota come *complessità di calcolo*; essa costituisce l'argomento di questo capitolo.

Nei primi due paragrafi verranno esaminati i concetti fondamentali che costituiscono le basi per analizzare i programmi dal punto di vista delle risorse richieste.

Nei paragrafi successivi questi concetti verranno applicati a due diversi problemi. In particolare, nel par. 3 verrà considerato il problema della gestione di una tavola, e, nel par. 4, il problema dell'ordinamento di un insieme. Nel paragrafo finale faremo alcune considerazioni conclusive sul progetto di programmi efficienti.

1. EFFICIENZA DEI PROGRAMMI

Un programma è tanto più efficiente quanto minore è l'utilizzo di risorse di calcolo necessarie per la sua esecuzione. Le risorse di calcolo che si considerano sono due: il tempo di calcolo e la quantità di memoria necessari per l'esecuzione. Nel seguito considereremo prevalentemente la risorsa *tempo*, valutando così il costo o la *complessità* del programma. Pertanto diremo che un programma è tanto più efficiente quanto meno tempo viene richiesto per la sua esecuzione.

Un primo modo di valutare il costo di un programma è quello di esprimere il tempo di calcolo con unità di misura solari, come, ad esempio, i secondi. In questo caso la valutazione del costo di un programma è, in teoria, molto semplice: basta eseguire il programma disponendo di un orologio con cui

misurare il tempo impiegato. Inoltre, se si vogliono confrontare due programmi e stabilire quale sia il più efficiente, è sufficiente eseguirli e confrontarne i rispettivi tempi di esecuzione per individuare il migliore.

Anche se questo metodo di valutazione è apparentemente corretto, i risultati che si ottengono non sono validi perché dipendono in modo essenziale dalle particolari condizioni in cui si effettuano le prove. Infatti essi dipendono:

1. dalla macchina e dal compilatore utilizzati: il confronto di due programmi è valido solo se li traduciamo con il medesimo compilatore e li eseguiamo con lo stesso elaboratore;
2. dai dati di ingresso ai due programmi. Per trarre una conclusione valida è necessario che i due programmi ricevano in ingresso gli stessi dati. Inoltre per stabilire quale dei due programmi sia più efficiente non è sufficiente eseguire i due programmi una sola volta, ma più volte con dati differenti. In questo caso è necessario stabilire dopo quante esecuzioni possiamo terminare il confronto.

Le osservazioni precedenti portano alla conclusione che non è possibile valutare il costo di un programma in unità di tempo solari, perché in tal modo non si effettua un'analisi sufficientemente rigorosa e completa del suo costo. Nel seguito vedremo come sia possibile esprimere il costo di un programma tramite una funzione che non dipende dal particolare sistema di elaborazione a disposizione, dal compilatore usato e dai particolari dati di ingresso utilizzati. In questo modo è possibile ottenere valutazioni oggettive; tuttavia i risultati che questo metodo fornisce sono approssimati, perché esso si basa su alcune ipotesi semplificative, che verranno considerate in dettaglio nel seguito di questo paragrafo.

1.1. Il modello di costo

L'analisi della complessità di un programma è basata sulla ipotesi che il costo di esecuzione di ogni istruzione semplice e di ogni operazione di confronto sia pari a una unità di costo indipendentemente dal linguaggio e dal sistema usato.

Il costo delle altre istruzioni dipende dal numero di istruzioni semplici o di test che sono richiesti. In particolare, nel caso del linguaggio Pascal si assume che:

1. il costo di esecuzione di ogni istruzione semplice (istruzione di assegnazione, di lettura, di scrittura) è 1;

2. il costo di un'istruzione composta è pari alla somma dei costi delle istruzioni che la compongono;
3. il costo di un'istruzione di ciclo è dato dalla somma del costo totale di esecuzione del test di fine ciclo e dal costo totale di esecuzione del corpo del ciclo. Assumiamo che il costo di esecuzione del test sia unitario e, quindi, il primo termine è pari al numero di volte che il ciclo viene ripetuto. Il secondo termine è pari alla somma dei costi di esecuzione delle istruzioni del corpo del ciclo tenendo conto di quante volte ciascuna istruzione del ciclo è eseguita;
4. il costo di un'istruzione di tipo **if ... then ...** è dato dal costo di esecuzione del test (che assumiamo unitario) più il costo di esecuzione della istruzione che segue la parola **then**, se la condizione è vera. In modo analogo si valuta il costo di esecuzione di un'istruzione **if ... then ... else**;
5. il costo di una attivazione di una procedura (o di una funzione) è pari alla somma dei costi di esecuzione di *tutte* le istruzioni che la compongono tenendo eventualmente conto del fatto che all'interno della procedura (o funzione) possono essere presenti altre attivazioni di procedure o funzioni. In questo modo assumiamo che il costo di attivazione della procedura (passaggio di parametri, allocazione di variabili locali ecc.) sia zero.

Le ipotesi precedenti eliminano la dipendenza dalla macchina e dal compilatore nell'analisi della complessità di un programma, ma portano a risultati approssimati. Infatti, si assume che il costo dell'istruzione di assegnazione:

$$A := A + 1$$

in cui la valutazione dell'espressione richiede una addizione, sia pari al costo dell'istruzione

$$A := B * (C + 3) + ((D + 19) \text{ div } (5 + A * 2)) + 2 * E$$

in cui la valutazione dell'espressione richiede 5 addizioni, 3 moltiplicazioni e una divisione. Questo chiaramente non è vero: la seconda istruzione comporta l'esecuzione di un numero maggiore di operazioni; si noti inoltre che le operazioni di moltiplicazione e di divisione richiedono un maggior tempo di calcolo della addizione.

Osserviamo però che, se indichiamo con $t(1)$ il tempo di esecuzione della prima istruzione e con $t(2)$ quello della seconda istruzione, allora esiste una costante intera c tale che

$$c t(1) > t(2)$$

In altre parole $t(2)$ è maggiore di $t(1)$ al più per un fattore costante c , e, quindi, l'ipotesi di assumere il costo di ogni istruzione pari a 1 è esatta a meno di un fattore costante. Un ragionamento analogo può essere fatto per ogni istruzione di un qualunque programma e, quindi, possiamo concludere che *il costo di un programma viene valutato a meno di un fattore costante*.

Un altro vantaggio di queste semplificazioni è quello di poter prescindere dal linguaggio di programmazione in cui viene scritto il programma e di poter far riferimento direttamente all'algoritmo descritto in linguaggio naturale. Nel seguito, parleremo del costo di esecuzione (o della complessità) di un algoritmo intendendo il costo di esecuzione di un programma, scritto in un qualunque linguaggio di programmazione, che lo realizza.

Il secondo problema che è necessario affrontare per giungere ad un'analisi rigorosa della complessità di un programma è stabilire in funzione di quali parametri deve essere definita la sua complessità.

Il costo di esecuzione di un programma dipende quasi sempre dai dati di ingresso; ad esempio, il tempo di esecuzione di un programma P di ordinamento di un insieme di numeri dipende dalle dimensioni dell'insieme che si considera: generalmente un programma impiega meno tempo per ordinare un insieme di 10 elementi che per ordinare un insieme di 10000 elementi.

Pertanto, per ottenere una valutazione rigorosa del tempo di esecuzione è necessario individuare la funzione $f(n)$, che esprime il numero delle istruzioni semplici e di test eseguite in funzione del numero n dei dati del problema, cioè in funzione della *dimensione dell'input*.

Ad esempio, la complessità di un programma di ordinamento può essere espressa da una funzione del tipo

$$n(n+1)/2 + n + 5$$

In questo caso si intende che $n(n+1)/2 + n + 5$ è il numero di operazioni e confronti richiesti per ordinare insiemi di n elementi.

In molti casi il costo di esecuzione del programma dipende non solo dalle dimensioni dell'input, ma anche dai particolari valori dei dati stessi. In particolare, è possibile distinguere diversi casi: il caso migliore, il caso peggiore, il caso medio. Nel seguito si valuterà la complessità facendo riferimento generalmente al caso peggiore e talvolta al caso medio.

Nella valutazione di un algoritmo dal punto di vista del caso peggiore si fa riferimento a quei valori dei dati di ingresso per cui il costo di esecuzione è maggiore. Invece, nella valutazione di un algoritmo dal punto di vista del caso medio, si valuta la media aritmetica dei costi delle esecuzioni rispetto a tutti i

possibili dati di ingresso.

L'esempio seguente illustra i concetti visti.

Esempio. Consideriamo il programma per la ricerca esaustiva di un elemento in una tavola con n elementi, rappresentata tramite un array.

In questo caso possiamo assumere che la dimensione dell'input sia data dal numero di elementi dell'array.

Consideriamo ora il programma di ricerca esaustiva in una tavola dato in fig. 1.

Il costo di esecuzione del programma dipende dalla posizione del particolare elemento che si vuole individuare: se l'elemento cercato è il primo della tavola allora si effettua un solo confronto; se l'elemento cercato è il secondo della tavola allora si effettuano due confronti e così via. Il caso peggiore è costituito dalla ricerca dell'ultimo elemento o da una ricerca infruttuosa, perché in questo caso l'algoritmo esamina tutte le componenti dell'array ed esegue il ciclo n volte.

Valutiamo ora il costo del programma nel caso di ricerca dell'ultimo elemento:

- l'istruzione 1 è eseguita 1 volta;
- l'istruzione 3 è eseguita n volte;
- il test dell'istruzione **repeat** è eseguito n volte;
- il test dell'istruzione **if then else** è eseguito 1 volta;
- l'istruzione in 6 è eseguita 1 volta;
- l'istruzione in 7 non viene eseguita.

Pertanto il costo di esecuzione del programma è

$$1 + n + n + 1 + 1 = 3 + 2n$$

È facile vedere che la stessa funzione esprime anche il costo del programma nel caso di ricerca infruttuosa.

Se si vuole valutare il comportamento del programma nel caso medio, è necessario distinguere il caso di ricerca con successo da quello di ricerca infruttuosa. Nel caso di ricerca fruttuosa, se si assume che tutti gli elementi dell'array possano essere ricercati con uguale probabilità pari a $1/n$, allora è facile vedere che il programma richiede mediamente $(n+1)/2$ confronti. Infatti, se ricerchiamo l' i -esimo elemento si effettuano i confronti e, quindi, il numero di confronti medio è dato da

$$\sum_{1 \leq i \leq n} \text{Prob}(E(i)) i = \sum_{1 \leq i \leq n} i/n = (n+1)/2$$

dove $\text{Prob}(E(i))$ è la probabilità che l'elemento da cercare nell'array sia quello

```

procedure ricerca_esauriva (var t : tipotavola;
                           k : tipochiave; var trovato : boolean);

{la procedura ricerca nella tavola l'elemento avente chiave k; se la ricerca
ha successo allora la variabile booleana trovato viene posta a true,
altrimenti a false}

var i : integer;

begin
  {1}   i := 0;
  {2}   repeat
  {3}     i := i + 1
  {4}   until (t[i].chiave = k) or (i = n);
  {5}   if (t[i].chiave = k)
  {6}     then trovato := true
  {7}   else trovato := false
end;

```

Fig. 1 - Programma per la ricerca esauriva in una tavola

in posizione i.

Osserviamo inoltre che, nel caso di ricerca infruttuosa, il programma richiede ogni volta di effettuare n confronti.

1.2. Comportamento asintotico

Individuare esattamente la funzione che esprime il costo di un algoritmo è, nella maggioranza dei casi, molto difficile. Generalmente, per semplificare l'analisi, non si cerca di individuare esattamente la funzione che esprime il costo del programma ma si ritiene sufficiente stabilire il suo comportamento asintotico quando le dimensioni dell'input tendono all'infinito. Più precisamente, si cerca di valutare l'andamento della funzione che esprime il costo del programma a meno di costanti moltiplicative e di termini additivi di ordine inferiore. Diremo, ad esempio, che un programma o un algoritmo ha complessità lineare se il suo costo in funzione delle dimensioni dell'input è

$$c_1 n + c_2$$

dove c_1 e c_2 sono costanti intere (n rappresenta le dimensioni dell'input).

Ad esempio, due programmi i cui costi sono espressi rispettivamente dalle funzioni $(3 + n)$ e $(100n + 3027)$, sono caratterizzati dalla stessa complessità asintotica, e cioè da una complessità lineare. Si noti che la funzione $(n+3)$ esprime un costo minore della funzione $(100n + 3027)$ e, quindi, ignorare la costante moltiplicativa (pari in un caso a 1 e nell'altro a 100) e il termine additivo di ordine inferiore (pari in un caso a 3 e nell'altro a 3027) può essere in alcuni casi una semplificazione eccessiva. Pertanto, i risultati che si ottengono nella valutazione della complessità di un programma devono essere interpretati e valutati tenendo conto delle semplificazioni fatte. È importante però sottolineare che ignorando le costanti moltiplicative e i termini additivi di ordine inferiore, l'analisi per stabilire il costo di un algoritmo viene molto semplificata e le valutazioni ottenute, pur con questa approssimazione, permettono di giungere a risultati significativi nella maggioranza dei casi.

2. LA COMPLESSITÀ DI UN PROGRAMMA E DI UN PROBLEMA

Le osservazioni e le considerazioni del paragrafo precedente permettono di esprimere il costo di esecuzione di un programma, o di un algoritmo, come una funzione delle dimensioni dell'input in cui si ignorano le costanti moltiplicative. Nel resto del paragrafo verranno definite diversi tipi di delimitazioni che formalizzano il concetto di complessità di un programma (o di un algoritmo) e di un problema. Nel seguito si assume che n rappresenti il parametro in funzione del quale viene espresso il costo di un programma o di un algoritmo.

2.1. Le notazioni O e Ω

In base alle considerazioni svolte, la complessità di un programma è lineare quando il suo costo di esecuzione è, ad esempio, $2n$; analogamente, la complessità di un algoritmo è quadratica se il suo costo di esecuzione è una funzione quadratica nelle dimensioni dell'input, come ad esempio $n^{**}2 + 2n$ (qui e nel seguito $n^{**}i$, i intero, indica la potenza i -esima di n ; pertanto $n^{**}2$ indica il quadrato di n e $2^{**}n$ indica una funzione esponenziale).

In generale si utilizza la seguente definizione.

Definizione 1. Un programma (o un algoritmo) ha *costo* $O(f(n))$, o ha *complessità* $O(f(n))$, se esistono opportune costanti a , b e n' tali che il numero

di istruzioni $t(n)$ che vengono eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) < a f(n) + b$$

per ogni $n > n'$.

Utilizzando la definizione precedente possiamo scrivere, ad esempio, che

$$50n^{**2} + 10n + 100 = O(n^{**2})$$

$$0.5n^{**3} + 1000n^{**2} = O(n^{**3})$$

$$2^{**n} + 100 = O(2^{**n}).$$

Se, invece, il programma richiede, per ogni possibile input, l'esecuzione di un numero costante di operazioni, allora la sua complessità è $O(1)$.

Si noti che nella definizione 1 è sufficiente ottenere una valutazione per eccesso. Quindi, in base alla definizione, se un programma ha costo $O(n)$ banalmente ha anche costo $O(n^{**2})$. È chiaro che in questo caso $O(n)$ esprime in modo più adeguato di $O(n^{**2})$ il costo del programma con complessità lineare. In generale tra le varie funzioni $f(n)$ tali che il costo del programma è $O(f(n))$ si cerca, quindi, la minorante.

La notazione O fornisce una delimitazione superiore al costo di esecuzione di un algoritmo, cioè fornisce una valutazione approssimata per eccesso.

Per specificare una delimitazione inferiore introduciamo la notazione Ω .

Definizione 2. Un programma (o un algoritmo) ha *costo* $\Omega(g(n))$, o ha *complessità* $\Omega(g(n))$ se esiste una opportuna costante positiva c tale che il numero di istruzioni $t(n)$ che vengono eseguite nel caso peggiore con input di dimensione n verifica

$$t(n) > c g(n)$$

per un numero infinito di valori di n .

Si noti la differenza tra le due definizioni di delimitazione superiore ed inferiore alla complessità di un algoritmo. In base alla prima definizione un algoritmo ha costo $O(f(n))$ se, per ogni n e per ogni input di dimensione n , impiega una quantità di risorse proporzionale a $f(n)$ (a meno di costanti additive). Invece, nella seconda definizione, un algoritmo ha costo $\Omega(g(n))$ se esiste una sequenza infinita di istanze del problema aventi dimensioni crescenti, ciascuna delle quali richiede una quantità di risorse maggiore di $c*g(n)$ (per un'opportuna costante c). In altre parole, sapere che un algoritmo ha costo $\Omega(g(n))$ non è sufficiente per affermare che, per ogni istanza del problema di dimensione n , il costo di esecuzione è almeno $c*g(n)$.

La ragione di questa differenza è che capita frequentemente di avere algoritmi che si comportano efficientemente in molti casi ma non in tutti. La definizione 2 non considera i casi "facili", ma fa riferimento ai casi più costosi per stabilire la delimitazione inferiore del problema.

Chiaramente la delimitazione inferiore al costo di un algoritmo non può essere rappresentata da una funzione che, al crescere delle dimensioni dell'input, cresce più velocemente della delimitazione superiore della complessità. Abbiamo una valutazione esatta del costo di un algoritmo quando le due delimitazioni coincidono.

Nel seguito quando non è specificato espressamente per delimitazione della complessità si intende la delimitazione superiore.

Come abbiamo già osservato nel paragrafo precedente ignorare la costante moltiplicativa porta ad una valutazione approssimata perché permette di asserire che il costo di un programma è $O(n)$, cioè lineare, quando il numero delle istruzioni eseguite è, ad esempio, $2n$ o $1000n$. Tuttavia l'ipotesi semplificativa permette di valutare, almeno in modo approssimato, l'efficienza di un programma o di un algoritmo e di effettuare confronti di efficienza fra due programmi o algoritmi.

Per capire meglio l'ultima affermazione consideriamo la fig. 2, che esprime il costo di esecuzione di 4 programmi che risolvono lo stesso problema e aventi complessità $100n$, $10n^{**}2$, $(n^{**}3)/2$ e $2^{**}n$, rispettivamente. Se supponiamo di utilizzare un elaboratore in grado di compiere un milione di operazioni al secondo, allora i diversi programmi risolvono in un microsecondo istanze del problema aventi dimensioni comparabili (circa 10). Al crescere delle dimensioni del problema questo non è più vero. Infatti i tempi necessari per eseguire istanze del problema aventi dimensioni 30 risultano molto diversi; la differenza risulta ancora più drammatica per dimensioni maggiori, come è illustrato dalla fig. 2.a. Ad esempio, il tempo richiesto per risolvere un'istanza di dimensione 90 con l'algoritmo esponenziale richiede un tempo di calcolo maggiore dell'età della terra.

La fig. 2.b evidenzia le dimensioni massime dei problemi che sono risolubili in un secondo, un minuto, un'ora o un giorno. Ad esempio, se si utilizza il programma avente costo $100n$, è possibile risolvere, in un minuto o in un'ora, istanze del problema aventi dimensione 600000 e 36000000, rispettivamente; con il programma di complessità $2^{**}n$ in un minuto o in un'ora possiamo risolvere istanze aventi dimensioni 25 e 31, rispettivamente.

<i>Complessità del programma</i>	<i>Dimensioni dell'input</i>		
	10	30	60
$100 n$	1 millisec.	3 millisec.	6 millisec.
$10 n^{**2}$	1 millisec.	9 millisec.	36 millisec.
n^{**3}	1 millisec.	27 millisec.	2.1 secondi
$2^{** n}$	1 millisecond.	18minuti	366 secoli

Fig. 2.a - La tabella indica il tempo di calcolo in funzione delle dimensioni del problema

<i>Complessità del programma</i>	<i>Tempo impiegato</i>			
	1 sec.	1 min.	1 ora	1 giorno
$100 n$	10^4	$6 \cdot 10^5$	$36 \cdot 10^6$	$864 \cdot 10^6$
$10 n^{**2}$	316	2449	18970	92951
n^{**3}	100	392	1532	4420
$2^{** n}$	20	25	31	36

Fig. 2.b - La tabella indica le dimensioni massime del problema risolvibili in una data quantità di tempo

Fig. 2 - Confronto di diversi algoritmi con complessità diverse

2.2. Valutazione della complessità di un programma

La valutazione del costo di un programma o di un algoritmo richiede la determinazione di una delimitazione superiore ed una inferiore. In questo paragrafo presentiamo un insieme di regole che aiutano a trovare la delimitazione superiore della complessità.

Regola 1. Supponiamo che il programma sia composto di due parti P e Q da eseguire sequenzialmente e che i costi di P e Q siano $S(n)=O(f(n))$ e $T(n)=O(g(n))$. Allora il costo del programma è $O(\max(f(n), g(n)))$.

Per dimostrare la correttezza della regola osserviamo che in questo caso il costo complessivo del programma è, chiaramente, pari a

$$S(n) + T(n).$$

Inoltre si noti che

1. poiché $S(n) = O(f(n))$ allora esistono costanti a' , b' e n' tali che $S(n) < a'f(n) + b'$ per $n > n'$;
2. poiché $T(n) = O(g(n))$ allora esistono costanti a'' , b'' e n'' tali che $T(n) < a''g(n) + b''$ per $n > n''$.

Da queste due osservazioni deriva che, per $n > \max(n', n'')$

$$\begin{aligned} S(n) + T(n) &< a'f(n) + b' + a''g(n) + b'' \\ &< (a' + a'') \max(f(n), g(n)) + (b' + b'') \end{aligned}$$

Quindi, $O(S(n) + T(n))$ è proprio $O(\max(f(n), g(n)))$.

La regola precedente può essere generalizzata nel seguente modo.

Regola 1'. Se un programma è composto da una successione finita di parti $P(1)$, $P(2)$, ... $P(k)$ da eseguire sequenzialmente allora la complessità del programma è pari alla complessità del passo più costoso.

Sia dato, ad esempio, un programma composto da 3 parti P, Q, R che vengono eseguite sequenzialmente. Se P, Q e R hanno, rispettivamente, complessità $O(n^{**}2)$, $O(n^{**}3)$ e $O(n \log n)$, allora il costo complessivo del programma è pari a $O(n^{**}3)$.

Utilizzando la definizione 1 la regola 1 può essere così espressa:

Date due funzioni $f(n)$ e $g(n)$, se $f(n)=O(g(n))$, allora

$$O(f(n) + g(n)) = O(g(n)).$$

La seconda regola che presentiamo permette di valutare il costo del programma quando esso richiede più volte l'esecuzione di un insieme di istruzioni o l'attivazione di una procedura.

Regola 2. Supponiamo che un programma richieda per k volte l'esecuzione di una istruzione composta o l'attivazione di una procedura, e sia $f_i(n)$ il costo relativo all'esecuzione i-esima, $i=1, 2, \dots, k$. Il costo complessivo del programma è pari a

$$O(\sum_i f_i(n))$$

La dimostrazione della validità di quest'ultima regola è simile a quanto fatto precedentemente e viene omessa.

La regola 2 è molto utile nella valutazione di programmi ricorsivi. In questo caso $f_i(n)$ rappresenta il costo della i -esima attivazione della procedura. In molti casi il valore che denota il numero di ripetizioni è anch'esso una funzione delle dimensioni del problema, abbiamo cioè $k=k(n)$. Non è difficile vedere che anche in questo caso la regola 2 risulta valida.

Un caso particolare di applicazione della regola 2 è quello in cui il programma richiede per $k(n)$ volte l'esecuzione di un'istruzione composta o l'attivazione di una procedura avente ogni volta lo stesso costo $f(n)$; questo è il caso in cui le funzioni $f_i(n)$ sono tutte uguali a $f(n)$. In questo caso è facile osservare che il costo del programma è pari a $O(k(n)f(n))$.

2.3. Istruzione dominante

Introduciamo ora il concetto di istruzione dominante che permette, in molti casi, di semplificare in modo drastico la valutazione della complessità di un programma.

Definizione 3. Sia dato un programma o un algoritmo P il cui costo di esecuzione è $t(n)$. Una istruzione di P si dice istruzione o operazione dominante quando, per ogni intero n , essa viene eseguita, nel caso peggiore di *input* avente dimensione n , un numero di volte $d(n)$ che verifica la seguente condizione

$$t(n) < a d(n) + b$$

per opportune costanti a e b .

In altre parole, un'istruzione dominante viene eseguita un numero di volte proporzionale al costo di esecuzione di tutto l'algoritmo. È importante osservare che in un programma, più istruzioni possono essere dominanti, ma può anche accadere che il programma non contenga affatto istruzioni dominanti.

Regola 3. Supponiamo che un programma contenga un'istruzione dominante che, nel caso peggiore di *input* di dimensione n , viene eseguita $d(n)$ volte. La delimitazione superiore alla complessità del programma è $O(d(n))$.

La dimostrazione della correttezza della regola 3 è lasciata al lettore. Ci limitiamo ad osservare che questa semplificazione è possibile solo perché nella valutazione della complessità si trascurano costanti moltiplicative e termini additivi di ordine inferiore.

Per individuare un'istruzione dominante è sufficiente, in molti casi, esaminare le operazioni che sono contenute nei cicli più interni del programma. Ad esempio, nel caso della ricerca esaustiva considerata nell'esempio precedente, è possibile valutare il costo del programma di fig.1 osservando che l'istruzione {3} è dominante. Infatti essa viene eseguita, nel caso peggiore, n volte; questo è sufficiente per dire che il programma ha costo lineare. Osserviamo inoltre che il test {4} del ciclo **repeat until** viene eseguito nel caso peggiore n volte e, quindi, rappresenta un'altra istruzione dominante.

2.4. Delimitazioni alla complessità di un problema

La notazione O esprime una delimitazione superiore alla complessità di un programma che risolve un particolare problema. Se un problema ha diversi algoritmi di soluzione con complessità diversa è naturale utilizzare l'algoritmo più efficiente per esprimere la complessità del problema.

Definizione 4. Un problema ha una *delimitazione superiore* $O(f(n))$ alla sua complessità, se esiste almeno un algoritmo di soluzione che ha complessità $O(f(n))$.

La ricerca di algoritmi sempre più efficienti ha permesso, in alcuni casi, di abbassare la complessità del problema da una funzione di tipo esponenziale ad una di tipo lineare. Può accadere, però, che questi miglioramenti non siano sempre possibili e che la delimitazione superiore alla complessità di un problema non possa essere abbassata oltre una certa soglia. Siamo perciò interessati a caratterizzare la delimitazione inferiore alla complessità del problema o, in altre parole, la complessità intrinseca del problema.

Definizione 5. Un problema ha *delimitazione inferiore* $\Omega(g(n))$ alla sua complessità, se è possibile provare che ogni algoritmo di soluzione deve avere costo $\Omega(g(n))$.

Dimostrare che un problema ha una delimitazione inferiore $\Omega(g(n))$ alla sua complessità vuole dire che *ogni algoritmo di soluzione* per quel problema deve avere complessità almeno $\Omega(g(n))$. Si noti che la definizione precedente fa riferimento non solo agli algoritmi noti, ma ad ogni possibile algoritmo progettato o ancora da progettare.

Sottolineiamo la differenza tra le due definizioni di delimitazione superiore ed inferiore alla complessità di un problema. Nella definizione 4 un problema ha costo $O(f(n))$ se esiste un algoritmo che, per ogni n e per ogni input di

dimensione n , impiega una quantità di risorse minore di $f(n)$. Invece nella definizione 5 un problema ha costo $\Omega(g(n))$ se per ogni algoritmo di soluzione esiste una sequenza infinita di istanze del problema aventi dimensioni crescenti, ciascuna delle quali richiede una quantità di risorse maggiore di $g(n)$.

Pertanto, se un problema ha costo $\Omega(g(n))$ non è detto che, per ogni istanza del problema di dimensione n , il costo di esecuzione di ogni algoritmo di risoluzione sia almeno $c g(n)$ (per un'opportuna costante c). Ad esempio, per stabilire che un problema ha una delimitazione $\Omega(n^{**2})$ non è necessario dimostrare che ogni algoritmo di soluzione ha costo almeno $\Omega(n^{**2})$ per ogni istanza di dimensione n ; è, invece, sufficiente mostrare che, per ogni algoritmo di soluzione esiste una sequenza infinita di istanze "difficili" di dimensione crescente che richiedono una quantità di risorse pari almeno a $\Omega(n^{**2})$.

L'ultima definizione di questa sezione è quella di *algoritmo ottimale*. Intuitivamente un algoritmo è ottimale quando non può esistere un altro algoritmo avente complessità inferiore (a meno di un fattore moltiplicativo costante).

Definizione 6. Un algoritmo di soluzione di un problema P è *ottimale* quando l'algoritmo ha complessità $O(f(n))$ e la delimitazione inferiore alla complessità del problema P è $\Omega(f(n))$. In questo caso la notazione $\theta(f(n))$ denota la complessità del problema.

Esempio. Consideriamo il problema del calcolo del valore in un punto di un polinomio avente grado n . Supponiamo che i valori dei coefficienti del polinomio siano memorizzati in un **array** il cui tipo è così definito

```
type vett= array [0..n] of real
```

La componente i -esima dell'array contiene il valore del coefficiente di x^{**i} e la componente zero dell'array contiene il valore del termine noto del polinomio. In questo caso la dimensione dell'input è pari a n , grado del polinomio.

La prima soluzione che consideriamo è data in fig. 3. La valutazione del costo di esecuzione del programma precedente è semplificata se osserviamo che l'istruzione 6, posta all'interno di due cicli, è un'istruzione dominante. Quando la variabile i vale 1 il ciclo più interno viene eseguito una sola volta. Successivamente la variabile i assume il valore due e il ciclo più interno viene eseguito due volte. Analogamente quando la variabile i vale 3, 4, ... il ciclo più interno viene eseguito 3, 4, ... volte. Pertanto l'istruzione 6 viene eseguita

$$1 + 2 + 3 + \dots + (n - 1) + n$$

```

function pol_1 (var a : vett): real;
var i, j : integer;
x, y, val : real;
{x contiene il valore del punto in cui si vuole valutare il polinomio, y il valore
di una potenza di x, val il valore del polinomio nel punto x}

begin
{1} val := a[0];
{2} readln (x);
{3} for i := 1 to n
    do begin
{4}         y := 1;
{5}         for j := 1 to i
{6}             do y := y * x;
{7}             val := val + (a[i] * y)
        end;
{8} pol_1 := val
end;

```

Fig. 3 - Programma quadratico per la valutazione di un polinomio in un punto

volte. La somma precedente è una serie aritmetica il cui valore è $n(n+1)/2$. Pertanto il costo di esecuzione del programma è $O(n^2)$.

La seconda soluzione che consideriamo evita di calcolare ogni volta la potenza di x elevato a i a partire da zero, ma utilizza i calcoli precedentemente fatti. A questo scopo è sufficiente memorizzare il valore di y e poi aggiornarlo all'interno del ciclo principale. La funzione è data in fig. 4.

È facile vedere che le istruzioni 5 e 6, contenute all'interno del ciclo, sono le istruzioni dominanti e vengono eseguite n volte ciascuna. Pertanto la complessità della procedura pol_2 è $O(n)$. Il programma che abbiamo ottenuto è quindi più efficiente del programma pol_1 . Osserviamo inoltre che l'algoritmo pol_2 è ottimale perché è possibile dimostrare che ogni algoritmo per il calcolo del valore in un punto di un polinomio di grado n , deve compiere almeno $O(n)$ operazioni.

```

function pol_2 (var a : vett): real;
var i : integer;
    x, y, val : real;
{x contiene il valore del punto in cui si vuole valutare il polinomio, y il valore
di una potenza di x, val il valore del polinomio nel punto x}
begin
{1}   val := a[0];
{2}   readln (x);
{3}   y := 1;
{4}   for i := 1 to n
      do begin
{5}         y := y * x;
{6}         val := val + (a[i] * y)
      end;
{7}   pol_2 := val
end;

```

Fig. 4 - Programma lineare per la valutazione di un polinomio in un punto

3. LA GESTIONE DI UNA TAVOLA

In questo paragrafo consideriamo il problema della gestione di una tavola che è stato già studiato nel capitolo dedicato alle strutture di dati.

Consideriamo una tavola di elementi così definiti

```

type elem = record
    inf : tipo_informazione;
    chiave : integer
  end

```

La tavola è composta da *n* elementi, dove *n* rappresenta la dimensione dell'input. Nel seguito del paragrafo analizziamo la complessità di diversi metodi per la gestione di una tavola che sono stati introdotti nel cap. 3.

3.1. Gestione sequenziale

Il metodo di rappresentazione della tavola che dapprima consideriamo è quello mediante un array di record: ogni componente dell'array memorizza un elemento e gli elementi sono posti in ordine qualunque. La ricerca di un elemento nella tavola avviene con l'algoritmo di ricerca esaustiva, che abbiamo già analizzato nel paragrafo precedente, dimostrando che ha complessità $O(n)$. Osserviamo che $O(n)$ è anche il costo degli algoritmi di inserimento e di eliminazione di un elemento dalla tavola.

Nel caso che la tavola sia realizzata utilizzando record e puntatori, allora è facile vedere che il costo delle operazioni di ricerca e di eliminazione è ancora $O(n)$, perché nel caso peggiore sono necessari n confronti. Per la valutazione del costo dell'operazione di inserimento si distinguono due casi. Se è noto che l'elemento da inserire non è già presente nella tavola, allora è possibile inserirlo all'inizio della lista con un numero costante di operazioni. Nel caso che non sia noto se l'elemento è già presente, è necessario effettuare prima dell'inserimento una ricerca preliminare di costo $O(n)$.

3.2. Gestione sequenziale ordinata: la ricerca binaria

Come abbiamo visto nel cap. 3, par. 8.1, e nel cap. 6, par. 2.2, il metodo della ricerca binaria può essere utilizzato per ricercare elementi in una tavola ordinata.

La complessità della ricerca binaria dipende dal particolare elemento da individuare: se ricerchiamo l'elemento mediano allora è sufficiente un solo confronto, mentre in tutti gli altri casi dobbiamo eseguirne un numero maggiore. Non è difficile convincersi che la complessità dell'algoritmo è data dal numero di confronti effettuato e che il caso peggiore si verifica quando ricerchiamo un elemento non presente nella tavola.

Per valutare la complessità della procedura di ricerca binaria nel caso peggiore osserviamo che, dopo un confronto, dobbiamo proseguire la ricerca su metà degli elementi. Inoltre ogni successivo confronto permette di dimezzare ulteriormente la dimensione della tavola su cui proseguire la ricerca. Dopo h confronti la dimensione della tavola su cui dobbiamo continuare la ricerca è

$$n / (2^{**h})$$

Pertanto, nel caso peggiore, il numero di confronti è il più piccolo intero m che soddisfa la seguente relazione

$$2^{**m} \geq n$$

Eseguendo il logaritmo del termine destro e sinistro della espressione precedente, otteniamo che non appena il numero di confronti m è maggiore di $\log n$ (dove il logaritmo è in base 2) la ricerca termina. Poiché il confronto di due chiavi è un'operazione dominante, l'analisi precedente ci permette di ottenere il seguente teorema che esprime la complessità della ricerca binaria.

Teorema (Complessità della ricerca binaria). L'algoritmo di ricerca binaria ha complessità $O(\log n)$.

Per completare la nostra analisi osserviamo che per inserire un nuovo elemento in una tavola ordinata è necessario:

1. ricercare la posizione in cui inserire il nuovo elemento;
2. inserire il nuovo elemento al suo posto.

Non è difficile modificare la procedura di ricerca per risolvere il passo 1 con complessità $O(\log n)$. Invece il passo 2 richiede di spostare di una posizione tutti gli elementi che seguono l'elemento dato. Nel caso che l'elemento da inserire sia il primo elemento della tavola si richiedono n spostamenti. Quindi, la complessità del passo 2 è $O(n)$.

Il caso dell'eliminazione è analogo; in questo caso è necessario spostare tutti gli elementi che seguono l'elemento eliminato. Se si elimina il primo elemento della tavola allora bisogna effettuare n spostamenti; quindi, l'eliminazione di un elemento da una tavola ordinata ha, nel caso peggiore, complessità $O(n)$.

3.3. Alberi binari di ricerca

Se utilizziamo un albero binario di ricerca per realizzare una tavola allora la complessità dell'algoritmo di ricerca è, nel caso peggiore, pari alla profondità dell'albero (lunghezza del cammino più lungo fra la radice dell'albero e una foglia). Quindi essa dipende dalla particolare struttura dell'albero. Per un albero avente n nodi, la situazione peggiore è quella illustrata in fig. 5.a. In questo caso la complessità dell'algoritmo è $O(n)$ perché n sono i nodi da visitare prima di esaminare la foglia dell'albero.

Il caso migliore dell'algoritmo è quello di fig. 5.b; nella figura è rappresentato un albero binario completamente bilanciato in cui tutti i cammini fra la radice e le foglie differiscono al massimo di uno. È possibile dimostrare che la profondità di un albero binario completamente bilanciato con n nodi è $O(\log n)$ (dove il logaritmo è in base 2); questo implica che la ricerca in questo caso ha

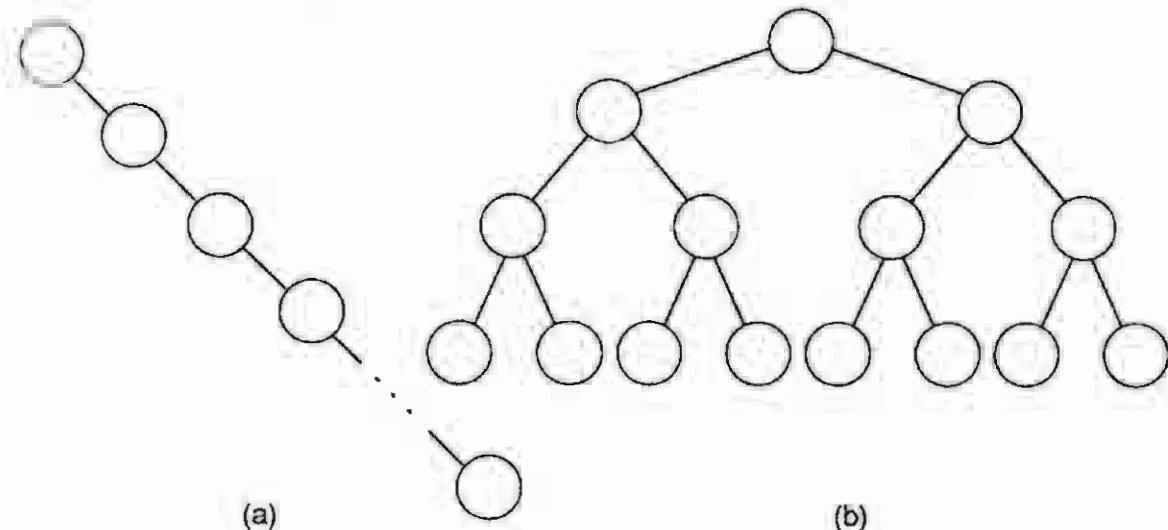


Fig. 5 - Caso peggiore e caso migliore di albero binario di ricerca

complessità $O(\log n)$.

Quindi per diminuire il costo dell'operazione di ricerca è importante cercare di ottenere alberi bilanciati o, se questo non è possibile, alberi in cui la lunghezza del cammino più lungo fra la radice ed una foglia è minore di ($c \log n$), dove c è un'opportuna costante. Per questo scopo sono stati studiati diversi algoritmi che permettono di mantenere quest'ultima proprietà anche in presenza di inserimenti e di eliminazioni. Lo studio di tali algoritmi va oltre gli scopi di questo testo.

3.4. Tavole Hash

La valutazione della complessità della ricerca in una tavola hash dipende dal numero di collisioni presenti nella tavola e da come queste vengono trattate.

Supponiamo per esempio di utilizzare il metodo delle liste di trabocco. Abbiamo visto nel cap. 3 che questo metodo associa una lista a ciascuna chiave che provoca una collisione. In ciascuna lista vengono inseriti tutti gli elementi che hanno lo stesso indirizzo hash.

Per valutare la complessità del metodo, supponiamo di avere una tavola con dieci elementi che memorizza chiavi numeriche. Se si utilizza come funzione hash l'ultima cifra della chiave dopo aver inserito nella tavola gli elementi di chiave

21, 31, 41, 51, 61,

abbiamo che tutti gli elementi della tavola sono posti in un'unica lista. È facile rendersi conto che se ricerchiamo la chiave memorizzata nell'ultima posizione dobbiamo scandire tutta la lista di trabocco che in questo caso contiene tutti gli elementi della tavola.

L'esempio precedente può essere generalizzato al caso di tavole con un numero qualunque di elementi. In questo caso si ottiene una tavola di n elementi che hanno tutti lo stesso valore della funzione hash. Quindi, la ricerca di un elemento può richiedere la scansione di tutta la lista di trabocco e la complessità della ricerca è nel caso peggiore, $O(n)$.

Considerazioni analoghe valgono anche per altri metodi di gestione delle collisioni proposti nel cap. 3.

Osserviamo che la valutazione del costo della ricerca nel caso peggiore fa riferimento al caso altamente improbabile in cui tutte le chiavi abbiano lo stesso indirizzo hash. Infatti è possibile mostrare che nel caso medio il numero di confronti richiesto risulta molto minore.

4. IL PROBLEMA DELL'ORDINAMENTO

In questo paragrafo consideriamo il problema dell'ordinamento, che ha un'importanza fondamentale in moltissime applicazioni. Lo studio di questo problema è importante anche perché permette di mostrare algoritmi che raggiungono il medesimo scopo utilizzando metodi molto diversi. I diversi algoritmi di soluzione presentati verranno confrontati dal punto di vista dell'efficienza. In questo modo sarà possibile osservare che miglioramenti sostanziali nell'efficienza possono essere ottenuti utilizzando algoritmi sofisticati al posto di algoritmi semplici ed intuitivi.

Nel seguito consideriamo solo metodi di ordinamento cosiddetti interni, in cui gli elementi da ordinare risiedono nella memoria centrale dell'elaboratore (algoritmi che ordinano insiemi memorizzati su memorie di massa sono detti metodi di ordinamento esterni). Assumiamo, pertanto, di avere in memoria un array $vett$ di tipo *vett* definito nel modo seguente:

```
type elem = record
    inf : tipo_informazione;
    chiave : integer;
end;

vett =array [1..n] of elem;
```

e di dover ordinare le componenti del vettore secondo l'ordine crescente del campo chiave.

4.1. Ordinamento per selezione (Selectionsort)

Abbiamo presentato l'algoritmo di ordinamento per selezione nel cap. 5 par. 2.2. In questo paragrafo ci limitiamo ad analizzare la complessità del programma di fig. 6 che implementa l'algoritmo di ordinamento per selezione; esso utilizza la procedura *scambia* per effettuare lo scambio dei valori fra due componenti dell'array.

```
procedure ordinamento_per_selezione ( var z : vett; n : integer);
var i, j, imin : integer;
begin
for i := 1 to n-1
do begin
    {ricerca della componente minima fra z[i] e z[n]}
    imin := i;
    for j := i + 1 to n
        do if z[j].chiave < z[imin].chiave
            then imin := j;
    { scambia z[imin] e z[i]}
    scambia(z[i],z[imin])
end
end;
```

Fig. 6 - Algoritmo di ordinamento per selezione

Non è difficile dimostrare che la complessità dell'algoritmo è $O(n^2)$. Infatti l'istruzione condizionale

```
if z[j] < z[imin]
    then imin := j
```

è un'istruzione dominante. Essa viene eseguita $n-1$ volte quando i vale 1; alla generica iterazione i del ciclo **for** più esterno essa viene eseguita $n-i$ volte. Pertanto il numero totale di esecuzioni dell'istruzione dominante è

$$(n-1) + (n-2) + (n-3) \dots + 3 + 2 + 1 = n(n-1)/2$$

che è proprio $O(n^{**}2)$.

Pertanto si può enunciare il seguente teorema.

Teorema (ordinamento per selezione). La complessità asintotica dell'algoritmo di ordinamento per selezione è $O(n^{**}2)$.

Osserviamo che l'algoritmo di ordinamento per selezione è composto da due cicli **for** che vengono eseguiti un numero di volte che non dipende dai particolari dati di ingresso. Quindi l'algoritmo ha lo stesso costo per ogni possibile configurazione dei dati di ingresso; in particolare notiamo che esso impiega $O(n^{**}2)$ operazioni anche se l'array è già completamente ordinato.



4.2. Ordinamento a bolle (Bubblesort)

L'algoritmo di ordinamento a bolle si basa sul principio che in un array ordinato presi comunque due elementi adiacenti $z[i]$ e $z[i+1]$ abbiamo che il primo è minore o uguale del secondo. L'algoritmo confronta ripetutamente coppie adiacenti che vengono scambiate se non rispettano l'ordinamento. Eseguendo opportunamente questa operazione si ottiene l'array ordinato.

La prima iterazione inizia con il confronto delle chiavi di $z[n]$ e $z[n-1]$. Se $z[n].chiave < z[n-1].chiave$, allora i due valori non rispettano l'ordinamento e vengono scambiati; se $z[n].chiave \geq z[n-1].chiave$ allora i due elementi rispettano l'ordinamento crescente e non si fa nulla.

La prima iterazione dell'algoritmo prosegue con l'esame delle chiavi di $z[n-1]$ e $z[n-2]$; se $z[n-1].chiave < z[n-2].chiave$ i due valori vengono scambiati. Quindi si prosegue confrontando ed eventualmente scambiando le componenti $z[n-2]$ e $z[n-3]$, poi $z[n-3]$ e $z[n-4]$, e così via fino a quando non si esaminano $z[2]$ e $z[1]$. Termina in questo modo la prima fase dell'algoritmo; è facile vedere che a questo punto la più piccola componente dell'array si trova nella prima posizione.

Le iterazioni successive sono analoghe. All'inizio della iterazione i le prime $(i-1)$ posizioni dell'array contengono i primi $(i-1)$ elementi ordinati. Quindi, si esaminano le chiavi $z[n]$ e $z[n-1]$; se le chiavi non verificano l'ordinamento desiderato, i due elementi vengono scambiati; si prosegue confrontando $z[n-1]$ e $z[n-2]$, $z[n-2]$ e $z[n-3]$ ecc., fino a quando non si confrontano (ed eventualmente si scambiano) $z[i+1]$ e $z[i]$. A questo punto termina l'iterazione i -esima e

```

procedure ordinamento_a_bolle ( var z : vett; n : integer);
var i, j : integer;
    fine : boolean;
begin
  i := 1;
repeat
  i := i + 1;
  fine := true;
  for j := n downto i
  do if (z[j].chiave < z[j-1].chiave)
    then begin
      {scambia z[j] e z[j-1]}
      scambia (z[j],z[j-1]);
      {aggiorna la variabile fine}
      fine := false
    end
until fine or (i = n)
end;

```

Fig. 7 - Algoritmo di ordinamento a bolle

l'algoritmo ha posto l' i -esimo più piccolo elemento dell'array in posizione i . Proseguendo in questo modo dopo $(n-1)$ iterazioni l'array è ordinato.

Osserviamo che non sempre sono necessarie $(n-1)$ iterazioni. Infatti, osserviamo che, se durante una generica iterazione non avviene nessuno scambio, allora il vettore è stato ordinato e l'algoritmo può quindi terminare.

In fig. 7 viene data la procedura che realizza l'algoritmo, utilizzando la procedura *scambia* per effettuare lo scambio di due elementi.

Esempio. Eseguiamo l'algoritmo di ordinamento a bolle supponendo che l'array di chiavi da ordinare sia dato in fig 8.a.

All'inizio si confrontano fra loro la quarta e la terza componente. Poiché $z[4].chiave$ è minore di $z[3].chiave$ si scambiano fra loro i due elementi ottenendo la situazione di fig. 8.b. Successivamente si confrontano e si scambiano $z[3]$ e $z[2]$ (vedi fig. 8.c), poi $z[2]$ e $z[1]$ ottenendo alla fine la situazione di fig 8.d.

Alla fine della prima iterazione l'elemento più piccolo è salito dal fondo fino alla prima posizione. Se immaginiamo di associare a ciascun elemento del

38	38	$38 \leftarrow j-1$	25
63	$63 \leftarrow j-1$	$25 \leftarrow j$	38
$41 \leftarrow j-1$	$25 \leftarrow j$	63	63
$25 \leftarrow j$	41	41	41
a	b	c	d

Fig. 8 - Esecuzione dell'algoritmo di ordinamento a bolle

vettore un peso pari al suo valore, allora, si può osservare che l'elemento più leggero è risalito fino in cima come una bolla; da questa similitudine deriva il nome di ordinamento a bolle dell'algoritmo.

Durante l'iterazione successiva otteniamo via via gli array di figg. 9.a, 9.b e 9.c. Si noti in particolare che durante la terza iterazione, si confrontano le chiavi di $z[4]$ e $z[3]$, ma non avverrà nessuno scambio, perché l'array è stato ordinato.

Osserviamo che i due algoritmi di ordinamento per selezione e di ordinamen-

25	25	25
38	$38 \leftarrow j-1$	38
$63 \leftarrow j-1$	$41 \leftarrow j$	41
$41 \leftarrow j$	63	63
a	b	c

Fig. 9 - Successive iterazioni dell'algoritmo di ordinamento a bolle

to a bolle utilizzano due diverse, ma equivalenti, definizioni di ordinamento. Nel caso del primo algoritmo si definisce ordinato un array se, per ogni valore dell'indice i , la componente di chiave i -esima si trova in posizione i . Invece, nel caso dell'algoritmo di ordinamento a bolle, come è già stato detto, si definisce ordinato un array quando, per ogni indice i , le chiavi delle componenti i e $i+1$ risultano ordinate.

La complessità dell'algoritmo di ordinamento a bolle dipende dai valori dei dati di ingresso. Supponiamo che l'array di ingresso sia già ordinato in senso crescente. In questo caso l'algoritmo compie una iterazione durante la quale non si effettua nessuno scambio, ma che serve a verificare che l'array sia effettivamente ordinato in senso crescente. Pertanto, in questo caso il costo di esecuzione dell'algoritmo è $O(n)$. Non è difficile dimostrare che il caso peggiore è quello in cui l'array è ordinato in senso decrescente. In questo caso

6	1	1	1	1	1
5	6	2	2	2	2
4	5	6	3	3	3
3	4	5	6	4	4
2	3	4	5	6	5
1	2	3	4	5	6
a	b	c	d	e	f

Fig. 10 - Comportamento più sfavorevole dell'algoritmo di ordinamento a bolle

l'algoritmo richiede $n-1$ fasi ed ha complessità $O(n^{**}2)$. Consideriamo infatti la situazione di fig. 10.a.

In fig. 10.b è dato l'array alla fine della prima fase, durante la quale l'elemento con chiave minore è posto in prima posizione. Nelle figg. 10.c, 10.d, 10.e, 10.f viene dato l'array alla fine delle successive iterazioni. Chiaramente sono necessarie $n-1$ iterazioni perché alla fine di ciascuna di esse un solo elemento del vettore è stato posto nella posizione finale. Il numero dei confronti è $n-1$ durante la prima iterazione $n-2$ durante la seconda iterazione e $n-i$, durante la i -esima. Pertanto il numero totale di confronti è

$$(n-1) + (n-2) + \dots + 3 + 2 + 1 = n(n-1)/2$$

Pertanto abbiamo dimostrato il seguente teorema.

Teorema (ordinamento a bolle). La complessità asintotica dell'algoritmo di ordinamento a bolle è $O(n^{**}2)$.

4.3. Ordinamento per fusione (Mergesort)

In questo paragrafo consideriamo un algoritmo di ordinamento avente complessità $O(n \log n)$, che perciò risulta migliore di quelli finora visti.

Prima di presentare l'algoritmo consideriamo il seguente problema di *fusion* (*merge*) di due array ordinati:

Dati due array di interi x e y ordinati in ordine crescente, con m componenti ciascuno, si desidera ottenere un unico array z ordinato costituito dai $2m$ elementi degli array x e y .

L'algoritmo di soluzione che verrà presentato ha complessità lineare ed

utilizza due variabili indici i e j che servono per scandire i due array x e y e che sono poste inizialmente a uno. Quindi si confrontano $x[i]$ e $y[j]$; i casi possibili sono:

1. se $x[i]$ risulta minore o uguale di $y[j]$ allora si pone $x[i]$ in prima posizione nel vettore z e si incrementa i di uno;
2. se $x[i]$ risulta maggiore di $y[j]$, allora si pone $y[j]$ in prima posizione nel vettore z e si incrementa j di uno.

Successivamente si confrontano nuovamente gli elementi $x[i]$ e $y[j]$ (si noti che uno dei due elementi è cambiato perché una variabile fra i e j è stata incrementata) e si pone nella seconda posizione dell'array z il più piccolo dei due. Si procede in questo modo fino a quando uno dei due vettori non è esaurito. A questo punto si pongono in z gli elementi rimanenti dell'array non ancora esaurito.

Il programma di fusione è dato in fig. 11.

Esempio. Supponiamo di applicare l'algoritmo di fusione ai due vettori di fig. 12.a.

Alla prima iterazione si confrontano 2 e 3 e si pone 2 in $z[1]$. Successivamente si incrementa i (vedi fig. 12.b). Dopo aver eseguito un nuovo confronto fra $x[i]$ e $y[j]$ poniamo $y[j]$ in $z[2]$ ottenendo la situazione di fig. 13.a.

Si prosegue in modo analogo fino a quando non si ottiene la situazione di fig. 13. b. A questo punto tutti gli elementi dell'array y sono stati considerati e, quindi, si deve completare z con le componenti rimaste di x ($x[4] = 9$ e $x[5] = 10$).

Per analizzare la complessità dell'algoritmo di fusione è sufficiente osservare che ogni volta che si inserisce un nuovo elemento in z (nel programma sono segnalati i punti con 1, 2, 3, 4) si eseguono un numero costante di operazioni. Dato che in z vengono inseriti $2m$ elementi, la complessità dell'algoritmo è $O(m)$.

Siamo ora pronti per vedere l'algoritmo di ordinamento per fusione. L'algoritmo è intrinsecamente ricorsivo e può essere così descritto: se l'array ha due elementi con un confronto ed, eventualmente uno scambio, si ottiene l'array ordinato. Se l'array ha più di due elementi lo si divide in due array aventi ciascuno la metà degli elementi, si ordinano i due array usando lo stesso metodo, e, quindi, si realizza la fusione dei due array ordinati in un unico array.

```

procedure fusione (var x, y : vett1; var z : vett2);
{ gli array x e y hanno m componenti ciascuno }
var i, j, k : integer;
{ la variabile i serve per scorrere l'array x, la variabile j l'array y e la
variabile k l'array z }

begin
i := 1; j := 1; k := 1;
while (i <= m) and (j <= m)
do begin
    if (x[i] <= y[j])
    then begin
        {1} z[k] := x[i]; i := i + 1
        end
    else begin
        {2} z[k] := y[j]; j := j + 1
        end;
    k := k + 1
    end;

{ a questo punto è stata completata la scansione di almeno uno dei due
array }
if (i <= m)
then { si completa z con elementi da x }
repeat
    {3} z[k] := x[i]; i := i + 1; k := k + 1
until (i > m)
else { si completa z con elementi da y }
repeat
    {4} z[k] := y[j]; j := j + 1; k := k + 1
until (j > m)
end;

```

Fig. 11 - Programma di fusione

ordinato.

In fig. 14 è dato lo schema dell'algoritmo.

Per passare dall'algoritmo ad un programma è necessario modificare la procedura *fusione* vista precedentemente affinché operi su un unico array z, e non su tre array diversi. In particolare la procedura di fusione *merge* opera su

x	y	z
$i \rightarrow 2$	$3 \leftarrow j$	
6	4	
9	5	
10	8	

Fig. 12.a

x	y	z
2	$3 \leftarrow j$	2
$i \rightarrow 6$	4	
9	5	
10	8	

Fig. 12.b

Fig. 12 - Fasi iniziali dell'algoritmo di fusione

x	y	z
2	3	2
$i \rightarrow 6$	$4 \leftarrow j$	3
9	5	
10	8	

Fig. 13.a

x	y	z
2	3	2
6	4	3
$i \rightarrow 9$	5	4
10	8	5
	$\leftarrow j$	6
		8

Fig. 13.b

Fig. 13 - Fasi finali dell'algoritmo di fusione

Schema dell'algoritmo di ordinamento per fusione

{L'algoritmo ordina la parte dell'array z compresa fra le componenti $z[i]$ e $z[j]$ }

```
begin
  if "l'array ha solo due componenti"
    then "ordina l'array confrontando ed eventualmente
          scambiando le due componenti"
  else begin
    "poni k pari a  $(i+j) \text{ div } 2$ ";
    "esegui l'algoritmo di mergesort su  $z[i,k]$ ";
    "esegui l'algoritmo di mergesort su  $z[k+1,j]$ ";
    "esegui la fusione dei due array ordinati"
  end
end;
```

Fig. 14 - Schema dell'algoritmo di ordinamento per fusione

un solo array, ha come parametri di ingresso tre indici inf , med e sup , tali che:

- le chiavi degli elementi dell'array di posizione compresa fra inf e med sono fra loro ordinate;
- le chiavi degli elementi dell'array di posizione compresa fra $(med + 1)$ e sup sono fra loro ordinate.

Al termine dell'esecuzione della procedura *merge* le chiavi degli elementi dell'array di posizione compresa fra inf e sup sono fra loro ordinate.

Esempio. Supponiamo di dover ordinare l'array di chiavi di fig. 16.a di 8 componenti.

Inizialmente viene attivata la procedura *msort* (1,8), la quale a sua volta attiva la procedura *msort* (1,4). Quindi la procedura *msort* (1,4) attiva *msort* (1,2) e *msort* (3,4); in questi ultimi due casi gli array hanno dimensione 2 e quindi vengono ordinati, ottenendo il vettore di fig. 16.b.

Una volta completata l'esecuzione di *msort* (1,2) e *msort* (3,4) si prosegue con la fusione dei due array ottenendo la situazione di fig 16.c. L'esecuzione continua attivando la procedura *msort* (5,8), la quale a sua volta attiva le procedure *msort* (5,6) e *msort* (7,8). In questi casi ambedue le dimensioni degli array sono pari a 2 e nonabbiamo altre attivazioni ricorsive. In tal modo si

```

procedure mergesort (var z: vett; n: integer);
procedure msort (i, j : integer);
{la procedura ordina le componenti del vettore z, variabile non locale,
comprese tra i e j}

var k : integer;

begin {msort}
  if (i + 1) = j
    then if z[i].chiave > z[j].chiave
      then scambia (z[i],z[j]);
  if (i + 1) < j
    then begin
      k := (i + j) div 2;
      msort (i,k);
      msort (k+1,j);
      merge (i,k,j)
    end
  end; {msort}

begin {mergesort}
  msort (1,n)
end; {mergesort}

```

Fig. 15 - Algoritmo di mergesort

ottiene la situazione di fig. 16.d.

Operando la fusione dei due sottovettori si completa l'ordinamento della seconda metà del vettore (vedi fig. 16.e). In questo modo applicando nuovamente l'algoritmo di merge si ottiene l'array ordinato. In fig. 17 è dato l'albero delle attivazioni di procedura.

Valutiamo ora il numero di attivazioni della procedura *msort*. Per semplicità consideriamo il caso in cui l'array ha un numero di componenti pari a una potenza di due, cioè $n = 2^{**}k$ per qualche k intero; osserviamo anche che $k = \log n$, dove il logaritmo è in base 2. In questo caso si possono raggruppare le attivazioni della procedura *msort* nei seguenti k gruppi:

- gruppo 1) 1 attivazione su un array di $2^{**}k$ componenti;
- gruppo 2) 2 attivazioni su due array con $2^{**}(k - 1)$ componenti ciascuno;
- gruppo 3) 4 attivazioni su quattro array con $2^{**}(k - 2)$ componenti ciascuno;

...

	23	
	5	
	7	
	9	
	45	
	21	
	2	
	75	
a		
	5	5
	23	7
	7	9
	9	23
	45	45
	21	21
	2	2
	75	75
b		c
	5	5
	7	7
	9	9
	23	23
	21	2
	45	21
	2	45
	75	75
d		e

Fig. 16 - Fasi dell'algoritmo di *mergesort*

- gruppo i) 2^{i-1} attivazioni su 2^{i-1} array con $2*(k-i)$ componenti ciascuno;
 - ...
 - gruppo k) $n/2$ attivazioni su $n/2$ array con 2 componenti ciascuno.
- È facile dimostrare che il numero totale delle attivazioni della procedura *msort* è $n - 1$.

Prima di fornire una dimostrazione formale della complessità del *mergesort*

La soluzione dell'equazione di ricorrenza è

$$T(n) = c n \log n + c n$$

in cui il logaritmo si intende in base 2.

Verifichiamo ora che la soluzione dell'equazione di ricorrenza è $O(n \log n)$. A questo proposito supponiamo ancora che n sia una potenza di due, cioè $n = 2^{**}k$ per qualche intero k . Applicando più volte la definizione dell'equazione di ricorrenza che stabilisce il costo dell'algoritmo si ottengono le seguenti uguaglianze:

$$\begin{aligned} T(n) &= 2 T(n/2) + c n \\ &= 2(2 T(n/4) + c n/2) + c n \\ &= (2^{**}2) T(n/4) + (2 c) n \\ &= (2^{**}2)(2 T(n/8) + c (n/4)) + (2 c) n \\ &= (2^{**}3) T(n/8) + (3 c) n \\ &= \dots\dots \\ &= (2^{**}k) T(n/(2^{**}k)) + (k c) n \end{aligned}$$

Poiché $n/(2^{**}k) = 1$, si ha che $T(n/(2^{**}k)) = T(1) = c$. Quindi si ottiene

$$T(n) = (2^{**}k)T(1) + (k c) n = c n + (\log n) c n = O(n \log n)$$

Lo stesso risultato lo si ottiene in modo analogo nel caso in cui n non sia una potenza di due.

Abbiamo pertanto dimostrato il seguente teorema che esprime la complessità dell'algoritmo di ordinamento per fusione.

Teorema (ordinamento per fusione). La complessità asintotica dell'algoritmo di ordinamento per fusione è $O(n \log n)$.

L'analisi effettuata indica che l'algoritmo *mergesort* è migliore, nel caso più sfavorevole, degli altri algoritmi visti. Osserviamo però che le ipotesi che stanno alla base della nostra analisi danno una valutazione approssimata: in particolare assumiamo che il costo di attivazione di una procedura sia zero. Questo non è vero dal punto di vista pratico e può dare valutazioni errate soprattutto nel caso di programmi ricorsivi. Ritorneremo su questo argomento nel par. 5 del capitolo; per ora ci limitiamo ad osservare che per rendere pratico l'algoritmo di ordinamento per fusione è necessario utilizzare un programma

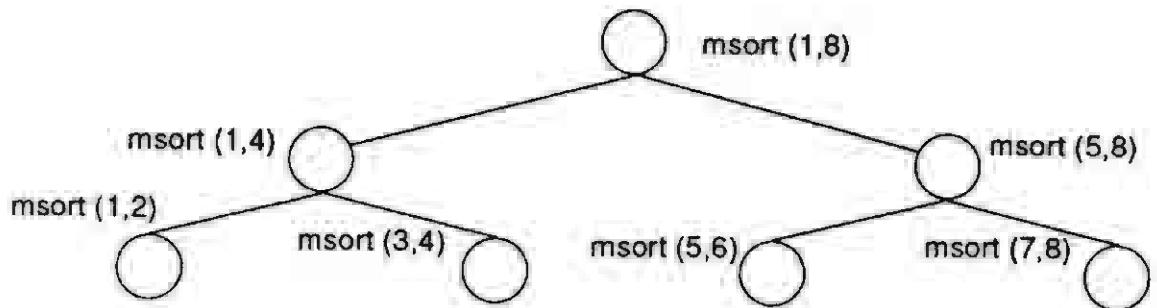


Fig. 17 - Albero delle attivazioni di procedura

notiamo che il costo totale di tutte le attivazioni della procedura fusione è $O(n \log n)$. Infatti, con riferimento alla divisione in gruppi precedentemente mostrata, si può vedere che il costo totale delle attivazioni di ciascun gruppo è $O(n)$. Essendo i gruppi in numero di $\log n$, otteniamo la tesi.

Per analizzare la complessità dell'algoritmo di *mergesort* distinguiamo due casi:

1. il vettore da ordinare ha uno o due elementi.
In questo caso la procedura *mergesort* ha come

Il vettore da ordinare ha più di due elementi.

2. il vettore da ordinare ha più di due elementi.

In questo caso sia n la dimensione del vettore da ordinare allora il costo di esecuzione della procedura $msort(1,n)$ è dato da

costo $msort(1, n) =$

costo $m\text{sort}(1, n/2) + \text{costo } m\text{sort}(n/2+1, n) + \text{costo fusione}$

Il costo della procedura di fusione è $c n$. Invece il costo di $msort(1, n/2)$ (o di $msort(n/2 + 1, n)$) è pari al costo di ordinare un array con la metà degli elementi. Se con $T(n)$ indichiamo il costo della procedura di ordinamento di un array con n elementi allora la formula precedente può essere riscritta nel seguente modo

$$T(n) = \underbrace{T(n/2)}_{\text{costo}} + \underbrace{T(n/2)}_{\text{costo}} + \underbrace{(c n)}_{\text{costo}} \text{ se } n >= 3$$

$$msort(1, n/2) \qquad msort(1+n/2, n) \qquad \text{fusion}$$

Pertanto il costo della procedura *msort* è quella funzione che soddisfa la seguente *equazione di ricorrenza (o induttiva)*

$$T(n) = \begin{cases} c \text{ costante} & \text{se } n < 3 \\ 2 T(n/2) + c n & \text{se } n \geq 3 \end{cases}$$

non ricorsivo.

Inoltre, ricordiamo che la nostra valutazione fa riferimento al caso più sfavorevole per dimensioni crescenti dell'input; quindi, può accadere che per piccoli valori di n o per particolari configurazioni dell'ingresso l'algoritmo di ordinamento a bolle o di ordinamento per inserzione siano più efficienti.

4.4. Ordinamento veloce (Quicksort)

In questo paragrafo consideriamo un algoritmo che, risulta essere uno degli algoritmi di ordinamento interno sperimentalmente più efficiente, pur non essendo il migliore dal punto di vista teorico.

L'idea principale del metodo consiste nello scegliere un elemento dell'array che viene detto *sentinella* (o "pivot") e suddividere l'array in due array, uno costituito dagli elementi con chiave minore dell'elemento di pivot e l'altro da quelli con chiave maggiore o uguale. Quindi, il metodo si applica ricorsivamente ai due array, fino a quando non si ottengono array con un solo elemento. Chiaramente a questo punto l'array iniziale è stato ordinato.

Schema di ordinamento veloce di un array $z[i..j]$, in cui l'indice può variare da i a j

begin

if "l'array ha due o più componenti"

then begin

"Scegli a caso un elemento dell'array (elemento di pivot); sia x la sua chiave";

"Dividi l'array in due parti $z[i..m]$, $z[m+1..j]$ tali che $z[i..m]$ contiene elementi con chiavi minori di x e $z[m+1..j]$ contiene elementi con chiavi maggiori o uguali di x ";

"Applica ricorsivamente il metodo a $z[i..m]$ e $z[m+1..j]$ "

end

end;

Fig. 18 - Schema algoritmo di ordinamento veloce

Per passare dallo schema di algoritmo ad un programma è necessario innanzitutto realizzare il passo di suddivisione dell'array. A questo scopo, se x è la chiave dell'elemento di pivot, si esaminano le componenti dell'array in ordine di indice decrescente a partire dall'ultima, fino a quando non si incontra un elemento con chiave minore o uguale di x ; quindi si esaminano le componenti dell'array a partire dalla prima in ordine di indice crescente fino a quando non si incontra un elemento con chiave maggiore a x oppure tutto l'array è stato esaminato. Si scambiano i due elementi che sono stati individuati e si ripete il procedimento fino a quando i due indici che permettono la scansione non si incontrano.

Consideriamo, ad esempio, di scegliere nel seguente array l'elemento 52 in quarta posizione come pivot

60	40	10	52	25	70	13	20	59
\wedge								
pivot								

In questo caso sono necessari due scambi (uno dei quali coinvolge l'elemento di pivot) che ci permettono di ottenere

20	40	10	13	25	70	52	60	59
----	----	----	----	----	----	----	----	----

Una volta realizzato il partizionamento rispetto ad un elemento di pivot è abbastanza semplice ottenere un algoritmo ricorsivo di ordinamento.

Nel programma di fig. 19, la procedura *quicksort* attiva la procedura *sort*; la procedura *sort* ordina le componenti di z avente indice compreso fra *sin* e *des*, i cui valori vengono passati come parametri valore al momento dell'attivazione di *sort*. Dato che ogni elemento può essere scelto come elemento di pivot, per semplicità viene scelto la componente con indice più piccola, $z[sin]$. Si noti, infine, che anche in questo caso lo scambio fra due componenti dell'array viene effettuato attivando la procedura *scambia* già utilizzata.

Nel seguito dimostriamo che il metodo di ordinamento veloce ha complessità $O(n \log n)$ nel caso peggiore.

Innanzitutto osserviamo che la procedura *sort*, se escludiamo le due attivazioni ricorsive, effettua un numero di operazioni minore di

$$c(des - sin + 1) + 3$$

(lineare, quindi, nel numero di elementi su cui la procedura opera).

Per dimostrare questa affermazione è sufficiente osservare che ciascuna delle istruzioni poste all'interno del ciclo **repeat** (le due istruzioni **while** e

```

procedure quicksort (var z : vett; n:integer);

procedure sort (sin,des:integer);
var i, j, x : integer;
begin
if (sin < des)
then begin
    i := sin; j := des + 1;
    x := z[sin].chiave;
    while (i < j)
        do begin
            repeat j := j-1 until (z[j].chiave <= x);
            repeat i := i+1 until (z[i].chiave > x) or (i=j);
            if i < j
                then scambia (z[i],z[j]);
            end ;
            if (sin<>j) then scambia (z[sin], z[j]);
            if (sin < j - 1) then sort (sin,j-1);
            if (des > i ) then sort (i, des)
        end
    end;

begin {quicksort}
    sort (1,n)
end;

```

Fig. 19 - Algoritmo di ordinamento veloce

l'istruzione **if...then**) incrementa almeno un indice tra *i* e *j*; quindi il ciclo **repeat....until** viene eseguito al massimo (*des-sin+1*) volte.

La valutazione della complessità dell'algoritmo dipende chiaramente dal numero di attivazioni ricorsive della procedura *sort*. La valutazione di questo numero dipende dalla scelta dell'elemento di pivot.

Per calcolare il costo complessivo dell'algoritmo si procede nel seguente modo: sia $Q(n)$ la funzione, per il momento ignota, che esprime il tempo necessario per ordinare *n* elementi con il metodo quicksort. Se $n = 1$ allora $Q(n)$ è pari ad una costante *c*. Se $n > 1$ supponiamo che la prima attivazione ricorsiva di *sort* richieda di ordinare un vettore con *t* elementi e la seconda un vettore con

$(n-t)$ elementi. Allora il costo delle due attivazioni sarà $Q(t)$ e $Q(n-t)$ rispettivamente. Pertanto otteniamo un'equazione di ricorrenza che esprime il costo dell'algoritmo di ordinamento veloce e che ammette soluzioni diverse al variare del parametro t .

$$(1) Q(n) = \begin{cases} c \text{ se } n = 1 \\ c n + Q(t) + Q(n-t) \text{ se } n > 1 \end{cases}$$

Il caso peggiore dell'algoritmo è quello in cui ogni volta viene scelto come elemento di pivot quello con chiave minore o maggiore. Se facciamo riferimento al programma di fig. 19, in cui l'elemento sentinella è il primo, questo avviene quando il vettore è già ordinato.

Per valutare il costo del programma in questo caso osserviamo che se la procedura *sort* sceglie come pivot l'elemento con chiave più piccola, essa divide un insieme di n elementi in due sottoinsiemi, uno avente $n-1$ elementi e l'altro un solo elemento. Se ad ogni attivazione ricorsiva della procedura *sort* si continua a scegliere come pivot l'elemento con chiave minore, allora ogni volta si divide l'insieme in due sottoinsiemi, uno avente un solo elemento e l'altro i rimanenti. In altre parole ogni attivazione di *sort*, permette di ottenere come nuova informazione sull'ordinamento, solo la posizione dell'elemento di pivot. La fig. 20 rappresenta l'albero delle attivazioni di procedura in questo caso.

Per valutare il costo nel caso di un array già ordinato, ricordiamo che, se si prescinde dal costo delle attivazioni ricorsive, la complessità della procedura *sort(sin,des)* è $O(\sin-\des+1)$; pertanto il costo delle diverse attivazioni di *sort* può essere così riassunto

costo di

$$\begin{aligned} \text{sort}(1, n) &= O(n) + (\text{costo di sort}(1, n-1)) \\ \text{sort}(1, n-1) &= O(n-1) + (\text{costo di sort}(1, n-2)) \\ \text{sort}(1, n-2) &= O(n-2) + (\text{costo di sort}(1, n-3)) \\ \text{sort}(1, n-3) &= O(n-3) + (\text{costo di sort}(1, n-4)) \\ \dots &= \dots \\ \dots &= \dots \\ \text{sort}(1, 2) &= O(2) + (\text{costo di sort}(1, 1)) \\ \text{sort}(1, 1) &= O(1) \end{aligned}$$

sommmando si ottiene che nel caso in cui l'elemento di pivot sia il più piccolo,

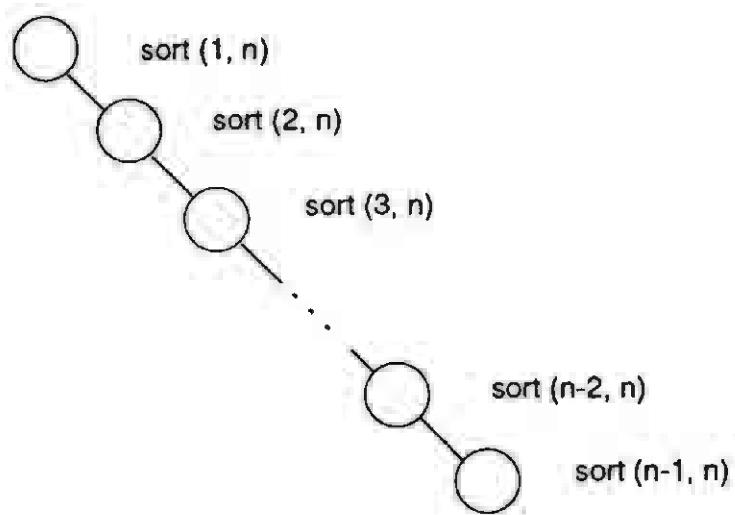


Fig. 20 - Albero delle attivazioni della procedura *quicksort* nel caso di array ordinato

la complessità dell'algoritmo è $O(n^{**}2)$.

È possibile ottenere lo stesso risultato utilizzando l'equazione di ricorrenza (1); poiché ogni volta viene scelto come pivot l'elemento con chiave minore, si può riscrivere l'equazione nel seguente modo (dove l'indice p sta ad indicare il caso peggiore):

$$Q_p(n) = c \cdot n + Q(1) + Q_p(n-1) = O(n) + Q_p(n-1).$$

In questo caso il costo complessivo dell'algoritmo è

$$\begin{aligned} Q_p(n) &= cn + c + Q_p(n-1) \\ &= cn + c + c(n-1) + c + Q_p(n-2) \\ &= cn + c + c(n-1) + c + c(n-2) + Q_p(n-3) \\ &\quad \dots\dots \\ &= cn + c(n-1) + c(n-2) + c(n-3) \dots + c + nc \\ &= O(n^{**}2) \end{aligned}$$

Abbiamo pertanto dimostrato che la procedura *quicksort* può eseguire $O(n^{**}2)$ operazioni. Non è difficile dimostrare che $O(n^{**}2)$ rappresenta anche una delimitazione superiore al costo dell'algoritmo.

Teorema (caso peggiore dell'ordinamento veloce). L'algoritmo di ordinamento veloce ha complessità $O(n^{**}2)$ nel caso peggiore.

Il teorema precedente dimostra che, nel caso peggiore, il comportamento dell'algoritmo di *quicksort* è pari a quello degli algoritmi meno sofisticati come, ad esempio, l'ordinamento per inserzione e l'ordinamento a bolle. L'analisi del caso peggiore, pertanto, non permette di spiegare il buon comportamento pratico del *quicksort*. Per spiegare come mai il *quicksort* risulti così efficiente in pratica, è necessario analizzare in modo diverso la complessità facendo riferimento al caso medio invece che al caso peggiore.

Prima di procedere all'analisi del caso medio del metodo di ordinamento veloce, è utile valutare il suo caso migliore. Questo si verifica quando l'elemento di pivot permette di partizionare ogni volta il vettore in due sottovettori aventi lo stesso numero di elementi. In tal caso riconduciamo il problema a due problemi di ordinamento di $n/2$ elementi e ritroviamo un'equazione simile alla equazione di ricorrenza studiata per l'algoritmo di *mergesort*. Infatti se supponiamo che il numero di elementi da ordinare sia una potenza di due ($n = 2^{m+1}$ per qualche m) e che ogni volta venga scelto come pivot quell'elemento che permette di dividere l'array in due parti aventi lo stesso numero di elementi. In questo caso possiamo riscrivere l'equazione (1) nel seguente modo (dove l'indice o sta per il caso ottimo):

$$Q_o(n) = c n + Q_o(n/2) + Q_o(n/2) = c n + 2Q_o(n/2)$$

la cui soluzione è

$$Q_o(n) = O(n \log n)$$

È possibile dimostrare che questo valore rappresenta il costo dell'algoritmo nel caso migliore. Nel caso che n non sia una potenza di 2 allora partizionando ogni volta il vettore in due parti approssimativamente uguali è possibile scrivere una equazione di ricorrenza che ha la medesima soluzione.

È facile vedere che l'albero delle attivazioni ricorsive della procedura nel caso migliore risulta bilanciato analogamente all'albero di fig. 17 che abbiamo ottenuto per l'algoritmo di ordinamento per fusione.

Il buon comportamento medio del *quicksort* può essere spiegato in modo intuitivo utilizzando l'analisi del caso migliore appena vista. Facciamo riferimento allo schema dell'algoritmo di ordinamento veloce di fig. 18, in cui si effettua una scelta casuale dell'elemento di pivot. La scelta casuale del pivot rende improbabile l'evento che il pivot divida l'array in due parti, tali che una risulti molto più grande dell'altra. La situazione più probabile è che la divisione dell'array avvenga in due parti aventi all'incirca lo stesso numero di elementi.

In altre parole, il comportamento nel caso medio è simile al comportamento

nel caso migliore, in cui la divisione dell'array avviene in due parti con lo stesso numero di elementi, piuttosto che al caso peggiore, in cui si divide l'array in due parti, una avente un solo elemento e l'altra contenente tutti gli altri.

Senza entrare nei dettagli che una trattazione formale del comportamento medio di un algoritmo richiederebbe, osserviamo che, se facciamo riferimento allo schema di fig. 18, ciascun elemento dell'array può essere scelto come pivot con uguale probabilità, esattamente pari a $1/n$ (n rappresenta il numero di elementi dell'array). Pertanto la prima attivazione della procedura *sort* può partizionare l'array $z[1,n]$ in due parti aventi un numero di elementi pari a i e $(n-i)$, con uguale probabilità pari a $1/n$ per ogni valore $i=1, 2, \dots, n-1$. Se sommiamo rispetto ad ogni indice i e calcoliamo il valore atteso possiamo riscrivere l'equazione (1) nel seguente modo (dove l'indice m sta per medio)

$$Q_m(n) = c + n + (1/n) \sum_{i=1}^{n-1} (Q_m(i) + Q_m(n-i))$$

L'equazione precedente asserisce che il costo medio della attivazione della procedura *sort* ($1,n$) è cn più il tempo medio speso per le attivazioni ricorsive. Questo termine è espresso come la somma, su tutti i possibili valori di i , della probabilità che il primo gruppo sia composto da i elementi (e quindi il secondo da $(n-i)$ elementi) moltiplicato per il costo medio delle due attivazioni ricorsive. Si noti che

$$\sum_{i=1}^{n-1} Q_m(i) = \sum_{i=1}^{n-1} Q_m(n-i)$$

e, quindi, possiamo riscrivere l'equazione di ricorrenza nel seguente modo

$$Q_m(n) = c n + (2/n) \sum_{i=1}^{n-1} Q_m(i)$$

È possibile dimostrare che la soluzione di questa equazione di ricorrenza è $O(n \log n)$.

Abbiamo pertanto il seguente teorema riguardante la complessità dell'ordinamento veloce nel caso medio.

Teorema (caso medio e migliore dell'ordinamento veloce). L'algoritmo di ordinamento veloce ha complessità $O(n \log n)$ sia nel caso migliore che in quello medio.

Poiché nell'analisi del costo di un algoritmo si ignorano le costanti multipli-

cative, il teorema precedente asserisce che, a meno di un fattore moltiplicativo, il comportamento nel caso medio dell'algoritmo di ordinamento veloce è pari a quello dell'algoritmo di ordinamento per fusione nel caso peggiore.

Con un'analisi più raffinata è possibile valutare anche la costante moltiplicativa, mostrando che, nel caso medio, essa risulta minore per l'algoritmo di ordinamento veloce. In questo modo è possibile spiegare, da un punto di vista teorico, il buon comportamento pratico del quicksort. Senza approfondire oltre l'analisi ci si limita ad osservare che la grande efficienza, da un punto di vista pratico, del metodo quicksort è dovuta essenzialmente alla semplicità con cui è possibile partizionare un array in due parti rispetto ad un dato elemento di pivot.

4.5. Delimitazione inferiore alla complessità

In questo paragrafo restringiamo la nostra attenzione a quegli algoritmi di ordinamento basati sul confronto di due chiavi. Osserviamo che tutti gli algoritmi visti precedentemente utilizzano il confronto di due elementi per progredire nella conoscenza dell'ordinamento.

Assumiamo che gli elementi da ordinare siano tutti distinti e che i numeri da ordinare siano $a(1), a(2), \dots, a(n)$. Dato un algoritmo di ordinamento P ed un particolare input $I(n)$ di n dati, $I(n) = \langle a(1), a(2), \dots, a(n) \rangle$, sia $P[I(n)]$ il numero di confronti effettuato da P per ordinare l'istanza $I(n)$.

Nel seguito si dimostrerà che, per ogni algoritmo di ordinamento P e per ogni intero n , esiste almeno un input $J(n)$ di n valori tale che $P[J(n)]$ è $\Omega(n \log n)$ (dove il logaritmo è in base 2). In altre parole, il teorema che verrà dimostrato asserisce che $\Omega(n \log n)$ confronti sono necessari per ordinare un insieme di n elementi o, in modo ancora diverso, che il problema dell'ordinamento di n elementi ha complessità $\Omega(n \log n)$ (sempre che si limiti l'attenzione ad algoritmi che utilizzano il confronto fra elementi come operazione fondamentale).

La dimostrazione del teorema si basa sulle seguenti osservazioni:

1. il problema dell'ordinamento di una sequenza $I(n)$ di n valori, $I(n) = \langle a(1), a(2), \dots, a(n) \rangle$ tutti diversi fra di loro, consiste nella ricerca di una particolare permutazione di n oggetti che applicata a $I(n)$ fornisce una sequenza ordinata;
2. esistono $n!$ permutazioni di n elementi; quindi in base all'osservazione precedente l'ordinamento di n valori equivale alla ricerca di una particolare

- permutozione in un insieme di $n!$ permutazioni;
3. prima di effettuare il primo confronto tutte le $n!$ permutazioni sono candidate per essere la permutazione cercata. Successivamente, ogni operazione di confronto riduce il numero delle permutazioni fino a quando non è rimasta una sola permutazione che è quella cercata.

Per dimostrare il teorema è utile introdurre una rappresentazione grafica che permette di visualizzare la conoscenza che un algoritmo acquisisce sulla permutazione che individua l'ordinamento man mano che effettua confronti.

Dato un qualsiasi algoritmo di ordinamento utilizzato per ordinare array di n elementi possiamo associare ad esso un albero binario. Ogni nodo dell'albero rappresenta la conoscenza che il programma ha acquisito sull'ordinamento dei dati dopo che sono stati effettuati un certo numero di confronti. Più precisamente, la radice dell'albero rappresenta lo stato del programma quando non è stato effettuato alcun confronto; essa, pertanto, rappresenta tutti i possibili ordinamenti dei dati. Supponiamo inoltre che il primo confronto effettuato dall'algoritmo sia fra le chiavi $a(i)$ e $a(j)$. In questo caso la radice dell'albero ha due figli: il figlio sinistro rappresenta tutti i possibili ordinamenti in cui $a(i)$ precede $a(j)$; il figlio destro rappresenta tutti i possibili ordinamenti in cui $a(j)$ precede $a(i)$.

Si procede in modo analogo: se un nodo rappresenta due o più possibili ordinamenti allora è necessario effettuare ulteriori confronti. Se il test che viene effettuato è il confronto delle due chiavi $a(k)$ e $a(m)$ allora il nodo ha due figli: il figlio sinistro rappresenta quegli ordinamenti possibili in cui $a(k)$ precede $a(m)$, mentre il figlio destro rappresenta quegli ordinamenti in cui $a(m)$ precede $a(k)$.

Invece, se un nodo i rappresenta un solo ordinamento, allora questo significa che l'insieme dei confronti associati al cammino fra la radice dell'albero e il nodo i permette di individuare in modo unico la permutazione associata a i .

La massima distanza fra la radice e una foglia dell'albero individua il numero di confronti che l'algoritmo deve effettuare nel caso peggiore. È facile convincersi che ad algoritmi diversi corrispondono alberi diversi.

Esempio. Supponiamo di dover ordinare i tre valori $a(1), a(2), a(3)$ utilizzando l'algoritmo di fig. 21.

Vediamo ora come all'algoritmo di fig. 21 possiamo associare l'albero di fig. 22. All'inizio ciascuno dei seguenti 6 possibili ordinamenti viene associato alla radice (nodo 1): $a(1) a(2) a(3)$, $a(1) a(3) a(2)$, $a(2) a(1) a(3)$, $a(2) a(3) a(1)$, $a(3) a(1) a(2)$, $a(3) a(2) a(1)$.

```

if  $a(1) < a(2)$ 
  then if  $a(1) < a(3)$ 
    then if  $a(2) < a(3)$ 
      then writeln ( $a(1), a(2), a(3)$ )
      else writeln ( $a(1), a(3), a(2)$ )
    else writeln ( $a(3), a(1), a(2)$ )
  else if  $a(1) < a(3)$ 
    then writeln ( $a(2), a(1), a(3)$ )
  else if  $a(2) < a(3)$ 
    then writeln ( $a(2), a(3), a(1)$ )
  else writeln ( $a(3), a(2), a(1)$ )

```

Fig. 21 - Algoritmo di ordinamento di tre numeri

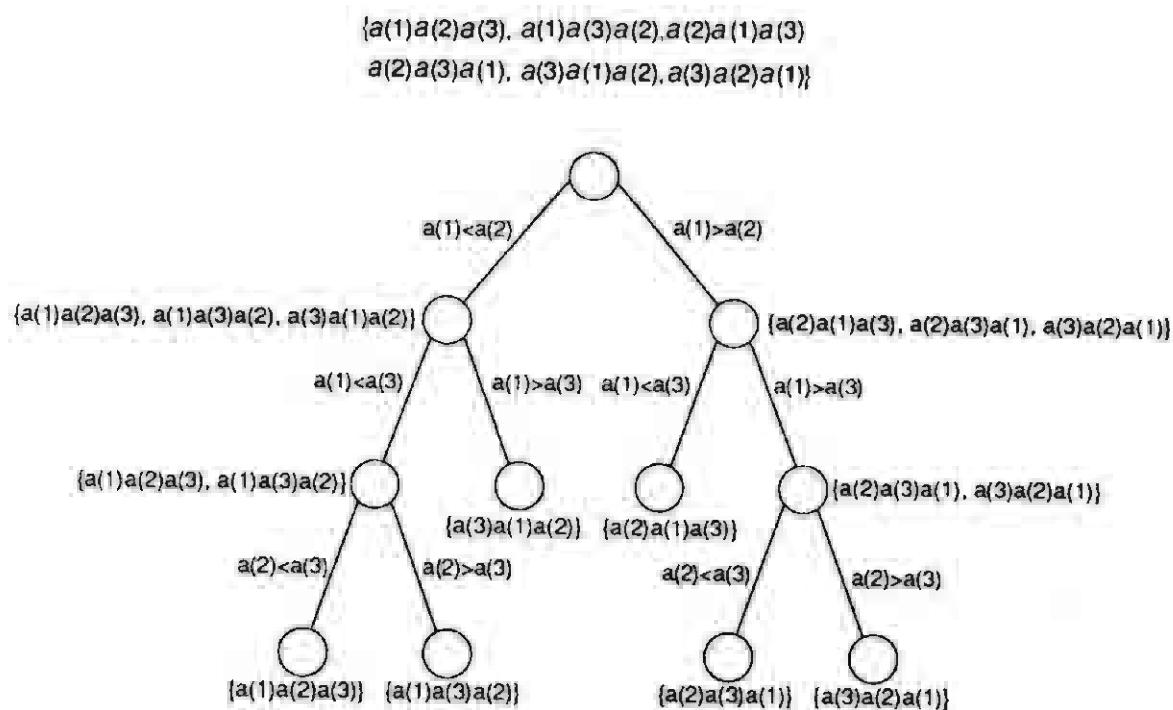


Fig. 22 - Albero associato al programma di fig. 21

Dato che il primo confronto è fra $a(1)$ e $a(2)$ allora al nodo 2, figlio sinistro della radice, si associano i tre ordinamenti in cui $a(1)$ precede $a(2)$, cioè $a(1) a(2) a(3)$, $a(1) a(3) a(2)$, $a(3) a(1) a(2)$; invece al nodo 3, figlio destro, si associano gli ordinamenti in cui $a(2)$ precede $a(1)$, cioè $a(2) a(1) a(3)$, $a(2) a(3)$

$a(1)$, $a(3)$ $a(2)$ $a(1)$.

Una volta raggiunto il nodo 2 si effettua il confronto fra $a(1)$ e $a(3)$ ottenendo due nuovi nodi 4 e 5. Il nodo 4 rappresenta le permutazioni $a(1) a(3) a(2)$, $a(1) a(2) a(3)$ compatibili con le condizioni $a(1) > a(2)$ e $a(1) > a(3)$; il nodo 5 rappresenta l'ordinamento $a(3) a(1) a(2)$, che è l'unico compatibile con le condizioni $a(1) > a(2)$ e $a(3) > a(1)$. Nel primo caso l'algoritmo non può terminare perché ancora non sappiamo se $a(2)$ precede $a(3)$ oppure no; nel secondo caso, invece, l'algoritmo termina perché ha individuato l'ordinamento dei dati.

Si noti che ciascuna foglia dell'albero rappresenta un possibile ordinamento dei dati. Infatti l'albero di fig. 22 ha 6 foglie corrispondenti a sei possibili ordinamenti dei valori $a(1)$ $a(2)$ $a(3)$. In generale con n elementi vi sono $n!$ possibili ordinamenti. Pertanto ogni albero associato ad un algoritmo di ordinamento basato su confronti deve avere almeno $n!$ foglie.

Osserviamo, inoltre, che la lunghezza del cammino fra la radice ed una foglia rappresenta il numero di confronti effettuato quando la foglia rappresenta l'ordinamento cercato. Pertanto la profondità dell'albero rappresenta il numero di confronti necessario nel caso peggiore. Poiché ogni nodo può avere solo due figli allora alberi binari con molte foglie devono avere cammini lunghi. In generale un albero binario di profondità i ha al più $(2^{**}i)$ foglie. Questo equivale a dire che un albero binario con k foglie ha profondità almeno $\log k$ (dove il logaritmo è in base 2).

Se poniamo $k=n!$ allora sappiamo che ogni algoritmo di ordinamento che usa solo confronti deve effettuare almeno $\log(n!)$ confronti. Dato che $n! \approx (n/e)^{**}n$ dove $e=2,7183\dots$ allora sono necessari

$$\log(n!) \approx \log((n/e)^{**}n) = n \log n - n \log e = O(n \log n)$$

confronti.

Poiché i ragionamenti fatti valgono per ogni algoritmo di ordinamento basato sul confronto di chiavi, possiamo enunciare il seguente teorema.

Teorema (delimitazione inferiore dell'ordinamento). Ogni algoritmo di ordinamento basato sul confronto di chiavi ha complessità $\Omega(n \log n)$.

4.6. Binsort

In questo paragrafo presentiamo ed analizziamo un algoritmo di ordinamento che utilizza la conoscenza dell'intervallo dei possibili valori delle chiavi, e, al posto dell'operazione di confronto, un'operazione di indirizzamento con

indici.

Si supponga inizialmente di dover ordinare l'array $z[1..n]$ le cui chiavi assumano valori interi fra 1 e n senza duplicazioni. Siano z e b gli array di input e output, rispettivamente (in z sono memorizzati gli elementi da ordinare e in b verranno memorizzati gli elementi ordinati). In questo caso l'algoritmo di ordinamento è banale: per ordinare il vettore è sufficiente inserire l'elemento con chiave 1 nella prima posizione di b , l'elemento di chiave 2 nella seconda posizione e così via. Lo schema dell'algoritmo è dato in fig. 23.

La dimostrazione della correttezza dell'algoritmo è banale, come pure è

Schema dell'algoritmo binsort di un array $z[1..n]$ con chiavi comprese fra 1 e n

```
begin
  for i := 1 to n
    do begin
      k := z[i].chiave;
      "poni ciascun campo di b[k] pari
       al corrispondente campo di z[i]"
    end
  end;
```

Fig. 23 - Schema dell'algoritmo binsort

facile osservare che la sua complessità è $O(n)$.

È possibile applicare un'idea simile anche in un caso più generale. Supponiamo infatti che i valori delle chiavi siano compresi tra 1 e m e che si possano avere duplicazioni di chiavi. In questo caso si utilizza un array b di m componenti dello stesso tipo di z . L'elemento $b[i]$ è un puntatore ad una lista in cui poniamo tutti gli elementi di chiave i . Quando tutti gli elementi sono stati considerati, le liste non vuote vengono concatenate per ottenere un'unica lista.

Per quanto riguarda la complessità del metodo osserviamo che il passo di inizializzazione ha complessità $O(m)$, mentre il passo di inserimento degli elementi ha costo $O(n)$. Inoltre non è difficile implementare il passo di collegamento in modo tale da ottenere una complessità $O(m)$. Pertanto il costo totale è $O(n+m)$; nel caso che m sia $O(n)$ allora l'algoritmo di binsort risulta

```

procedure binsort (var z : vett; m,n : integer; piniz: punt);
{ si assume che le chiavi abbiano valori compresi tra 1 e m; la procedura
 prende in ingresso il vettore z da ordinare e restituisce un puntatore piniz ad
 una lista contenente gli elementi ordinati}
var i, k : integer;
    b : array [1..m] of elem;
begin
{inizializza l'array }
for i := 1 to m do b[i] := nil;

{inserimento degli elementi nelle liste corrispondenti}
for i := 1 to n
do begin
    k := z[i].chiave;
    "inserisci z[i] nella lista b[k]"
    end;

{collegamento delle liste}
piniz := nil;
for i := 1 to m
do if b[i] <> nil
    then "collega la lista puntata da
          b[i] alla lista puntata da piniz"
end;

```

Fig. 24 - Algoritmo binsort

lineare.

Questo algoritmo di ordinamento risulta, quindi, più efficiente degli altri algoritmi visti se l'intervallo dei possibili valori delle chiavi risulta limitato.

5. CONCLUSIONI

Terminiamo il capitolo con due osservazioni finali. La prima osservazione riguarda la scelta dell'algoritmo da implementare. Abbiamo visto nel corso del capitolo, che possiamo avere diversi algoritmi di soluzione per uno stesso problema e ciò pone il problema di scegliere il migliore. Le proprietà che desideriamo dall'algoritmo scelto sono principalmente due:

1. l'algoritmo deve essere semplice, in modo tale da facilitare la comprensione, la programmazione e la correzione del programma;
2. l'algoritmo deve essere efficiente, in modo tale che il programma richieda una quantità limitata di risorse per la sua esecuzione.

La prima proprietà fa riferimento al costo umano necessario per la scrittura del programma, mentre la seconda fa riferimento al costo di esecuzione del programma. Le due proprietà sono spesso in contraddizione fra loro. Regole precise che aiutino a decidere quale algoritmo applicare nei diversi casi non esistono, ma è importante considerare che

1. se il programma deve essere eseguito poche volte su input aventi piccole dimensioni, allora è generalmente più importante ricercare la semplicità. Infatti, in questo caso il costo umano prevale sul costo della macchina;
2. se, invece, il programma deve essere eseguito molte volte su input aventi dimensioni grandi allora conviene cercare di implementare algoritmi efficienti perché il risparmio di risorse della macchina può convenire rispetto al maggior tempo che si richiede al programmatore.

In relazione a quanto detto è utile un confronto con quanto visto nel cap. 4 sulle metodologie di programmazione.

La seconda osservazione riguarda il problema dell'implementazione efficiente dell'algoritmo scelto. Una volta scelto l'algoritmo è necessario cercarne una implementazione efficiente. Nella valutazione asintotica della complessità di un programma abbiamo ipotizzato che il costo di esecuzione sia uguale per tutte le istruzioni e abbiamo ignorato costanti moltiplicative della funzione che esprime la complessità di un programma, ottenendo in tal modo una valutazione approssimata del costo di un programma.

In particolare, se si vogliono progettare algoritmi che richiedono un limitato tempo di calcolo per la loro esecuzione, è opportuno evitare di scrivere programmi ricorsivi. Infatti ogni attivazione di procedura comporta l'esecuzione di numerose operazioni. Il metodo proposto per la valutazione del costo del programma, non tiene conto di questo tempo e suppone che tutte le operazioni relative alla attivazione di una procedura abbiano costo zero. La semplificazione introdotta non crea grossi problemi di valutazione se il numero delle chiamate di procedura è piccolo. Questo non è più vero nel caso di programmi ricorsivi. Abbiamo visto, ad esempio, che per ordinare un vettore con il programma *mergesort* sono necessarie un numero di attivazioni della procedura *msort* all'incirca pari al numero degli elementi da ordinare. Da un punto di vista pratico questo comporta che il tempo di esecuzione della versione ricorsiva dell'algoritmo di ordinamento per fusione è maggiore del tempo di

esecuzione degli altri algoritmi visti (anche se la complessità dell'ordinamento per fusione è minore). A questo proposito ricordiamo che non è necessario implementare un algoritmo ricorsivo mediante un programma ricorsivo. Infatti, per ogni programma ricorsivo esiste un programma non ricorsivo equivalente la cui esecuzione risulta più efficiente.

7.

Elementi di calcolabilità

In questo capitolo viene introdotta la *teoria della calcolabilità*. Essa si occupa dei fondamenti del calcolo, inteso nella sua accezione più ampia.

La teoria della calcolabilità ha due scopi principali: il primo è scoprire quali sono i limiti del calcolo, cioè caratterizzare le funzioni che è possibile calcolare in un dato formalismo e quelle che non sono calcolabili. Per fare questo è necessario innanzitutto introdurre un modello di calcolo, cioè definire formalmente un insieme di regole attraverso cui procede il calcolo, e, quindi, stabilire l'insieme delle funzioni che è possibile calcolare con esso. Il secondo obiettivo è quello di confrontare i modelli via via proposti per formalizzare il concetto di calcolo, con particolare attenzione alla potenza espressiva e alla semplicità; siamo cioè interessati a quei modelli che utilizzino un insieme di regole il più ridotto possibile.

La teoria della calcolabilità ha raggiunto i suoi principali risultati negli anni '30, quindi molto tempo prima che fossero introdotti elaboratori per il calcolo. I risultati ottenuti allora hanno una grande importanza dal punto di vista matematico e logico. Essi sono stati successivamente utilizzati per caratterizzare i limiti e la potenza del calcolo automatico e, quindi, hanno ora anche una notevole importanza pratica.

1. FUNZIONI CALCOLABILI E MODELLI DI CALCOLO

In questo capitolo verranno esposti i risultati principali della teoria della calcolabilità facendo riferimento a un modello di calcolo basato sul linguaggio Pascal.

Abbiamo visto nella introduzione al testo che si può definire un algoritmo

come una serie di passi che elaborano dei dati di ingresso, fornendo, eventualmente dei dati di uscita. Rientrano in questa definizione algoritmi che risolvono sia problemi di calcolo di funzioni matematiche (come ad esempio il calcolo degli zeri di un polinomio o il calcolo del prodotto di due matrici) che problemi di natura apparentemente più generale, come, ad esempio, la ricerca di una strategia vincente in giochi come gli scacchi o la dama a partire da una data configurazione dei pezzi sulla scacchiera, oppure la traduzione di un programma Pascal in un linguaggio macchina di un particolare elaboratore.

È importante osservare che ogni problema che intendiamo automatizzare può essere ricondotto al calcolo di una funzione, in quanto stabilisce una corrispondenza tra un insieme di dati di ingresso ed un insieme di dati di uscita.

Il dominio e il codominio della funzione possono essere di natura molto varia. Però, per non complicare il nostro studio, limiteremo, la nostra attenzione al calcolo di funzioni definite sull'insieme N dei numeri naturali. Si noti che questa semplificazione non limita in modo sostanziale la nostra trattazione, perché è sempre possibile codificare per mezzo di numeri naturali qualunque dato. In questo modo il problema originario viene ricondotto al calcolo di una funzione matematica definita sui naturali. Abbiamo, ad esempio, che una qualunque posizione del gioco degli scacchi può essere codificata associando ad ogni pezzo quattro numeri positivi che rappresentano il suo colore (bianco o nero), il tipo del pezzo (re, regina, torre, ...), e le coordinate della sua posizione. Analogamente ogni mossa può essere codificata utilizzando sei numeri: i primi due rappresentano il pezzo che si muove e gli altri rappresentano, rispettivamente, la sua posizione iniziale e finale. Pertanto, la determinazione di una strategia vincente può essere ricondotta al calcolo della funzione avente come dominio una codifica delle possibili configurazioni sulla scacchiera e come codominio la codifica delle possibili mosse. Analogamente la traduzione di un programma Pascal può essere vista come il calcolo di una funzione che fa corrispondere ad una stringa di caratteri alfanumerici (il programma Pascal) a una nuova stringa alfanumerica (il programma scritto in linguaggio macchina). Se utilizziamo un qualunque codice binario i due programmi appariranno come due stringhe formate da 0 e 1 e, quindi, possono essere interpretati come due numeri positivi scritti in notazione binaria. Pertanto, la traduzione di un programma può essere vista come il calcolo di una particolare funzione $f: N \rightarrow N$.

Nel resto del capitolo cercheremo di caratterizzare le funzioni calcolabili, cioè quelle funzioni per cui esiste un algoritmo che permette di calcolare il valore della funzione per ogni valore del dominio. Ricordiamo a questo proposito che nella definizione di algoritmo data nell'introduzione non è

necessaria l'esistenza di un elaboratore in grado di eseguire le istruzioni, ma si richiede che queste ultime verifichino le proprietà di non ambiguità, eseguibilità e finitezza.

L'insieme di regole non ambigue ed eseguibili utilizzate per definire un algoritmo definisce un *modello di calcolo*.

Esempi di modello di calcolo sono dati dai linguaggi programmativi le cui istruzioni possono essere eseguite da un elaboratore e, quindi, devono essere necessariamente non ambigue ed eseguibili.

2. UN MODELLO DI CALCOLO BASATO SUL PASCAL

Scopo principale di questo paragrafo è ricercare il modello di calcolo più semplice che è in grado di calcolare tutte le funzioni che sono calcolabili con il Pascal. Sono noti diversi modelli di calcolo che possono essere definiti con poche regole e che risultano equivalenti al Pascal dal punto di vista computazionale. Nel seguito presenteremo uno di tali modelli, il linguaggio mini-Pascal, che è essenzialmente un sottoinsieme del linguaggio Pascal molto simile a quello presentato nella introduzione. Nel mini-Pascal le variabili possono assumere solo valori interi non negativi e, inoltre, le istruzioni sono un sottoinsieme delle istruzioni del Pascal. Vedremo che, nonostante queste limitazioni, con il mini-Pascal è possibile calcolare lo stesso insieme di funzioni definite sui numeri naturali calcolabili con il Pascal. In altre parole possiamo dire che la potenza di calcolo del Pascal dipende da un insieme molto ridotto di istruzioni.

2.1. Il linguaggio mini-Pascal

Un programma mini-Pascal è composto da una intestazione, da una sezione di dichiarazione delle variabili che possono assumere solo valori interi e da una sezione istruzioni.

Le istruzioni del linguaggio ammesse sono le istruzioni di lettura o di scrittura, l'istruzione composta, l'istruzione **while** e l'istruzione di assegnazione. Le prime quattro istruzioni sono definite in modo analogo a quanto fatto nell'introduzione al testo a cui si rimanda. L'istruzione di assegnazione può assumere una delle possibili seguenti forme

- `<identificatore> := 0`
- `<identificatore> := <identificatore>`

- $\langle\text{identificatore}\rangle := \langle\text{identificatore}\rangle + 1$

Si noti la forma estremamente semplice dell'istruzione rispetto al Pascal, in cui è possibile assegnare alla variabile a sinistra il valore di una espressione aritmetica che utilizza una o più variabili e le quattro operazioni aritmetiche. Osserviamo inoltre che nel mini-Pascal non è definita l'istruzione condizionale che può essere eseguita mediante l'istruzione **while**.

2.2. Funzioni calcolabili in mini-Pascal

Mostriamo alcuni esempi di funzioni definite sugli interi calcolabili da un programma scritto in mini-Pascal.

Prima di procedere è necessario definire formalmente il concetto di funzione calcolabile in mini-Pascal. Poiché siamo interessati a programmi che calcolano funzioni matematiche è innanzitutto opportuno distinguere tra le variabili utilizzate dal programma due sottoinsiemi disgiunti, le variabili di ingresso, indicate nel seguito con x_1, x_2, \dots, x_r , e le variabili di uscita indicate nel seguito con y_1, y_2, \dots, y_s .

Definizione 1. Un programma p scritto in mini-Pascal avente come variabili di ingresso x_1, x_2, \dots, x_r e come variabili di uscita y_1, y_2, \dots, y_s , calcola la funzione

$$f: \mathbb{N}^r \rightarrow \mathbb{N}^s$$

se valgono le seguenti condizioni:

- se $f(X_1, X_2, \dots, X_r) = Y_1, Y_2, \dots, Y_s$, allora l'esecuzione di p su valori X_1, X_2, \dots, X_r delle variabili di ingresso termina producendo i valori Y_1, Y_2, \dots, Y_s delle variabili di uscita;
- se $f(X_1, X_2, \dots, X_r)$ non è definita, allora l'esecuzione di p su valori X_1, X_2, \dots, X_r delle variabili di ingresso non termina.

Definizione 2. Una funzione è *calcolabile con il mini-Pascal* (per brevità *mP-calcolabile*) se esiste un programma scritto in mini-Pascal che la calcola.

Diamo ora alcuni esempi di funzioni mP-calcolabili.

1. *Somma di due numeri a,b.* La somma c di due numeri interi a, b può essere realizzata sommando 1 al numero a per b volte (vedi il programma di fig. 1).

2. *Differenza di due numeri.* Poiché nel mini-Pascal le variabili non possono assumere valori negativi è necessario definire una nuova operazione di sottrazione.

```

program somma;
var a, b, c, i : integer;
begin
  readln (a,b);
  c := b ; i := 0 ;
  while i < b
  do begin
    c := c + 1 ;
    i := i + 1
  end ;
  writeln (c)
end.

```

Fig. 1 - Programma mini-Pascal per l'operazione di somma

```

program a_meno_1 ;
var a, b, c, d : integer;
begin
  readln (a);
  d := 0 ;
  while d < a
  do begin
    c := d;
    d := d + 1
  end;
  writeln (c)
end.

```

Fig. 2 - Programma mini-Pascal per l'operazione di sottrazione di uno

zione denotata nel seguito come \perp nel seguente modo

$$a \perp b = \begin{cases} a - b & \text{se } a \text{ è maggiore di } b \\ 0 & \text{se } a \text{ è minore o uguale } b \end{cases}$$

Il programma che calcola $a \perp 1$ secondo la nuova definizione viene dato in fig. 2.

Il calcolo di $a \perp b$ può essere ottenuto sottraendo 1 per b volte (vedi il programma di fig. 3).

```

program sottrazione;
var a , b , c , d , e , i : integer;
begin
  readln (a,b);
  c := a ;
  i := 0 ;
  while i < b
  do begin
    d := 0 ;
    e := c ;
    while d < e
    do begin
      c := d ;
      d := d + 1
    end;
    i := i +1
  end;
  writeln (c)
end.

```

Fig. 3 - Programma mini-Pascal per l'operazione di sottrazione

L'esempio della fig. 3 mostra che i programmi mini-Pascal hanno una notevole complessità strutturale e sono molto lunghi anche nel caso di funzioni semplici. Questo non deve sorprendere poiché il mini-Pascal è un linguaggio dotato di istruzioni molto elementari.

Per rendere i programmi più brevi e, quindi più facili da comprendere, introduciamo una istruzione simile a quella di attivazione di funzione nel linguaggio Pascal. In particolare se denotiamo con

$$f : N^r \rightarrow N^s$$

una funzione mP-calcolabile, con la notazione

$$(y_1, y_2, \dots, y_s) := f(x_1, x_2, \dots, x_r)$$

rappresentiamo sinteticamente il frammento di programma che calcola la funzione f e modifica solo le variabili y_1, y_2, \dots, y_s lasciando invariato il valore delle altre variabili del programma. Questa notazione è in un certo modo equivalente alle funzioni del Pascal e verrà usata nel seguito ognqualvolta in

un programma mini-Pascal incontriamo una funzione per cui sappiamo che esiste un programma mini-Pascal che la calcola.

In particolare il programma per la sottrazione può essere semplificato utilizzando la funzione sottrazione per uno come mostrato in fig. 4.

```
program sottrazione_z;
var a, b, c, j: integer;
begin
readln (a,b);
c := a ;
j := 0;
while j < b
do begin
  c := c - 1;
  j := j + 1
end;
writeln (c)
end.
```

Fig. 4 - Programma mini-Pascal per l'operazione di sottrazione utilizzando funzioni

3. *Moltiplicazione*. Utilizzando la funzione somma di due numeri, mostrata nell'esempio 1, otteniamo il programma di fig. 5 che calcola c pari a $a*b$ sommando per b volte a alla variabile c .

Vediamo ora come sia possibile rappresentare in mini-Pascal il calcolo delle funzioni booleane ed ottenere altre istruzioni del linguaggio Pascal non direttamente presenti nel mini-Pascal.

```
program moltiplicazione;
var a, b, c, i : integer;
begin
readln (a,b);
c := 0;
i := 0;
while i < b
do begin
  i := i + 1;
  c := c + a
end;
writeln (c)
end.
```

Fig. 5 - Programma mini-Pascal per l'operazione di moltiplicazione

Per rappresentare variabili booleane utilizziamo una variabile intera che assume solo i valori 0 e 1, che rappresentano, rispettivamente, i valori falso e vero.

4. Not. Il programma di fig. 6 calcola la funzione booleana **not**; in particolare dato *a* esso assegna a *b*, il valore pari a **not a**.

```
program not ;
var a, b, c :integer ;
begin
  b := 0;
  b := b + 1;
  c := a;
  while c = 1
  do begin
    b := 0;
    c := 0
  end ;
  writeln (b)
end.
```

Fig. 6 - Programma mini-Pascal per l'operazione di negazione

5. Or. Il programma di fig. 7 legge i valori di due variabili booleane (che possono assumere solo i valori 0 e 1 e calcola *c* pari a *a or b*, utilizzando la funzione somma e poi normalizzando il risultato.

I programmi per il calcolo delle altre funzioni booleane possono essere costruiti in modo analogo. Integrando insieme i programmi che realizzano le singole operazioni è anche possibile scrivere programmi per il calcolo di espressioni booleane complesse.

```
' program or;
var a, b, c :integer;
begin
  readln (a,b);
  c := a + b;
  while c > 1
  do c := c - 1;
  writeln (c)
end.
```

Fig. 7 - Programma mini-Pascal per l'operazione **or**

A conclusione del paragrafo vediamo come sia parimenti possibile simulare altre istruzioni di controllo del Pascal.

6. *Istruzione while in forma generale*. Nel linguaggio Pascal, la condizione che determina la terminazione dell'esecuzione dell'istruzione **while** può essere una qualunque espressione logica *f* funzione delle variabili x_1, x_2, \dots, x_n . Supponiamo ora che *f* sia mP-calcolabile e che l'istruzione

$$z := f(x_1, x_2, \dots, x_n)$$

assegni alla variabile *z* il valore 1 se l'espressione logica è soddisfatta, il valore 0 altrimenti.

Il frammento di programma mini-Pascal di fig. 8 simula l'istruzione **while** nella sua forma più generale. La variabile *w* è una variabile di comodo che permette di calcolare l'espressione booleana prima di verificarne il valore. Si assume che *z* e *w* siano variabili non utilizzate in altra parte del programma.

```
w := 0 ;
w := w + 1 ;
while w = 1 do
  begin
    z := f(x1,x2,...,xr) ;
    w := 0 ;
    while z = 1 do
      begin
        < istruzione > ;
        w := 1
      end
  end
```

Fig. 8 - Programma mini-Pascal per l'istruzione **while** in forma generale

7. *Istruzione condizionale (if then else)*. Ricordiamo che la sintassi dell'istruzione **if then else** è

```
if <espressione booleana>
  then <istruzione>
  else <istruzione>
```

Se l'espressione è vera allora viene eseguita l'istruzione che segue la parola chiave **then**, se è falsa si passa ad eseguire l'istruzione che segue la parola chiave **else**. Supponiamo che l'espressione logica sia mP-calcolabile e che l'istruzione

$$z := f(x_1, x_2, \dots, x_r)$$

assegni a z il valore 1 se l'espressione booleana è soddisfatta, il valore 0 altrimenti.

Il frammento di programma mini-Pascal di fig. 9 simula l'istruzione condizionale. Assumiamo che z e zz siano variabili non utilizzate in altra parte del programma.

```
z := f(x1,x2,...,xr);
zz := not z;
while z = 1
do begin
    <istruzione>;                                { la condizione è vera }
    z := 0
    end
while zz = 1
do begin
    <istruzione>;                                { la condizione è falsa }
    zz := 0
    end
```

Fig. 9 - Programma mini-Pascal per simulare l'istruzione *if then else*

I frammenti di programma che simulano le altre istruzioni di controllo del Pascal possono essere costruiti in modo analogo. In questo modo si dimostra che le funzioni calcolabili con il Pascal sono tutte e sole quelle calcolabili con il mini-Pascal; quindi possiamo affermare che i due linguaggi sono equivalenti dal punto di vista computazionale.

3. FUNZIONI NON CALCOLABILI

- In questo paragrafo studieremo i limiti di calcolo del mini-Pascal ed in particolare descriveremo una funzione che non è calcolabile con programmi mini-Pascal; alla luce dei risultati del paragrafo precedente, questa funzione non è calcolabile nemmeno con programmi scritti in Pascal.

Definizione 3. Una funzione non calcolabile in mini-Pascal è una funzione *indecidibile*.

La dimostrazione dell'esistenza di una funzione indecidibile utilizza come passo iniziale la definizione di una corrispondenza biunivoca tra i numeri

naturali ed i programmi. Una tale corrispondenza può essere ottenuta ordinando alfabeticamente l'insieme di tutti i programmi mini-Pascal. Successivamente si fa corrispondere il numero 0 al primo programma, il numero 1 al secondo programma ecc., ottenendo in questo modo una corrispondenza biunivoca tra l'insieme dei programmi mini-Pascal e l'insieme dei numeri naturali.

Si noti che utilizzando la enumerazione appena ottenuta è assai complesso determinare il numero intero che corrisponde ad un dato programma o il programma che corrisponde ad un dato numero. Non approfondiremo ulteriormente la questione, ma ci limitiamo ad osservare che è possibile definire diversamente la corrispondenza biunivoca tra programmi e numeri naturali in modo tale che questi due problemi siano risolvibili in modo semplice.

Teorema 1. L'insieme dei programmi scritti in mini-Pascal è numerabile, o, equivalentemente, può essere messo in corrispondenza biunivoca con l'insieme dei numeri naturali.

Mostriamo ora un primo risultato che asserisce l'esistenza di infinite funzioni non calcolabili.

Teorema 2. L'insieme delle funzioni $f: \mathbb{N} \rightarrow \mathbb{N}$ non è numerabile.

Dimostrazione. È sufficiente limitare la nostra attenzione alle sole funzioni del tipo

$$f: \mathbb{N} \rightarrow \{0,1\}.$$

L'insieme di queste funzioni può essere messo in corrispondenza biunivoca con l'insieme delle parti di \mathbb{N} e, quindi, ha potenza superiore al numerabile (vedi l'appendice).

Corollario. L'insieme delle funzioni indecidibili non è numerabile.

Dimostrazione. Basta osservare che l'insieme dei programmi è numerabile mentre l'insieme delle funzioni ha una cardinalità superiore.

3.1. Il problema della fermata

Il corollario precedente asserisce l'esistenza di funzioni non calcolabili basandosi su un semplice argomento di conteggio, ma non fornisce un esempio di funzione indecidibile.

Vogliamo ora mostrare un esempio di tale funzione; per fare ciò restringeremo la nostra attenzione ad una particolare classe di programmi e di funzioni.

Sia P l'insieme dei programmi mini-Pascal con una sola variabile di ingresso ed una sola variabile di uscita. Non è difficile convincersi che l'insieme dei programmi P è ancora numerabile e, quindi, esiste una corrispondenza biunivoca $g: \mathbb{N} \rightarrow P$ tra gli interi e i programmi di P .

Il seguente teorema ci mostra una funzione non calcolabile.

Teorema 3. Data una qualunque corrispondenza biunivoca g tra l'insieme dei numeri naturali e l'insieme di programmi P , la funzione $h: \mathbb{N} \rightarrow \{0,1\}$ tale che

$$h(x) = \begin{cases} 0 & \text{se il programma } x\text{-esimo nella corrispondenza } g \\ & \text{si ferma con input } x \\ 1 & \text{se il programma } x\text{-esimo nella corrispondenza } g \\ & \text{non si ferma con input } x \end{cases}$$

non è calcolabile da un programma mini-Pascal.

Dimostrazione. Prima di procedere alla dimostrazione del teorema notiamo il particolare ruolo svolto dal numero x : x determina il programma di P cui fa riferimento h , e rappresenta il valore di ingresso al programma stesso.

Supponiamo per assurdo che la funzione h sia calcolabile e che quindi esista un programma p avente x e y rispettivamente come variabili di ingresso e di uscita, che termina per ogni valore dell'input con y pari a $h(x)$. Pertanto è possibile scrivere il programma *controesempio* dato in fig. 10 che risulta appartenere anch'esso all'insieme P dei programmi con una variabile di ingresso e una di uscita ed in cui l'istruzione

$$y := h(x)$$

assegna a y il valore $h(x)$.

Dato che il calcolo di $h(x)$ termina sempre, allora il programma *controesem-*

```
program controesempio;
var x, y : integer;
begin
  readln (x);
  y := 0;
  while y = 0
    do y := h(x)
  end.
```

Fig. 10 - Programma per dimostrare l'indecidibilità del problema della fermata

pio si ferma solo quando l' x -esimo programma della enumerazione data con input x non si ferma e cicla quando l' x -esimo programma della enumerazione si ferma con input x . Se il programma *controesempio* esiste allora esso sarà associato ad un numero nella corrispondenza biunivoca definita da g ; sia k tale numero.

A questo punto abbiamo ottenuto una contraddizione; infatti se $h(k)=0$ allora il programma *controesempio* non si ferma, se $h(k)=1$ allora il programma *controesempio* si ferma, contraddicendo in ambedue i casi la definizione stessa di h .

Il teorema precedente rimane valido anche quando consideriamo un arbitrario programma mini-Pascal o Pascal con un numero qualunque di variabili di ingresso e di uscita.

Il teorema può perciò essere riformulato nel modo seguente.

Teorema 4. Non è possibile scrivere un programma in mini-Pascal (o in Pascal) che, avendo in input la codifica di un arbitrario programma p scritto in mini-Pascal (o Pascal), ed un insieme arbitrario di dati i , decide se p con input i , si ferma oppure no.

Il teorema stabilisce l'indecidibilità del *problema della fermata* (in inglese *halting problem*) ed ha una importanza fondamentale per motivi sia pratici che teorici.

Infatti, dal punto di vista pratico, il teorema esclude la possibilità di avere un metodo generale che permetta di sapere per un qualunque programma e per un qualunque insieme di dati di ingresso se il programma non termina. Si noti che l'indecidibilità del problema della fermata non esclude che il problema sia risolvibile in casi particolari come in particolare abbiamo visto nel cap. 5.

Oltre alla sua importanza pratica l'indecidibilità del problema della fermata è un risultato fondamentale dal punto di vista teorico; esso assume la medesima importanza, all'interno della teoria della computabilità, di altri risultati negativi ottenuti in matematica, fisica e meccanica quali ad esempio:

- non è possibile costruire con riga e compasso un quadrato avente area pari a quella di un cerchio con raggio unitario (problema della quadratura del cerchio);
- non è possibile determinare esattamente la posizione e la velocità di un elettrone (il principio di indeterminazione di Heisenberg);
- non è possibile costruire una macchina dal moto perpetuo.

È inoltre facile rendersi conto che la medesima tecnica di dimostrazione può

essere applicata ad altri linguaggi di programmazione. Infatti la dimostrazione del teorema 3 utilizza solamente la proprietà che l'insieme dei programmi mini-Pascal può essere messo in corrispondenza con i numeri naturali; questa è una caratteristica comune a tutti i linguaggi di programmazione. Quindi è possibile applicare lo stesso schema di dimostrazione ad altri linguaggi di programmazione.

Si noti che, se consideriamo linguaggi di programmazione meno espressivi del mini-Pascal è però possibile, in alcuni casi, risolvere il problema della fermata. Ad esempio, se eliminiamo dal mini-Pascal l'istruzione **while**, allora ogni programma avrà solo istruzioni di assegnazione o di lettura e scrittura ed è quindi banale poter dimostrare che tutti i programmi si fermano. Analogamente se si elimina l'istruzione **while** e si introduce un'istruzione di ciclo come l'istruzione **for**, si ottengono programmi che terminano per ogni possibile valore dei dati di ingresso.

3.2. La tesi di Church

Dopo aver studiato la potenza e i limiti del mini-Pascal è naturale chiedersi se esiste un modello di calcolo più potente del mini-Pascal, che permette di calcolare alcune delle funzioni non calcolabili con esso. In particolare è stato definito un insieme di funzioni noto come l'insieme delle *funzioni ricorsive*. È possibile dimostrare che una funzione è mP-calcolabile se e solo se è una funzione ricorsiva. Pertanto la ricerca di modelli di calcolo più potenti del mini-Pascal equivale a costruire un modello di calcolo in grado di calcolare una funzione al di fuori di questo insieme.

Molti modelli di calcolo sono stati proposti ma il principale risultato ottenuto è che tutti quanti calcolano un insieme di funzioni coincidente con l'insieme delle funzioni ricorsive. In altre parole è possibile dimostrare che una funzione è calcolabile in un modello se e solo se è calcolabile da uno qualunque degli altri. Il risultato è tanto più sorprendente se consideriamo che tutti questi modelli sono stati proposti in contesti diversi e ha portato Church a ipotizzare la non esistenza di un modello di calcolo in grado di calcolare funzioni non ricorsive. Questa ipotesi, formulata nel 1936, è nota con il nome di *Tesi di Church*.

Tesi di Church. Una funzione è calcolabile se e solo se essa è una funzione ricorsiva.

Ovviamente non si potrà mai dimostrare la validità della tesi di Church perché in essa vi è un riferimento intuitivo al concetto di calcolabilità. Al contrario se ne può dimostrare la falsità, costruendo un modello in grado di calcolare una funzione non calcolabile con uno dei formalismi proposti. Questo è stato l'obiettivo di molti ricercatori, ma come abbiamo detto, con risultati infruttuosi. Pertanto oggi la tesi di Church è generalmente accettata.

4. CENNI STORICI

Il concetto di algoritmo ovvero di procedura effettiva di risoluzione di un problema matematico, è un concetto che sta alla base di molti dei risultati raggiunti dalla matematica in ogni epoca storica. Comunque è stato soltanto nel corso di questo secolo, e più precisamente dal finire degli anni venti in poi, che sono stati formalizzati rigorosamente i concetti di funzione calcolabile e non calcolabile, di problema risolubile e non risolubile ponendo così le basi della teoria della calcolabilità.

Per illustrare questo concetto ricordiamo che fin dall'antichità sono stati scoperti e inventati importanti algoritmi per la risoluzione di un problema. Un esempio è dato dall'algoritmo delle divisioni successive per il calcolo del massimo comun divisore di due interi. Questo algoritmo è descritto per la prima volta negli *Elementi* di Euclide (V secolo a.c.) ma potrebbe essere addirittura anteriore. Metodi algoritmici per la risoluzione di problemi algebrici furono sviluppati da studiosi arabi e la stessa parola algoritmo ha origine dal nome del matematico arabo Al Khuwarizmi, vissuto probabilmente nel nono secolo dopo Cristo.

Successivamente, nel corso dei secoli, diventa sempre più chiara la necessità di una formalizzazione generale del concetto di funzione calcolabile. Tuttavia, questa necessità non si accompagna alla messa in discussione dell'esistenza di funzioni non calcolabili ovvero di problemi di cui non si possa trovare una soluzione matematica rigorosa.

Infatti, anche se era stata notata l'irrisolubilità di certi problemi matematici mediante l'uso di determinate tecniche, vi era la convinzione unanime che bastasse impiegare mezzi più potenti per poter ancora trovare una soluzione. Ad esempio, era stata dimostrata l'impossibilità di risolvere il problema geometrico della trisezione di un angolo usando un compasso ed una riga non graduata; ma era bastato impiegare un compasso e una riga graduata per risolvere il problema. Analogamente, era stata dimostrata l'impossibilità di

risolvere un'equazione algebrica di quinto grado o di grado superiore mediante radicali; tuttavia erano stati trovati metodi approssimati che permettono di risolvere l'equazione con un qualsiasi grado di approssimazione.

Era quindi comune convinzione che un qualunque problema matematico potesse sempre essere risolto, eventualmente in modo approssimato. Questa convinzione fece nascere il bisogno di definire formalmente il concetto di algoritmo nell'ambito di un sistema logico.

Per capire meglio gli sviluppi successivi è necessario operare una breve digressione. Dopo la fondazione della teoria degli insiemi, dovuta a Cantor, alla fine del secolo scorso e all'inizio del nostro, importanti logici quali Frege, Russell, Peano cercarono di utilizzare la teoria degli insiemi per poter fondare rigorosamente la matematica. L'idea era quella di partire da alcune semplici proposizioni di teoria degli insiemi, la cui verità poteva essere intuitivamente accettata da tutti e applicare ad esse un insieme di regole di deduzione logica. Se in questo modo fosse stato possibile costruire le più importanti teorie matematiche (l'aritmetica, la teoria dei numeri reali, ecc.), si era garantiti di non aver mai introdotto il falso e quindi la contraddizione nel ragionamento matematico.

Sfortunatamente, da Russell in poi, fu chiaro che un tale progetto non era praticabile; infatti, sfruttando semplicemente lo stesso concetto di insieme, era assai facile introdurre contraddizioni nel sistema logico. Una delle più famose contraddizioni che possono essere create è il cosiddetto paradosso di Russell che stabilisce la non realizzabilità di un simile progetto. Il paradosso di Russell può essere parafrasato nel seguente modo:

Paradosso dei cataloghi. Supponiamo che un insieme di case editrici abbia ad una certa data pubblicato un catalogo di tutti i libri pubblicati dalla casa fino a quel momento. Alcune case editrici hanno inserito il catalogo stesso come libro citato nel catalogo, altre no. Supponiamo ora di voler costruire un nuovo catalogo che descriva tutti i cataloghi che non citano se stessi. Un tale catalogo non può esistere.

Infatti, supponiamo per assurdo che esista un catalogo C che risponda alla definizione data. Se C non cita se stesso, allora C non descrive tutti i cataloghi che non citano se stessi. Invece, se C cita se stesso non risponde più alla definizione.

In termini più formali il paradosso può essere così formulato. Si consideri l'insieme A di tutti quegli insiemi X tali che X non è un elemento di X. Allora se A è un elemento di A, per definizione A non è un elemento di A; viceversa,

se A non è un elemento di A, sempre per definizione, A è un elemento di A. Si arriva così al paradosso che A è un elemento di A se e soltanto se A non è un elemento di A.

I paradossi mettevano in crisi la possibilità di poter fondare tutta la matematica, senza introdurre contraddizioni. Per superare questa situazione furono proposte essenzialmente tre soluzioni: il logicismo, il formalismo, l'intuizionismo. Nel seguito ci limiteremo a considerare il formalismo, che fu proposto da Hilbert, poiché è la proposta più strettamente collegata alla teoria della calcolabilità.

Hilbert affermò che l'unico modo per poter superare la crisi dei fondamenti consisteva nell'identificare l'esistenza matematica con la non contradditorietà. Gli enti matematici devono essere definiti formalmente mediante assiomi e i metodi di dimostrazione devono essere ridotti a schemi di deduzioni formali. Un tale approccio avrebbe garantito, secondo Hilbert, l'impossibilità di introdurre contraddizioni e quindi la correttezza del ragionamento matematico. Naturalmente gli assiomi e le regole deduttive dovevano essere scelte in modo tale da poter definire, in particolare, le teorie matematiche classiche. Pertanto il programma hilbertiano si basava sul presupposto che data una proposizione formale, esiste un algoritmo che può decidere la falsità o verità della proposizione stessa. Ma questo presupposto del programma di Hilbert fu dimostrato non vero da Gödel.

Gödel provò che non poteva esistere un algoritmo per decidere la verità o la falsità di ogni affermazione riguardante l'aritmetica. Questo teorema, noto come il teorema di incompletezza dell'aritmetica, provava che in un sistema formale non contraddittorio comprendente l'aritmetica esistono proposizioni aritmetiche la cui verità o falsità non è dimostrabile entro il sistema stesso.

Possiamo interpretare il risultato di Gödel come la costruzione effettiva di una funzione non calcolabile, ove, in questo caso, il calcolo riguarda lo stabilire la verità o la falsità di una affermazione.

L'esistenza di funzioni non calcolabili conferma quindi la necessità di definire precisamente il concetto di procedura effettiva e di individuare modelli di calcolo in grado di calcolare tutte le funzioni calcolabili. Lo stesso Gödel propose nel 1934 la classe delle funzioni ricorsive primitive che erano state per la prima volta introdotte da Dedekind alla fine dell'Ottocento. Qualche anno più tardi veniva definita (Herbrand, Gödel, Kleene) la classe delle funzioni ricorsive. Il fatto che le funzioni ricorsive potessero rappresentare adeguatamente la nozione di funzione calcolabile fu, per la prima volta, affermato

esplicitamente nel 1936 da Church, che tenne conto di alcuni risultati che Kleene aveva appena provato. Kleene infatti aveva mostrato che le funzioni ricorsive coincidevano con le funzioni lambda-definibili (le funzioni lambda-definibili, ovvero il lambda-calcolo, costituiscono un altro formalismo che era stato introdotto per caratterizzare l'effettiva computabilità). L'equivalenza tra questi due formalismi indusse Church a pensare che il concetto intuitivo di funzione calcolabile potesse coincidere con il concetto di funzione ricorsiva e lambda-definibile e a formulare la tesi che porta il suo nome.

Parte IV

Sistema di elaborazione

- 8. Sistema di elaborazione: architettura**
- 9. Sistema di elaborazione: software di base**
- 10. Linguaggi di programmazione**

L

8.

Sistema di elaborazione: architettura

In questo capitolo ed in quello successivo illustriamo gli aspetti più importanti di un sistema di elaborazione, descrivendo nel primo l'architettura hardware nota come "architettura di Von Neumann" ed il suo funzionamento elementare; e nel secondo gli aspetti più legati al software e quindi al funzionamento complessivo dell'elaboratore.

Nel primo paragrafo di questo capitolo viene riconsiderato lo schema funzionale dell'elaboratore già visto nell'introduzione. Nel secondo paragrafo esaminiamo l'elaboratore come un sistema complesso, corredata di componenti ausiliarie che garantiscono un funzionamento più potente ed efficiente. Presentiamo anche le problematiche della interconnessione di elaboratori diversi. Nell'ultimo paragrafo infine, descriviamo le forme di rappresentazione delle informazioni elementari come numeri e caratteri.

1. LA STRUTTURA DELL'ELABORATORE

Riprendiamo lo schema funzionale dell'elaboratore che è già stato introdotto nel capitolo iniziale del libro (vedi fig. 1), al fine di presentare una descrizione più dettagliata delle parti componenti del sistema di elaborazione e delle varie funzioni che esse svolgono.

In questo schema si possono individuare le seguenti unità:

- *memoria*, che ha lo scopo di memorizzare programmi, dati di ingresso, risultati intermedi e finali;
- *unità logico aritmetica*, unità componente del processore, che ha la funzione di eseguire calcoli logici ed aritmetici;

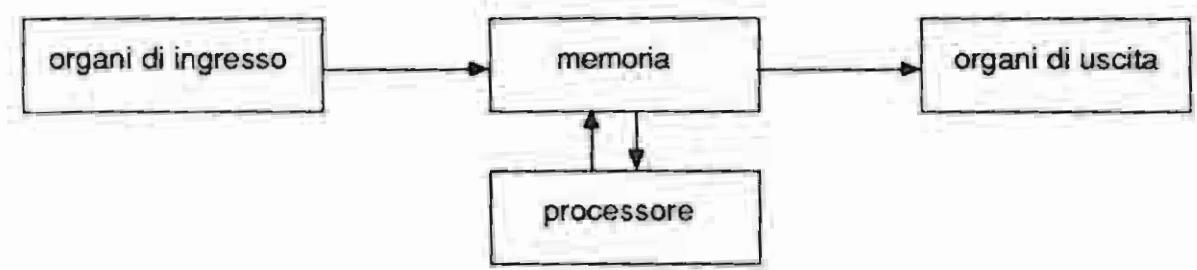


Fig. 1 - Schema funzionale dell'elaboratore

- *unità di controllo*, unità componente del processore, che governa il funzionamento complessivo del sistema di elaborazione attraverso il controllo dell'esecuzione delle singole istruzioni di un programma. Come vedremo più avanti, le istruzioni del programma vengono trasferite, una alla volta, dalla memoria all'unità di controllo, che ne garantisce l'esecuzione inviando comandi opportuni alle altre unità del sistema;
- *unità di ingresso e uscita*, che svolgono le funzioni di interfaccia tra l'elaboratore e l'esterno.

Nei paragrafi successivi analizziamo in maggior dettaglio le singole unità, presentando anche il funzionamento elementare dell'elaboratore.

1.1. La memoria centrale

In genere con il termine memoria si intende un qualunque dispositivo in cui si possano immettere e conservare informazioni (programmi e dati) e da cui sia possibile estrarli. La memoria viene spesso distinta in *memoria centrale* e *memoria secondaria o ausiliaria*. In questo paragrafo descriviamo la memoria centrale, mentre le memorie ausiliarie verranno trattate nel successivo par. 2. Tuttavia si può subito indicare una delle differenze principali tra questi due tipi di memoria. Le dimensioni della memoria centrale sono limitate, ma la velocità di accesso è molto elevata rispetto a quella delle memorie ausiliarie. Queste ultime, di dimensioni teoricamente illimitate, hanno però tempi di accesso notevolmente superiori.

L'informazione elementare prende il nome di *bit (binary digit)* e può assumere i valori zero od uno. La realizzazione fisica di tale informazione

elementare in memoria è ottenuta mediante l'uso di dispositivi fisici a due stati. Una qualunque informazione viene rappresentata in memoria mediante sequenze di bit.

Una sequenza di 6 (oppure 8) bit prende il nome di *carattere (byte)* e quindi permette di memorizzare 2^6 (o 2^8) diversi elementi. La memoria di un elaboratore elettronico è costituita da un insieme di celle o locazioni costituite da n bit, dove n è un valore *fisso* per ogni elaboratore. Esistono elaboratori con celle di 4, 8, 16, 32, 36, 64 ... bit. Il contenuto di una cella di memoria prende il nome di *parola*.

Ad ogni cella di memoria è associato un *indirizzo* che è il numero d'ordine della cella nell'intera memoria. Ad ogni cella corrisponde pertanto la coppia <indirizzo, valore>. La funzione che associa ad un dato indirizzo l'informazione in esso contenuta è una funzione non invertibile, nel senso che ad un valore possono corrispondere più indirizzi.

Accedere ad un dato significa selezionare mediante l'indirizzo la cella in cui esso è memorizzato e prelevarne il valore. Analogamente, la memorizzazione di un dato richiede di stabilire l'indirizzo della cella in cui si intende introdurlo.

Se una cella della memoria è formata da un numero di bit multiplo di sei o di otto, significa che in essa sono contenuti più byte ed in genere è possibile accedere direttamente sia alla parola che a ciascun byte in essa contenuto.

Una caratteristica importante della memoria è il tempo di accesso, cioè il tempo che intercorre tra l'istante in cui si richiede un'informazione e l'istante in cui tale informazione è disponibile. La memoria centrale viene anche detta ad accesso diretto (*Random Access Memory: RAM*) o uniforme, poiché il tempo di accesso è costante, indipendentemente dall'indirizzo della cella a cui si vuole accedere. Il tempo di accesso è, nell'attuale tecnologia, dell'ordine di alcune centinaia di nanosecondi.

Un altro parametro che caratterizza la memoria centrale, è il numero complessivo di celle di cui essa è formata, e può andare dalle decine di migliaia alle decine di milioni.

Un tipo particolare di memoria centrale è costituito dalle memorie a sola lettura (*Read Only Memory: ROM*) nelle quali non è possibile memorizzare nuovi valori nelle celle. Le memorie ROM sono perciò utilizzate specificamente per contenere programmi e dati che non vengono mai modificati durante l'elaborazione.

Si noti che le memorie ROM sono permanenti, mentre per quelle RAM il contenuto delle celle viene alterato ad ogni interruzione della corrente di alimentazione dell'elaboratore.

1.2. L'unità centrale

Facendo ancora riferimento allo schema funzionale mostrato nella fig. 1, per unità centrale (processore o *Central Processing Unit: CPU*) si intende l'insieme dell'unità di controllo e dell'unità aritmetica e logica. La distinzione tra Unità di Controllo (*Control Unit: CU*) e Unità Aritmetica e Logica (*Arithmetic and Logic Unit: ALU*) ha un significato puramente funzionale. In realtà l'evoluzione tecnologica ha portato ad integrare queste due unità in un unico modulo, aumentando allo stesso tempo le sue capacità di calcolo.

Come si vede nella fig. 1, l'unità di controllo dell'elaboratore può accedere solo alle informazioni che sono contenute nella memoria centrale. Compito dell'unità di controllo è infatti quello di reperire dalla memoria le istruzioni di un programma, interpretarle, utilizzando un organo di decodifica, e farle eseguire regolando le azioni delle varie unità. A queste vengono inviati segnali di attivazione, prelevando dalla memoria i dati, trasferendoli nell'unità che deve operare e riportando in memoria gli eventuali risultati ottenuti.

L'unità di controllo stabilisce, in base alle condizioni che vengono poste dal programma, quale istruzione debba essere eseguita in un determinato istante. Inoltre essa utilizza le unità di ingresso e di uscita a seconda che il programma richieda la lettura di dati dall'esterno o la scrittura di risultati.

L'unità aritmetica e logica è costituita da moduli capaci di eseguire un numero limitato di operazioni. Nel caso di piccoli elaboratori queste operazioni sono poche e semplici, e diventano via via più numerose con il crescere della potenza dell'unità centrale e con l'evoluzione della tecnologia impiegata nella sua costruzione.

Dall'unità di controllo l'unità aritmetica e logica riceve il comando da eseguire, insieme con l'operando, o gli operandi, su cui l'operazione deve essere effettuata.

Nell'unità centrale si trovano alcuni registri che svolgono funzioni fondamentali per l'esecuzione dei programmi. Questi sono dei dispositivi formati da sequenze di bit, in genere almeno tanti quanti sono quelli delle celle di memoria.

In generale tutti gli elaboratori hanno disponibili i seguenti registri:

- *contatore di programma (Program Counter: PC)*, il cui contenuto ad ogni istante indica l'indirizzo della successiva istruzione da eseguire, esso ha perciò la funzione di guidare il flusso dell'esecuzione di un programma;
- *registro istruzione (Instruction Register: IR)*, che contiene l'istruzione che deve essere interpretata e poi eseguita;

- *accumulatori*, che contengono gli operandi di una data istruzione e, alla fine dell'esecuzione dell'operazione, il risultato;
- *parola di stato (Program Status Word: PSW)* i cui bit forniscono particolari informazioni circa l'ultima istruzione eseguita: ad esempio, se il risultato di un'operazione aritmetica sia zero, quale ne sia il segno, o se sia stato generato un riporto. In ogni elaboratore mediante istruzioni elementari è possibile verificare il valore di questi bit e operare conseguentemente.

Possiamo a questo punto riassumere la struttura dell'unità di controllo attraverso la fig. 2. Non interessa approfondire qui ulteriormente gli aspetti connessi allo schema funzionale delle varie unità; vogliamo comunque sottolineare che esiste la necessità di dotare l'unità di controllo di una funzione di temporizzazione, che sia in grado di garantire l'esecuzione delle varie azioni elementari in modo coordinato.

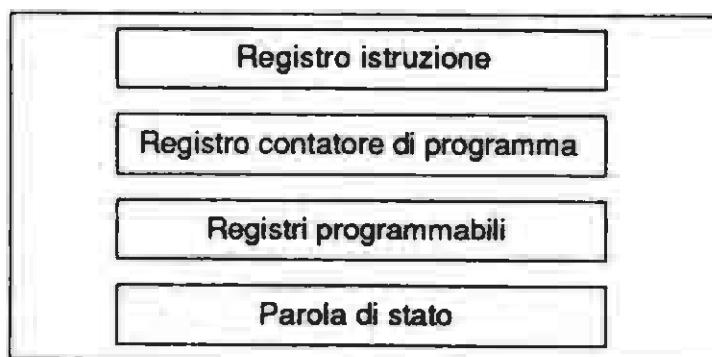


Fig. 2 - Struttura dell'unità di controllo

1.3. Il funzionamento elementare dell'elaboratore

Ogni elaboratore è in grado di interpretare ed eseguire un insieme finito di istruzioni elementari, ciascuna delle quali si riferisce ad un'operazione di base che una particolare unità può compiere. Questo insieme di istruzioni costituisce il *linguaggio macchina* dell'elaboratore considerato.

Un programma eseguibile da un dato elaboratore è quindi un insieme ordinato di queste istruzioni, che vengono eseguite, una alla volta, sequenzialmente, secondo l'ordine specificato dal programma stesso.

Per descrivere il funzionamento elementare di un elaboratore ci serviremo di un linguaggio macchina semplificato, a scopo didattico, utile per introdurre i meccanismi di funzionamento validi per ogni elaboratore.

In questo linguaggio ogni istruzione ha una struttura semplice del tipo:

codice istruzione | operando

e si assume che essa occupi una sola locazione di memoria.

Nella fig. 3 viene presentata la sintassi e la semantica delle istruzioni, facendo uso delle seguenti abbreviazioni:

IND Indirizzo di cella di memoria

ACC Accumulatore

\leftarrow Trasferimento

UL Unità di lettura

US Unità di scrittura

(x) Contenuto di x.

		<i>Sintassi</i>	<i>Semantica</i>
<i>Codice</i>	<i>Operando</i>		
<i>Istruz.</i>			
1	IND	(ACC)	\leftarrow (ACC) + (IND)
2	IND	(ACC)	\leftarrow (ACC) - (IND)
3	IND	(ACC)	\leftarrow (IND)
4	IND	(IND)	\leftarrow (ACC)
5	IND	(IND)	\leftarrow (UL)
6	IND	(US)	\leftarrow (IND)

Fig. 3 - Istruzioni di un linguaggio macchina

La prima istruzione, l'istruzione di addizione, consente di sommare il contenuto del registro accumulatore ACC con il contenuto della locazione di memoria di indirizzo IND, e di ottenere il risultato come nuovo contenuto del registro accumulatore. Da un punto di vista matematico un'operazione di addizione è un'operazione a due operandi. In questo caso l'esecuzione dell'o-

perazione prevede che uno dei due operandi sia stato precedentemente caricato nell'accumulatore, mediante un'istruzione specifica, come quella riportata in figura con il codice 3. Ciò costituisce un esempio di come si possano realizzare operazioni a più operandi mediante istruzioni ad un solo operando.

L'istruzione di sottrazione è analoga alla addizione. Quelle di caricamento e di memorizzazione consentono rispettivamente il trasferimento del contenuto di una locazione di memoria come nuovo contenuto del registro accumulatore, e viceversa.

Le istruzioni di lettura e scrittura consentono rispettivamente il trasferimento del dato acquisito attraverso un'unità di lettura come nuovo contenuto della locazione di memoria specificata dall'indirizzo IND, e il trasferimento del contenuto della locazione di memoria di indirizzo IND all'esterno, attraverso un'unità di stampa.

Esempio. Supponiamo di voler leggere, cioè introdurre, due dati in memoria, sommarli e stampare il risultato. Un possibile programma è riportato in fig. 4, dove accanto ad ogni istruzione è stata mostrata la sua semantica, secondo la notazione precedentemente introdotta. Si noti che in questo caso usiamo una cifra decimale per il codice dell'istruzione e tre cifre decimali per gli indirizzi degli operandi e delle istruzioni stesse.

Il programma occupa sei celle di memoria, una per ciascuna delle sue istruzioni, a partire da una cella fissata, quella di indirizzo 100.

Volendo ora eseguire questo programma a partire dalla sua prima istruzione, il valore 100 deve essere posto nel registro PC contatore di programma.

Esaminiamo ora in dettaglio il funzionamento elementare dell'elaboratore durante l'esecuzione di una generica istruzione del programma visto.

L'unità di controllo accede all'indirizzo di memoria specificato dal registro PC, e trasferisce il suo contenuto nel registro IR. Qui l'istruzione viene decodificata analizzando il codice istruzione. Questa fase prende il nome di *alimentazione* (*fetch*). Terminata questa fase, il contatore di programma viene incrementato di uno, in modo che il suo contenuto indichi l'indirizzo della successiva istruzione da eseguire. Nell'esempio in esame il contenuto di PC sarà quindi, a questo punto, 101.

La fase successiva prende il nome di *esecuzione* (*execute*). Una volta riconosciuta l'istruzione da eseguire, l'unità di controllo attiva i moduli delle unità coinvolte. Se l'istruzione contiene un operando viene eseguito un accesso in memoria nell'indirizzo specificato dall'istruzione. Se l'istruzione è un'istruzione aritmetica, ad esempio la quarta del programma della fig. 4, l'operazione

<i>Indirizzo</i>	<i>Istruzione</i>	<i>Semantica</i>
100	5 000	(000) \leftarrow (UL)
101	5 001	(001) \leftarrow (UL)
102	3 000	(ACC) \leftarrow (000)
103	1 001	(ACC) \leftarrow (ACC) + (001)
104	4 002	(002) \leftarrow (ACC)
105	6 002	(US) \leftarrow (002)

Fig. 4 - Esempio di programma in linguaggio macchina

viene eseguita, il risultato resta in ACC e la fase termina. Se l'istruzione è un'istruzione di trasferimento in memoria, come la quinta dell'esempio, viene eseguito l'accesso in memoria, il trasferimento del contenuto di ACC e quindi la fase termina.

Come già detto, un programma memorizzato viene eseguito istruzione dopo istruzione, secondo una sequenza temporale corrispondente alla sequenza fisica.

Spesso però è necessario modificare l'ordine strettamente sequenziale della esecuzione delle istruzioni. A tale scopo si fa uso di particolari istruzioni, dette di *salto*, che consentono di indicare l'istruzione da eseguire (generalmente diversa dalla successiva) attraverso l'indirizzo in cui essa è memorizzata. L'esecuzione di un'istruzione di salto modifica quindi il valore contenuto nel registro PC, che diviene l'indirizzo dell'istruzione a cui si intende trasferire il controllo: quella da eseguire nel passo successivo.

È spesso anche utile poter disporre di istruzioni di *salto condizionato*. Ad esempio, esistono istruzioni che realizzano il salto solo se il contenuto del registro ACC sia positivo oppure nullo. Si noti che queste condizioni vengono generalmente verificate analizzando i bit del registro PSW.

La fig. 5 descrive le istruzioni di controllo disponibili nel nostro linguaggio secondo la forma già usata precedentemente.

La prima è un'istruzione di salto incondizionato la cui esecuzione prevede il trasferimento dell'indirizzo IND come nuovo contenuto del registro PC. La seconda istruzione è di salto condizionato. L'Esecuzione di questa istruzione modifica il contenuto del registro PC solo se il contenuto del registro accumulator ACC è positivo o nullo; altrimenti il registro PC non viene modificato.

Codice *Operando*
Istruz.

7	IND	(PC) \leftarrow IND
8	IND	se (ACC) \geq 0 allora (PC) \leftarrow IND
9	STOP	(ferma l'esecuzione)

Fig. 5 - Altre istruzioni di un linguaggio macchina

Si noti che è stata anche introdotta l'istruzione STOP necessaria per concludere l'esecuzione di un programma.

Esempio. Consideriamo ora il seguente programma, supposto memorizzato a partire dalla cella 000 (v. fig. 6).

<i>Indirizzo</i>	<i>Istruzione</i>	<i>Semantica</i>
000	5 100	(100) \leftarrow (UL)
001	5 101	(101) \leftarrow (UL)
002	3 100	(ACC) \leftarrow (100)
003	2 101	(ACC) \leftarrow (ACC) - (101)
004	8 007	Se (ACC) \geq 0 vai a 007
005	6 101	(US) \leftarrow (101)
006	9	STOP
007	6 100	(US) \leftarrow (100)
008	9	STOP

Fig. 6 - Un altro semplice programma

Se il programma è mandato in esecuzione a partire dalla cella 000, esso procede inizialmente a leggere due dati, successivamente li confronta e stampa il maggiore dei due; questo è ottenuto sfruttando l'istruzione di salto condizionato.

Si noti ancora che invece di utilizzare due istruzioni di STOP avremmo potuto inserire nella cella 006 una istruzione di salto incondizionato all'indirizzo 008.

2. IL SISTEMA DI ELABORAZIONE

Abbiamo parlato finora del nucleo centrale dell'elaboratore, ed in particolare dell'unità centrale (di controllo e di calcolo) e della memoria. Tutti gli altri elementi che costituiscono il sistema di elaborazione prendono il nome di *unità periferiche*, proprio per sottolineare il loro ruolo di completamento delle funzioni di base dell'unità centrale. Con il termine unità periferiche viene fatto riferimento a unità tra loro molto diverse. Possiamo raggrupparle in classi, in relazione alle loro funzioni:

- *unità di ingresso e uscita*: mediante le quali è possibile introdurre dati e programmi nella memoria dell'elaboratore e prelevare da essa valori e risultati di elaborazioni;
- *unità di memoria secondaria*: che permettono di archiviare programmi e dati.

Esistono poi altre unità di tipo particolare, difficilmente classificabili in modo unico, impiegate solo per specifiche applicazioni come, ad esempio, l'acquisizione di dati sperimentali.

Nei paragrafi successivi presenteremo alcune di queste unità, facendo riferimento più alle loro caratteristiche funzionali che alla loro struttura fisica in quanto questa, essendo strettamente legata alla tecnologia costruttiva impiegata, è estremamente variabile nel tempo.

2.1. Le unità di memoria ausiliaria

Si è visto come la memoria centrale sia destinata a immagazzinare programmi e dati, immediatamente disponibili per il trattamento da parte dell'unità centrale. È comunque necessario disporre di unità di memoria ausiliaria o secondaria per archiviare programmi e dati; questi dovranno essere trasferiti in memoria centrale ogni qualvolta li si voglia elaborare.

- Le unità di memoria secondaria consentono di memorizzare una quantità teoricamente illimitata di informazioni e di poter trasferire fisicamente dati e programmi da un luogo ad un altro su supporti comodi e affidabili. Vengono a tale scopo utilizzati supporti magnetici e le relative unità di scrittura e lettura.

Ci limitiamo qui a considerare solo alcune unità di memoria ausiliaria, unità a nastro e unità a disco, che rappresentano due precise tipologie, di organizzazione delle informazioni memorizzate. In entrambi i casi il principio fisico di memorizzazione, o meglio di registrazione, delle informazioni è il medesimo: magnetizzare un'area del supporto di registrazione.

Nastri. Il nastro magnetico è un nastro di materiale plastico avvolto su bobine di formato e lunghezza variabile. Su queste bobine, che possono immagazzinare le informazioni solo in forma sequenziale, si possono registrare o rilevare informazioni per mezzo di apposite unità, dette *unità a nastro*.

L'informazione viene registrata sul nastro magnetizzando delle areole disposte su canali o piste, parallele al senso di scorrimento del nastro stesso. Un'area magnetizzabile rappresenta un bit. Vedi fig. 7.

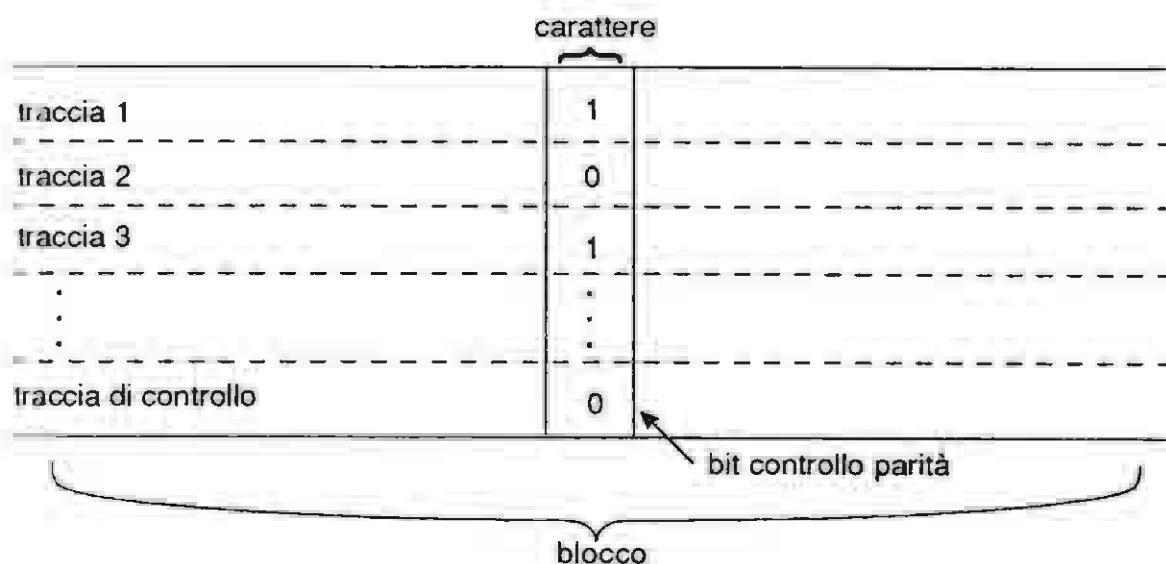


Fig. 7 - Organizzazione delle informazioni su nastro

La superficie del nastro magnetico può essere quindi immaginata come suddivisa in un certo numero di piste longitudinali, che prendono il nome di *tracce*, di lunghezza pari all'intero nastro e di larghezza sufficiente a poter contenere la memorizzazione di un bit. L'insieme dei bit che si trovano sulla stessa perpendicolare all'asse del nastro, in numero pari al numero delle tracce, rappresenta un singolo carattere in un opportuno codice.

Per ogni carattere una delle tracce viene generalmente utilizzata come controllo di parità: contiene cioè un 1 o uno 0 in modo che il numero totale degli 1 di quel carattere sia sempre pari. In questo modo un numero dispari di 1 in un carattere evidenzia un errore di registrazione. Il numero di tracce è pertanto 7 o 9, a seconda che si usino 6 o 8 bit per carattere.

Oltre al controllo di parità esistono controlli che avvengono confrontando ciò che su nastro è stato scritto con ciò che si voleva scrivere (rilettura) ed anche controlli alla fine di ogni blocco di caratteri, effettuati aggiungendo anche qui

controlli di parità, in questo caso longitudinali.

La velocità di lettura e registrazione varia, a seconda dei tipi di unità, da 20.000 a 120.000 caratteri al secondo; la capacità di una singola bobina può raggiungere i 12.000.000 caratteri.

È infine importante notare che il tempo di accesso di un'informazione su un nastro dipende strettamente dalla posizione dell'informazione stessa, tenendo conto della rigida organizzazione sequenziale della memorizzazione su nastro.

Dischi. Il disco magnetico è un disco metallico (generalmente una lega di alluminio per i dischi cosiddetti rigidi), oppure un disco di materiale plastico (per i dischi flessibili: *floppy disk*) ricoperto su entrambe le facce da ossido di ferro. I dischi rigidi possono anche venire utilizzati sovrapposti (*disk pack*) e distanziati l'uno dall'altro, montati su un perno verticale che garantisce una rotazione uniforme dell'intero insieme di dischi.

I dischi ruotano a velocità costante e scorrono quindi sotto delle testine di registrazione che attivano la lettura e la scrittura quando necessario.

Le informazioni su un disco vengono memorizzate su tracce o piste concentriche, individuate con un indirizzo, che sono presenti su ogni faccia del disco in numero di qualche centinaio. A loro volta le tracce vengono suddivise in settori separati da zone di interblocco che non contengono alcuna informazione.

Sui dischi le informazioni possono essere registrate o lette in due modi diversi: sequenzialmente, come per il nastro magnetico, oppure con accesso diretto, che consente l'individuazione diretta delle informazioni che interessano, senza l'accesso a tutte le registrazioni che le precedono. In quest'ultimo caso il tempo di accesso ad una data informazione può essere considerato indipendente dalla posizione dell'informazione stessa. Per individuare infatti una data informazione o una locazione del disco, la testina dell'unità disco prima si posiziona sulla pista contenente il settore interessato, poi sul settore; a questo punto avviene l'accesso.

Attualmente si hanno due tipi di *unità a disco*: quella a dischi fissi e quella a dischi mobili o asportabili. La capacità varia da tipo a tipo. Per il tipo asportabile, più diffuso, si può avere una capacità dell'ordine del milione di caratteri e una velocità di trasferimento dei dati dell'ordine delle decine di migliaia di caratteri al secondo. Per i dischi rigidi fissi la capacità va dall'ordine delle decine alle migliaia di milioni di caratteri e la velocità di trasferimento è dell'ordine delle centinaia di migliaia di caratteri al secondo.

2.2. Le unità di ingresso e uscita

Come per le unità di memoria ausiliaria anche in questo caso ci limiteremo a presentare solo alcune delle unità attualmente più diffuse e che consentono di descrivere le funzioni tipiche della trasmissione di informazioni tra l'utente e l'elaboratore; in particolare considereremo i terminali video-tastiera e le stampanti.

Va comunque ricordato che esiste un grandissimo numero di altre unità periferiche di ingresso ed uscita, utilizzabili per applicazioni speciali, e che sono spesso molto differenti tra loro sia funzionalmente che per la struttura fisica e tecnologica. Citiamo come esempi i video ad alta risoluzione, i tracciatori di curve (*plotter*), i digitalizzatori, i lettori ottici di documenti.

Sempre nell'ambito delle unità periferiche di ingresso e uscita va anche detto che esiste la possibilità di collegare ad un elaboratore unità di tipo analogico (strumenti di laboratorio, ed apparati ricetrasmettenti) per acquisizione dati. Affinché ciò sia possibile è necessario disporre di moduli particolari, detti interfacce, che rendono compatibili i segnali che tali apparecchiature emettono o ricevono e quelli rispettivamente ricevibili o emettibili dall'elaboratore specifico.

Video e tastiera. La tastiera consente all'utente di inviare comandi e dati all'elaboratore, mentre il video svolge una duplice funzione: quella di unità di uscita e quella di visualizzare le informazioni inserite.

L'informazione che appare sul video di tipo più semplice è composta da simboli appartenenti ad un insieme fissato di simboli alfabetici, numerici e speciali. I video di questa classe, poco costosi, sono chiamati alfanumerici per distinguerli dai video grafici che possono invece visualizzare, tramite una matrice di punti, anche caratteri e disegni di qualsiasi forma.

L'informazione trasferita consiste in caratteri rappresentati in un codice prefissato. Alcuni di questi caratteri sono alfanumerici, altri, detti di controllo, hanno funzioni operative speciali.

Stampanti. Tali unità, esclusivamente di uscita, sono di due tipi: parallele e seriali. Quelle parallele stampano una riga, cioè più caratteri per volta, su moduli di carta, trascinati meccanicamente. Il principio di stampa è generalmente quello della stampa ad impressione. La loro velocità si misura in righe al minuto, e può superare le 1000 righe.

Altre unità stampanti sono quelle di tipo seriale che stampano un carattere per

volta, serialmente. Esse spesso coincidono con l'unità di scrittura dei terminali telescriventi ed hanno velocità molto inferiori alle stampanti parallele.

2.3. Interfacciamento di unità periferiche

Gli elementi che compongono un sistema di elaborazione sono estremamente eterogenei tra loro: la logica di funzionamento della unità centrale è, ad esempio, molto diversa da quella di una stampante o di una tastiera. Tra tipi diversi di unità fluiscono nel tempo un gran numero di informazioni appartenenti a due tipi fondamentali: dati che dalla unità di ingresso vanno alla memoria o da questa alle unità di uscita, e informazioni di controllo che dall'unità di controllo vanno alle unità periferiche. Perché tale flusso di informazioni sia possibile è necessario operare una traduzione di segnali diversi. Tali trasformazioni vengono eseguite da unità che prendono il nome di *interfacce* in quanto permettono di collegare, o come si dice, interfacciare, una specifica unità periferica con una specifica unità centrale di un dato elaboratore. In generale, quindi il collegamento tra un'unità periferica ed un'unità centrale avviene tramite una *porta* e una *interfaccia* vera e propria. Inoltre il collegamento è governato da una *unità di controllo locale*.

La porta a cui viene collegata un'unità periferica è un'area di memoria accessibile sia all'unità centrale che all'unità periferica stessa. Attraverso la porta i programmi inviano o ricevono informazioni dall'esterno.

Il circuito di interfaccia, collega la porta all'unità periferica adattando tra loro i segnali che vengono scambiati. Le interfacce possono essere del tipo seriale o parallelo. Nel primo caso l'informazione è trasmessa (ad esempio un carattere per volta) lungo una singola linea con sequenze di 8 bit. Nel secondo caso l'interfaccia prevede più linee che consentono di trasmettere in parallelo tutti i bit di un carattere. Tipicamente, le interfacce parallele, più costose, sono usate per collegare unità che hanno un elevato tasso di trasmissione, come ad esempio i dischi.

Esaminiamo ora brevemente quali possono essere i supporti fisici tramite i quali avviene il trasferimento delle informazioni, in particolare tra unità periferiche e unità centrale.

Per il collegamento sono in genere usati dei cavi coassiali. Questi supporti, molto affidabili e veloci, ma costosi, vengono in genere utilizzati per brevi distanze, ad esempio nell'ambito di uno stesso locale, di un edificio o di un gruppo di edifici adiacenti. Per distanze maggiori il mezzo di trasmissione più

usato, è la linea telefonica. Questa può essere una *linea commutata*, cioè una normale linea che passa attraverso le centrali telefoniche e che può essere alternativamente utilizzata per la trasmissione dei dati e per la trasmissione della voce, oppure una *linea dedicata*, cioè una particolare linea che congiunge direttamente due sistemi.

Comunque sia stata scelta la linea telefonica di collegamento, per connettere due unità è necessario frapporre tra ciascuna di esse e la linea telefonica, un dispositivo che renda possibile la trasmissione dei bit che costituiscono i dati da trasmettere. Tali dispositivi prendono il nome di *modem* (modulatore/demodulatore), operano sempre a coppie svolgendo uno le funzioni inverse all'altro, ed hanno diverse caratteristiche di velocità e di modalità di trasmissione.

2.4. Collegamenti di elaboratori in rete

Si è visto che per collegare un'unità periferica ad un elaboratore è sufficiente che l'elaboratore riconosca i segnali ricevuti.

Volendo invece collegare due elaboratori è necessario disporre anche di un insieme di programmi che regolino lo scambio di informazioni e interpretino i dati ricevuti. Questi programmi gestiscono il cosiddetto *protocollo di collegamento*, cioè l'insieme di regole che governano l'inizio, la fine e le modalità di trasmissione dei dati.

La trasmissione può avvenire un carattere alla volta (trasmissione di tipo *asincrono*) ognuno dei quali è preceduto e seguito da due segnali che hanno la funzione di indicare l'inizio e la fine della trasmissione. Questo è il caso più semplice ed economico, sia a livello di software che di hardware, ed è quello comunemente utilizzato per collegare un'unità periferica con un elaboratore.

Più complessa, ma anche più efficiente, è la trasmissione di tipo *sincrono*, in cui vengono inviate sequenze di caratteri. In questo caso ad ogni blocco di dati viene premesso un segnale di sincronismo, mentre viene posposto un segnale di fine dati, insieme a segnali di controllo che hanno lo scopo di verificare la correttezza della trasmissione. Nel caso di trasmissione di tipo sincrono è necessario anche un orologio che garantisca la sincronizzazione della trasmissione.

Il protocollo di collegamento gestisce anche queste diverse modalità di trasmissione dei dati e tiene conto delle differenze dei diversi elaboratori che si vogliono collegare.

Cerchiamo ora di schematizzare i diversi tipi di collegamento possibili.

Un primo tipo consiste nel collegare un elaboratore con un altro elaboratore più potente. In questo caso è necessario che l'elaboratore meno potente sia dotato di un software che gestisca il collegamento e che sia tale da simulare uno dei terminali riconosciuti dall'elaboratore centrale. Tale software prende in genere il nome di *emulatore di terminali*.

Si osservi che questi tipi di collegamento sono sempre asimmetrici, nel senso che i due elementi che vengono connessi svolgono ruoli diversi, essendo uno costantemente controllato dall'altro. E opportuno sottolineare che questa è l'unica soluzione possibile nel caso di un terminale non intelligente che venga connesso con un elaboratore, ma non è l'unica soluzione nel caso di collegamento tra due elaboratori.

Un secondo tipo di collegamento è quello simmetrico tra due elaboratori simili; esso è possibile se si dispone del software necessario per ricevere e trasmettere dati, cioè per gestire il livello minimo del protocollo di trasmissione dei dati. L'utente può in questo caso utilizzare tutta la potenzialità del suo elaboratore e, in aggiunta a ciò, inviare richieste all'elaboratore remoto.

Per molte applicazioni è spesso importante poter collegare tra loro anche più di due elaboratori in modo da ottenere un'effettiva distribuzione delle elaborazioni. In tal caso anche la soluzione appena illustrata può risultare insufficiente.

Si fa allora ricorso ad un terzo tipo di collegamento che è quello delle reti di elaboratori, intendendo con tale termine un collegamento tra diversi elaboratori, tutti dotati di capacità di elaborazione autonoma, anche se eventualmente con diverse potenzialità.

Le reti sono fondamentalmente di due tipi: geografiche, se collegano tra di loro elaboratori posti a grandi distanze, locali se collegano elaboratori situati a non più di 10 Km. Le reti geografiche utilizzano in genere le linee telefoniche come supporti fisici per la trasmissione, mentre le reti locali impiegano cavi coassiali.

3. RAPPRESENTAZIONE DELL'INFORMAZIONE

I dati su cui generalmente si opera possono appartenere ad insiemi qualsiasi e devono essere opportunamente codificati per poter essere trattati automaticamente da un elaboratore. In questo paragrafo ci soffermiamo sulla rappresentazione in memoria centrale delle informazioni numeriche e dei caratteri.

3.1. Sistemi di numerazione e algoritmi di conversione

Un numero naturale, cioè un numero intero assoluto, è un ente matematico che può essere rappresentato mediante una stringa di caratteri, detti *cifre*, di un fissato alfabeto. Si deve quindi aver cura di distinguere tra un numero naturale ed una sua rappresentazione. A tale scopo si parlerà di numerale per intendere una delle possibili rappresentazioni di un numero.

Esistono evidentemente molte forme di rappresentazione per i numeri naturali; quelle più note sono la rappresentazione romana e la rappresentazione araba.

La prima è basata su un principio puramente additivo. Ad esempio, ricordando che i simboli I, V, X, L, C, D, M rappresentano rispettivamente i valori crescenti uno, cinque, dieci, cinquanta, cento, cinquecento e mille, la sequenza

MDCXLI

rappresenta il valore: mille + cinquecento + cento - dieci + cinquanta + uno, cioè milleseicentoquarantuno.

Si noti come il principio additivo sia inteso in senso algebrico, cioè se il simbolo corrispondente ad un dato valore non rispetta l'ordinamento crescente originario di tutti i simboli utilizzati, allora questo simbolo dà un contributo negativo.

Questo sistema di rappresentazione richiede sempre nuovi simboli man mano che i numeri da rappresentare crescono, ed inoltre in esso risulta complesso rappresentare numeri grandi ed effettuare calcoli complessi. Il principale limite di tale sistema risulta quindi essere l'enorme difficoltà di calcolo.

Il secondo sistema di rappresentazione (originato da popolazioni indù e trasferitoci dagli arabi nel Medio Evo) è invece basato su un principio posizionale, secondo cui ogni simbolo o cifra del sistema assume valori diversi a seconda della posizione che occupa in una data sequenza. Questo richiede l'introduzione di un simbolo particolare per rappresentare lo zero, che non era rappresentato formalmente nel sistema romano.

Secondo il metodo posizionale si fa riferimento ad un numero b , detto *base* del sistema di rappresentazione, ad esempio dieci, e ad un insieme di cifre che sono in numero pari al valore della base, ad esempio l'insieme $\{0, 1, \dots, 9\}$. In questo caso si può rappresentare un qualunque numero, in una base b fissata, mediante la sequenza finita di cifre (appartenenti all'insieme $\{0, 1, 2, \dots, b-1\}$):

$$c_{n-1}c_{n-2}\dots c_1c_0$$

che vanno interpretati come i coefficienti dell'espressione polinomiale

$$c_{n-1}b^{n-1} + \dots + c_1b^1 + c_0b^0$$

il cui valore, calcolato eseguendo le operazioni in base b , è il numero considerato.

Le due espressioni, quella sotto forma di sequenza di cifre e quella sotto forma polinomiale, costituiscono due forme equivalenti di rappresentazione dello stesso numero, e quindi si possono usare in modo intercambiabile, anche se la prima viene usata più frequentemente perché più compatta. Ad esempio, in base dieci, il numero naturale "trecentosettantadue", può essere espresso nella forma

$$3 \cdot 10^2 + 7 \cdot 10^1 + 2 \cdot 10^0$$

ed in quella equivalente mediante la sequenza di cifre 372.

Si noti che, evidentemente, un qualsiasi numero naturale b , assunto come base di un sistema di numerazione, in quello stesso sistema viene rappresentato dalla sequenza di cifre 10.

Per evidenziare la dipendenza dalla base scelta per la rappresentazione, si fa uso, in genere, anche della notazione

$$(c_{n-1}c_{n-2}\dots c_1c_0)_b$$

dove la base b viene rappresentata nella usuale rappresentazione del sistema decimale. Nell'esempio appena visto si può allora scrivere anche $(372)_{10}$.

Più in generale, una sequenza illimitata verso destra del tipo:

$$c_{n-1}\dots c_1c_0.c_{-1}c_{-2}\dots$$

rappresenta il numero reale assoluto:

$$c_{n-1}b^{n-1} + \dots + c_1b^1 + c_0b^0 + c_{-1}b^{-1} + c_{-2}b^{-2} + \dots$$

Ad esempio la sequenza di cifre 39,8 in base dieci rappresenta il numero

$$3 \cdot 10^1 + 9 \cdot 10^0 + 8 \cdot 10^{-1}$$

e la sequenza 53,14 in base sette rappresenta il numero

$$5 \cdot 7^1 + 3 \cdot 7^0 + 1 \cdot 7^{-1} + 4 \cdot 7^{-2}$$

Si noti che lo stesso numero è rappresentato diversamente in basi diverse. Ad esempio il numero dodici in base dieci viene espresso dalla sequenza 12, ed in base due, dalla sequenza 1100.

Conversione di base. Esaminiamo ora gli algoritmi di trasformazione da una rappresentazione ad un'altra. Partiamo dal caso particolarmente semplice in cui la base del sistema di rappresentazione di partenza sia la potenza k -esima della base del sistema di rappresentazione a cui vogliamo arrivare. In questo caso vi è una relazione molto semplice tra la rappresentazione di un numero in base b e la rappresentazione dello stesso numero in base b^k .

Dato un numero espresso in base b^k :

$$c_{n-1}(b^k)^{n-1} + \dots + c_1(b^k)^1 + c_0(b^k)^0$$

dove c_i è una cifra che assume valori compresi tra 0 e $b^k - 1$, e che in base b è rappresentata dalla sequenza

$$c_{i,k-1}, c_{i,k-2}, \dots, c_{i,0}$$

Sostituendo alla generica cifra c_i la sua rappresentazione in base b , si avrà:

$$\begin{aligned} & [c_{n-1,k-1} b^{k-1} + c_{n-1,k-2} b^{k-2} + \dots + c_{n-1,0} b^0] (b^k)^{n-1} + \\ & + [c_{n-2,k-1} b^{k-1} + c_{n-2,k-2} b^{k-2} + \dots + c_{n-2,0} b^0] (b^k)^{n-2} + \\ & + \dots + \\ & + [c_{0,k-1} b^{k-1} + c_{0,k-2} b^{k-2} + \dots + c_{0,0} b^0] (b^k)^0 \end{aligned}$$

e quindi

$$c_{n-1,k-1} c_{n-1,k-2} \dots c_{0,0}$$

è la rappresentazione in base b del numero dato.

Ad esempio un numero espresso in base 8 (cioè 2^3)

$$(653)_8 = 6 \cdot 8^2 + 5 \cdot 8^1 + 3 \cdot 8^0$$

può essere espresso in base 2 in questo modo:

$$\begin{aligned} & [1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0] (2^3)^2 + [1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0] (2^3)^1 + \\ & + [0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0] (2^3)^0 = \\ & = 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \end{aligned}$$

e quindi la sua rappresentazione binaria sarà:

$$(110\ 101\ 011)_2$$

Viceversa un numero espresso in base b può essere tradotto mediante una regola duale alla precedente in basi che siano potenze di b .

<i>Esadecimale</i>	<i>Decimale</i>	<i>Ottale</i>	<i>Binario</i>
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

Fig. 8 - Tabella di conversione di base

Ad esempio, il numerale $(1010011001)_2$ può essere tradotto in base 8 semplicemente raggruppando le cifre a tre a tre, a partire da destra, in questo modo:

$$(001\ 010\ 011\ 001)_2 = (1\ 2\ 3\ 1)_8$$

La rappresentazione in base 8 viene detta *ottale*. Analogamente si può ottenere, ad esempio, la rappresentazione in base 16, detta *esadecimale*.

Si vedano nella fig. 8 le rappresentazioni in base 16, 10, 8 e 2 dei primi sedici numeri interi.

Si noti che nel sistema esadecimale, dovendo utilizzare 16 cifre, si fa ricorso alle prime sei lettere dell'alfabeto per completare l'insieme delle cifre decimali.

Come già detto, per motivi tecnologici, negli elaboratori viene adottata la rappresentazione posizionale in base 2, utilizzando come cifre i simboli 0 e 1.

Nel trasferimento di dati da un operatore umano ad un calcolatore si pone quindi il problema di convertire i numeri dalla loro rappresentazione decimale a quella binaria e viceversa.

Esaminiamo ora, nel caso generale, gli algoritmi di conversione di rappresentazione da una generica base b_1 ad un'altra base b_2 .

Esistono algoritmi diversi per la conversione di base a seconda che si voglia effettuare le operazioni per la conversione nella base di arrivo (b_2) o in quella di partenza (b_1).

Nel primo caso l'espressione

$$c_{n-1}b_1^{n-1} + \dots + c_1b_1^1 + c_0b_1^0 + c_{-1}b_1^{-1} + c_{-2}b_1^{-2} + \dots$$

di un dato numerale, fornisce un metodo per convertire il numerale da base b_1 a base b_2 . Per ottenere la rappresentazione in base b_2 , basta infatti esprimere la base b_1 ed i coefficienti c_i nella base b_2 e poi eseguire, sempre in base b_2 , le operazioni indicate.

Esiste poi un altro algoritmo di conversione basato su operazioni eseguite nella base b_1 .

Dato un qualsiasi numero nella sua rappresentazione in base b_1 , lo si esprimerà in base b_2 con una espressione del tipo seguente

$$c_{n-1}b_2^{n-1} + \dots + c_1b_2^1 + c_0b_2^0 + c_{-1}b_2^{-1} + c_{-2}b_2^{-2} + \dots$$

dove le cifre c_i sono incognite.

Per determinare questi valori incogniti si procede considerando separatamente la parte intera da quella frazionaria del numero dato. Operando in base b_1 , si divide la parte intera per la base b_2 . Il resto di questa divisione coincide con la cifra c_0 , mentre per il quoziente vale l'espressione:

$$c_{n-1}b_2^{n-2} + \dots + c_2b_2^1 + c_1b_2^0$$

Ripetendo successivamente la divisione del quoziente ottenuto per la base b_2 , i resti via via calcolati costituiscono le cifre, dalla meno significativa alla più significativa, della rappresentazione in base b_2 della parte intera del numero dato.

Il procedimento termina quando si ottiene il primo quoziente nullo.

Per la parte frazionaria, continuando ad operare in base b_1 , si ha che moltiplicandola per la base b_2 , la parte intera del prodotto ottenuto coincide con la cifra c_{-1} , mentre la parte frazionaria dello stesso prodotto risulta essere:

$$c_{-2}b_2^{-1} + c_{-3}b_2^{-2} + \dots$$

Ripetendo successivamente la moltiplicazione della parte frazionaria del prodotto ottenuto per la base b_2 , le parti intere dei prodotti via via calcolati costituiscono le cifre, dalla più significativa alla meno significativa, della rappresentazione in base b_2 della parte frazionaria del numero dato. Il procedimento termina quando si è raggiunto il numero di cifre desiderate per la

rappresentazione in base b_2 .

Si noti che quest'ultimo algoritmo opera completamente nella base b_1 , mentre quello precedente opera completamente nella base b_2 .

La scelta tra i due metodi sarà allora fatta sulla base della maggiore semplicità dei calcoli da effettuarsi. In particolare se una delle due basi è la base dieci, evidentemente si preferirà eseguire le operazioni in quella base, indipendentemente dal fatto che sia la base di partenza o di arrivo del processo di trasformazione.

È facile allora verificare che, assumendo $b_1 = 2$ e $b_2 = 10$, si ha

$$(100011)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 32 + 2 + 1 = (35)_{10}$$

operando secondo il primo metodo; e che si ha anche

$$(35)_{10} = (100011)_2$$

facendo uso del secondo metodo, come specificato in fig. 9, avendo indicato nella due colonne, rispettivamente, i quozienti ed i resti, dei vari passi.

35	2
17	1
8	1
4	0
2	0
1	0
0	1

Fig. 9 - Esempio del metodo delle divisioni successive

3.2. L'aritmetica intera

I numeri interi vengono rappresentati in una o più celle di memoria dell'elaboratore, utilizzando il sistema di numerazione binario.

Se n è il numero di bit a disposizione per una cella, allora in essa è possibile rappresentare 2^n numeri.

Si noti che se nell'effettuare delle addizioni o moltiplicazioni si ottiene un risultato che richiede per la sua rappresentazione un numero di cifre maggiore di quello consentito, vengono rappresentate solo le cifre meno significative e

quelle più significative vengono perdute. Tale fenomeno viene detto *trabocco* (*overflow*).

Un numero intero relativo viene rappresentato, nell'aritmetica ordinaria, con valore assoluto e segno. Tale soluzione può essere adottata anche nell'aritmetica dei calcolatori rappresentando il segno con uno specifico bit. Questa notazione tuttavia dà luogo a considerevoli difficoltà nelle operazioni di addizione e sottrazione. Infatti sono necessari due sottosistemi di calcolo distinti: uno per addizionare ed uno per sottrarre. Considerando, ad esempio, una operazione di addizione, l'uso di uno dei due sottosistemi dipende dai segni degli operandi, che pertanto devono essere analizzati prima dell'esecuzione di qualsiasi operazione elementare.

Poiché, d'altra parte, anche la moltiplicazione e la divisione possono essere decomposte rispettivamente in addizioni e sottrazioni successive, vengono generalmente preferite altre forme di rappresentazione che rendono semplici le somme e le sottrazioni.

Presentiamo ora, in particolare, la forma di rappresentazione degli interi relativi come complemento alla base del sistema di numerazione prescelto. In questa rappresentazione si ha un duplice vantaggio: è richiesta una sola unità funzionale di calcolo (addizionatore) ed il trattamento dei dati è completamente ed univocamente determinato dall'operazione da eseguire.

Questa rappresentazione è basata su un'idea molto semplice, simile a quella usata per esprimere gli angoli con la notazione polare. Un angolo formato da un raggio e dall'asse di riferimento, ad esempio l'asse delle x del piano cartesiano, può essere misurato in senso antiorario in gradi da 0° a 360° .

A volte, se l'angolo supera i 180° ci riferiamo ad esso come angolo negativo. Così, ad esempio, l'angolo di $223^\circ = 180^\circ + 43^\circ$ può essere indicato anche come l'angolo di $-137^\circ = -180^\circ + 43^\circ$. Si osservi che 43° è il complemento di 137° a 180° . Si ha quindi che tutti gli angoli α tale che $0^\circ \leq \alpha < 180^\circ$ sono considerati come non negativi, mentre quelli dell'intervallo $180^\circ \leq \alpha < 360^\circ$ sono considerati come negativi.

Analogamente gli interi relativi possono essere rappresentati con n bit in modo da avere 2^n valori distinti a disposizione di interi negativi e non negativi.

Allo scopo di introdurre questa rappresentazione esaminiamo alcune trasformazioni algebriche dei numeri interi e alcune proprietà di queste trasformazioni che consentono di trattare in modo naturale i numeri relativi.

Dato un numero X e due numeri b ed n , è possibile calcolare il *residuo modulo b^n* di X , indicato con $|X|_{b^n}$, secondo l'espressione

$$X - \left\lfloor \frac{X}{b^n} \right\rfloor \cdot b^n$$

ove il simbolo $\lfloor \rfloor$ indica la funzione che calcola la parte intera inferiore.

Abbiamo ad esempio

$$\lfloor 25 \rfloor_{10^2} = 25,$$

e

$$\lfloor -25 \rfloor_{10^2} = -25 - \left\lfloor \frac{-25}{100} \right\rfloor_{100} = -25 + 100 = 75.$$

Per ogni intero relativo X esiste un solo residuo $|X|_{b^n}$. Se $|X| \leq b^n/2$ allora per ogni valore x esiste un solo X , tale che $x = |X|_{b^n}$. D'ora in poi faremo appunto l'ipotesi che $|X| \leq b^n/2$.

In questo caso, possiamo semplificare la definizione di residuo modulo b^n nel modo seguente:

$$|X|_{b^n} = \begin{cases} X & \text{se } 0 \leq X < b^n/2 \\ b^n - |X| & \text{se } -b^n/2 \leq X < 0 \end{cases}$$

La rappresentazione degli interi relativi X , nell'intervallo aperto a destra $(-b^n/2, b^n/2)$, tramite il residuo modulo b è detta *rappresentazione in complemento alla base*.

Si noti che se $X \geq 0$ la sua rappresentazione in complemento alla base è compresa nell'intervallo $[0, b^n/2)$ e coincide con la rappresentazione del suo valore assoluto nella base b . Se $X < 0$ la sua rappresentazione in complemento alla base è compresa nell'intervallo $[b^n/2, b^n)$.

Questa proprietà permette di distinguere le rappresentazioni dei numeri negativi da quelle dei positivi.

Nel caso particolare della base $b=2$ nei numeri positivi la cifra più significativa è pari a 0, ed in quelli negativi è pari a 1.

Ad esempio, con $b=2$ e $n=6$, la rappresentazione in complemento del numero $X = -1001$ è 110111, che si ottiene come $2^6 - |X|$, cioè $1000000 - 1001 = 110111$.

Il metodo di complementazione che abbiamo descritto, richiede una operazione di sottrazione. Si può tuttavia procedere in un modo più diretto, determinando le cifre della rappresentazione in complemento, operando sulle singole cifre della rappresentazione di partenza.

Se $c_{n-1}c_{n-2}\dots c_0$ è la sequenza di cifre della rappresentazione di un numero X, la rappresentazione in complemento alla base b del numero -X è ottenuta nel seguente modo:

1. si determina il numero rappresentato dalla sequenza di cifre

$$c'_{n-1}c'_{n-2}\dots c'_0, \text{ tale che } c'_i = (b-1) - c_i;$$

2. si somma 1 al numero ottenuto.

Esempio. Il complemento alla base 10 del numero di tre cifre -352 si calcola così:

- si determina il numero rappresentato dalla sequenza di cifre

$$c'_2 = (10-1) - 3 = 6, c'_1 = (10-1) - 5 = 4, c'_0 = (10-1) - 2 = 7$$

ottenendo 647;

- si somma 1, e si ottiene 648.

Esempio. Il complemento alla base 2 del numero di quattro cifre -0100 si calcola così:

- si determina il numero rappresentato dalla sequenza di cifre

$$c'_3 = (2-1) - 0 = 1, c'_2 = (2-1) - 1 = 0,$$

$$c'_1 = (2-1) - 0 = 1, c'_0 = (2-1) - 0 = 1,$$

ottenendo 1011;

- si somma 1, e si ottiene 1100.

Consideriamo infine le operazioni aritmetiche fra interi relativi.

Se si usa la forma complementata, come già accennato, in ogni caso si esegue una somma fra le rappresentazioni in complemento dei due numeri, ed il risultato è legato alla rappresentazione in complemento alla base dalle seguenti regole.

Facciamo riferimento alla base $b = 2$, e supponiamo di avere n bit per rappresentare un numero intero. Consideriamo due numeri positivi X e Y e indichiamo, per semplicità, con x e y le loro rispettive rappresentazioni in complemento.

Consideriamo i quattro casi di somme algebriche possibili tra X e Y.

Primo caso: + X + Y

In questo caso il risultato della somma effettuata sulle rappresentazioni in complemento è

$$x + y = X + Y$$

che risulta essere corretto se non si è avuto trabocco, cioè solo se il valore ottenuto è minore di $b^n/2$.

Ad esempio, per $n=5$, si ha:

$$\begin{array}{r} 01001 + \\ 00001 = \\ \hline 01010 \end{array} \quad \begin{array}{r} 9 + \\ 1 = \\ \hline 10 \end{array}$$

risultato corretto per assenza di trabocco;

$$\begin{array}{r} 01001 + \\ 01000 = \\ \hline 10001 \end{array} \quad \begin{array}{r} 9 + \\ 8 = \\ \hline 17 \end{array}$$

risultato errato per presenza di trabocco.

Secondo caso: + X - Y

In questo caso eseguiamo la somma tra le notazioni in complemento di X e di -Y :

$$x + y = X + 2^n - Y = 2^n + X - Y$$

Si hanno due sottocasi:

1. $X - Y > 0$ cioè $X > Y$

in questo caso per ottenere $X - Y$ da $x + y$ basta sottrarre 2^n cioè trascurare il bit più significativo.

Ad esempio, con $n=5$, dovendo calcolare $01001 - 00100$ effettuiamo la somma tra le rappresentazioni in complemento ottenendo

$$\begin{array}{r} 01001 + \\ 11100 = \\ \hline 100101 \end{array} \quad \begin{array}{r} 9 + \\ - 4 = \\ \hline 5 \end{array}$$

ed il risultato corretto sarà 00101.

2. $X - Y < 0$ cioè $X < Y$

in questo caso il risultato che si ottiene calcolando $x + y$ coincide con la rappresentazione in complemento di $X - Y$.

Ad esempio, con $n=5$, dovendo calcolare $01001 - 01101$ effettuiamo la somma delle rappresentazioni in complemento ottenendo

$$\begin{array}{r} 01001 + \\ 10011 = \\ \hline 11100 \end{array} \quad \begin{array}{r} 9 + \\ -13 = \\ \hline -4 \end{array}$$

che è il complemento di 00100 ,

Terzo caso: $-X + Y$

Valgono considerazioni analoghe a quelle fatte per il secondo caso, scambiando i due addendi tra loro.

Quarto caso: $-X - Y$

La somma tra le notazioni in complemento dà

$$x + y = 2^n - X + 2^n - Y = 2^n + 2^n - (X + Y)$$

che differisce di 2^n dal valore corretto della somma richiesta, (espressa nella notazione in complemento), solo se non si è avuto trabocco, cioè solo se il risultato di $-X - Y$ è maggiore di $b^n/2$, e sia quindi esprimibile in notazione complementata per il fissato valore di n .

Ad esempio, con $n=5$, dovendo calcolare $-00001 - 00010$ effettuiamo la somma delle rappresentazioni in complemento ottenendo

$$\begin{array}{r} 11111 + \\ 11110 = \\ \hline 111101 \end{array} \quad \begin{array}{r} -1 + \\ -2 = \\ \hline -3 \end{array}$$

da cui si ottiene la somma effettiva 11101 (rappresentazione in complemento di -3 , risultato esatto di $-X - Y$) trascurando il bit più significativo. Si noti in questo caso il valore 1 del bit più significativo del risultato ottenuto che evidenzia il mancato trabocco.

Dovendo calcolare invece $-01110 - 01111$ effettuiamo la somma delle rappresentazioni in complemento ottenendo

$$\begin{array}{r} 10010 + \\ 10001 = \\ \hline 1000011 \end{array} \quad \begin{array}{r} -14 + \\ -15 = \\ \hline -29 \end{array}$$

che, una volta eliminato il bit più significativo, fornisce il risultato 00011 che è errato a causa del trabocco.

Riassumendo si può notare che se invece della somma algebrica si esegue la somma tra le rappresentazioni complementate, a parte i casi di trabocco, la somma effettiva coincide con il risultato o è ricavata da questo trascurando l'ultimo bit.

Passiamo ora alla moltiplicazione nelle rappresentazioni in complemento. Nel caso della moltiplicazione, si deve assumere che se n è il numero di bit per la rappresentazione degli operandi, si ha bisogno di $2n$ bit per la rappresentazione del prodotto.

Notiamo inoltre che se si esegue direttamente il prodotto tra due numeri X ed Y , rappresentati in complemento alla base, il risultato dell'operazione è diverso dal risultato vero se uno o tutti e due gli operandi sono numeri negativi.

Ad esempio, con $b=2$ e $n=3$, dalla moltiplicazione dei due numerali 010 (che rappresenta 2) e 110 (che rappresenta -4) si ottiene il valore errato 001100, che rappresenta 12.

Per trattare in modo corretto il segno degli operandi di una moltiplicazione si può procedere in due modi. Il primo, più banale, consiste nel determinare i valori assoluti degli operandi e memorizzare i rispettivi segni, effettuare poi la moltiplicazione dei due valori assoluti, determinare il segno del prodotto, e infine effettuare il complemento del risultato della moltiplicazione.

Un secondo metodo si basa sulle seguenti considerazioni. Quando si effettua una moltiplicazione, generalmente nei prodotti intermedi si sottintendono degli zeri non significativi a sinistra della cifra più significativa, si fa cioè implicitamente una estensione della rappresentazione, passando da n a $2n$ bit.

Quando invece si opera con rappresentazioni complementate l'estensione della rappresentazione deve essere fatta con degli zero per i numeri positivi e degli uno per i numeri negativi.

Ad esempio, con $b=2$ e $n=6$, i numerali 010, 101 divengono 000010 e 111101.

Assumiamo ora $b=2$, e siano

$$X = x_n x_{n-1} \dots x_1 x_0$$

e

$$Y = y_n y_{n-1} \dots y_1 y_0$$

rispettivamente il moltiplicando e il moltiplicatore; il modo di operare è diverso a seconda che $X > 0$ e $Y < 0$, $X < 0$ e $Y > 0$ e infine $X < 0$ e $Y < 0$.

Primo caso: $X > 0$ e $Y < 0$

$X \cdot Y$ può calcolarsi come

$$X \cdot (100\ldots 0) + X \cdot (0 y_{n-1}\ldots 0) + \ldots + X \cdot (000\ldots y_0)$$

I prodotti intermedi sono tutti positivi eccetto il primo ($X \cdot (100\ldots 0)$) che va dunque complementato. A questo punto è possibile eseguire la somma che dà luogo alla rappresentazione complementata del risultato.

Ad esempio, con $n=3$, effettuiamo il prodotto $3 \cdot (-2)$

$$\begin{array}{r} 011 \\ 110 \\ \hline 000000 \\ 00011- \\ 1101- \\ \hline 111010 \end{array}$$

che è la rappresentazione di -6.

Secondo caso: $X < 0$ e $Y > 0$

In questo caso tutti i prodotti intermedi vanno lasciati inalterati e devono essere estesi aggiungendo a sinistra una sequenza di cifre uguali ad 1; a questo punto si esegue la somma.

Ad esempio, ancora con $n=3$, effettuiamo il prodotto $(-3) \cdot 2$

$$\begin{array}{r} 101 \\ 010 \\ \hline 000000 \\ 11101- \\ 0000- \\ \hline 111010 \end{array}$$

che è la rappresentazione complementata di -6.

Terzo caso: $X < 0$ e $Y < 0$

È una semplice estensione delle regole viste nei primi due casi.

Ad esempio, con $n=3$, effettuiamo il prodotto $(-3) \cdot (-2)$

$$\begin{array}{r}
 101 \\
 110 \\
 \hline
 000000 \\
 11101- \\
 0011- \\
 \hline
 1000110
 \end{array}$$

che è la rappresentazione in complemento di 6 (l'uno più significativo va trascurato in base alle regole mostrate per la somma di numerali in rappresentazione complementata).

3.3. L'aritmetica in virgola mobile

Ogni numero reale X , rappresentato in base b , può essere scritto nella forma

$$X = m \cdot b^e$$

detta *forma normalizzata in base b* di X , dove e , detta *caratteristica in base b* di X , è un intero relativo ed m , detta *mantissa in base b* di X , è un numero che soddisfa la diseguaglianza

$$1/b \leq |m| < 1$$

e rappresenta il valore

$$c_1 \cdot b^{-1} + c_2 \cdot b^{-2} + c_3 \cdot b^{-3} + \dots$$

qualora le sue cifre siano c_1, c_2, c_3, \dots

Se questa espansione è finita o periodica, X è un numero razionale, altrimenti è irrazionale.

Poiché l'insieme R è infinito non tutti i suoi elementi possono essere rappresentati nella memoria di un elaboratore.

Consideriamo una cella di memoria di lunghezza fissata, ad esempio di 32 bit. Operiamo in base $b=2$ e supponiamo di riservare 24 bit per rappresentare m e 7 bit per rappresentare e con la notazione in complemento. Il bit restante sarà dedicato a rappresentare il segno di m , secondo la convenzione che fa corrispondere zero al segno positivo e uno al segno negativo.

Sottolineiamo che la forma normalizzata fa riferimento ad una base b fissata

(nel nostro caso $b=2$). Ciò significa che se ci riferiamo a rappresentazioni di numeri in una base diversa, allora si deve prima procedere alla sua conversione nella corrispondente rappresentazione in base b , e poi si passa alla forma normalizzata in base b . Evidentemente l'ordine delle due operazioni non può essere invertito.

Consideriamo ad esempio il numero $X = 5$. Poiché $(5)_{10} = (101)_2$ si ha $m = |m| = (0.10100\dots0)_2$ ed anche $e = (11)_2$.

Il numero reale 5 sarà rappresentato in una cella di memoria di 32 bit nel modo seguente:

0	0000011	10100 ... 000
1	2	8 9 32

Si noti che è anche possibile operare inversamente. Se è nota una sequenza di 32 bit, da interpretare come un numero reale, ed è fissata quindi anche l'interpretazione dei singoli gruppi di bit, si può ottenere il valore reale rappresentato.

Ad esempio la sequenza

0	1111100	100 ... 000
1	2	8 9 32

è interpretabile nel modo seguente. Poiché il segno della mantissa m è positivo si ha:

$$m = |m| = (0.100\dots000)_2 = 1/2$$

ed essendo $e = -4$, si ha:

$$X = m \cdot 2^e = 1/2 \cdot 2^{-4} = 1/32 = 0.03125$$

Evidentemente in un elaboratore con celle da n bit, di cui k bit rappresentano la mantissa ed h bit rappresentano la caratteristica, con

$$n = k + h + 1$$

è possibile rappresentare solo quei numeri per cui

$$1 \cdot 2^{-1} \leq |m| \leq 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + \dots + 1 \cdot 2^{-k}$$

e

$$|e| \leq 1 \cdot 2^0 + 1 \cdot 2^1 + \dots + 1 \cdot 2^{h-2}.$$

Ciò esclude la possibilità di trattare qualsiasi numero razionale restringendo la rappresentabilità ad un limitato sottoinsieme dei numeri razionali. In altre parole

L'insieme dei numeri rappresentabili in un elaboratore è un sottoinsieme finito dei numeri razionali. Quest'insieme, indicato con F , è noto come l'insieme dei numeri a *virgola mobile* (*floating point*), poiché la virgola (il punto, nella convenzione anglosassone) di un numero frazionario può essere resa mobile semplicemente modificando la sua caratteristica.

Poiché l'insieme F è finito, esso contiene un elemento massimo ed un elemento minimo. È anche facile verificare che l'insieme F è simmetrico rispetto allo zero e per caratterizzarlo è quindi sufficiente esaminare gli elementi positivi.

Il valore massimo X_{\max} di F si ottiene per il massimo valore di m e di e , ed è rappresentato dai 32 bit:

0	0111111	111.....111		
1	2	8	9	32

con

$$m = |m| = 1 - 2^{-24}$$

e

$$e = 2^6 - 1 = 63$$

Per cui si ha:

$$X_{\max} = (1 - 2^{-24}) \cdot 2^{63} = 2^{63} - 2^{39}$$

Il valore positivo minimo X_{\min} di F si ottiene per

$$m = |m| = 2^{-1}$$

e

$$e = -64$$

ed è rappresentato dai 32 bit:

0	1000000	100...000		
1	2	8	9	32

Per cui si ha:

$$X_{\min} = 2^{-1} \cdot 2^{-64} = 2^{-65}$$

Si noti che nell'intorno di X_{\min} due valori rappresentabili sono estremamente vicini. Essi sono distanti circa 2^{-65} . Se si aumenta la caratteristica di una unità, l'intervallo tra due elementi adiacenti di F si raddoppia.

Pertanto gli elementi di F non sono distribuiti uniformemente sull'asse reale. Nell'intorno di X_{\max} due valori adiacenti di F distano circa 2^{39} .

Riassumendo si possono elencare le seguenti proprietà dell'insieme F:

1. F è un sottoinsieme finito dei numeri razionali;
2. F è simmetrico rispetto allo zero, e si hanno due diverse rappresentazioni di zero;
3. gli elementi di F non sono uniformemente distribuiti sull'asse reale;
4. molti numeri razionali non appartengono ad F. Ad esempio numeri quali $\frac{1}{3}$, $\frac{5}{6}$ e $\frac{1}{10}$ non sono elementi di F, poiché la proprietà di appartenenza ad F è limitata a solo alcuni dei numeri razionali con denominatore dato da una potenza di 2;
5. F non gode delle proprietà di commutatività, associatività, e distributività dell'insieme dei numeri reali, secondo cui, per una qualsiasi terna di numeri a, b, c si ha che:

$$a+b = b+a \quad a \cdot b = b \cdot a \quad (\text{commutatività});$$

$$(a+b)+c = a+(b+c) \quad (a \cdot b) \cdot c = a \cdot (b \cdot c) \quad (\text{associatività});$$

$$a \cdot (b+c) = a \cdot b + a \cdot c \quad (a+b) \cdot c = a \cdot c + b \cdot c \quad (\text{distributività}).$$

Si noti anche che a volte, componendo con addizioni e moltiplicazioni numeri di F si ottengono valori che non appartengono ad F.

Ad esempio:

$$X_1 = 1/2 \cdot X_{\min}$$

$$X_2 = 2 \cdot X_{\max}$$

$$X_3 = X_{\max}/2^{-64}$$

$$X_4 = 2^{14} + 2^{-14}$$

sono quattro numeri razionali che non appartengono ad F, pur essendo stati calcolati mediante elementi di F.

In pratica per rappresentare un numero reale X tale che

$$0 \leq |X| < 2^{63}$$

si sceglie l'elemento x di F che sia più vicino ad X. La funzione che ad ogni X di R associa l'elemento x di F che lo rappresenta è detta *funzione di arrotondamento* ed è indicata con fl; per cui si ha:

$$x = fl(X)$$

Conseguentemente l'aritmetica a virgola mobile è costruita in modo da garantire la chiusura di F, cioè in modo da garantire che ogni risultato di una

operazione sia rappresentato in F, facendo eventualmente uso della funzione di arrotondamento.

Consideriamo, ad esempio, il caso della somma. Abbiamo allora

$$X = a + b$$

$$x = fl(X) = fl(fl(a) + fl(b))$$

e l'espressione

$$\epsilon = x - X = fl(a+b) - (a+b)$$

indica l'errore compiuto nel calcolo della somma. Anche per le altre operazioni aritmetiche esistono formule analoghe per la determinazione dell'errore.

Poiché in F non valgono le proprietà associativa e distributiva si possono avere errori di calcolo. Ad esempio, dati i numeri reali a, b, c e d arbitrari, calcolando in F i valori delle quattro espressioni seguenti:

$$[a \cdot (b+c)/d], [a/d \cdot (b+c)], [(a \cdot b+a \cdot c)/d], [(b+c)/d \cdot a]$$

si ottengono generalmente elementi diversi di F, benché le quattro espressioni siano algebricamente equivalenti.

Per ottenere quattro valori uguali la funzione fl dovrebbe godere di alcune proprietà, quali ad esempio:

$$fl[fl(a) \cdot fl(b+c)] = fl[fl(a \cdot b) + fl(a \cdot c)]$$

per una arbitraria terna di elementi a, b e c. Ciò non è vero per nessuno degli elaboratori attuali.

Quando si opera in virgola mobile oltre alle operazioni sulla mantissa, che risultano uguali a quelle descritte per gli interi relativi, occorre anche operare sui fattori di scala ed, eventualmente, procedere alla normalizzazione del risultato.

Somma e sottrazione. Per eseguire queste operazioni occorre riportare i due operandi, rappresentati da m_x, e_x e da m_y, e_y , alla stessa caratteristica e poi eseguire l'operazione sulle mantisse.

In generale la nuova caratteristica viene scelta uguale al massimo fra e_x ed e_y . Il risultato non sarà in generale in forma normalizzata, quindi si deve procedere alla sua normalizzazione.

Esempio. Assumendo $b=10$, per sommare i due numeri $0,235 \cdot 10^3$ e $0,322 \cdot 10^2$ occorre procedere così:

- si riportano i due numeri alla stessa caratteristica, che viene scelta uguale a 10^3 . Si ottiene: $0,235 \cdot 10^3$ e $0,0322 \cdot 10^3$;
- si sommano le due mantisse, ottenendo il valore $0,2672 \cdot 10^3$, che risulta normalizzato.

Moltiplicazione. La moltiplicazione è molto semplice perché le caratteristiche degli operandi vanno sommate, mentre per le mantisse valgono le regole date nel paragrafo precedente. Anche in questo caso in generale si deve procedere ad una normalizzazione del risultato.

Esempio. Assumendo con $b=10$, il prodotto dei due numeri $0,27 \cdot 10^2$ e $0,35 \cdot 10^4$ si ottiene sommando le caratteristiche, ed eseguendo il prodotto delle mantisse. Si avrà così come risultato il valore $0,0945 \cdot 10^2$, che normalizzato fornisce il valore $0,945 \cdot 10^1$.

Divisione. Nella divisione il quoziente ha caratteristica pari alla differenza delle caratteristiche degli operandi. Anche in questo caso generalmente si deve procedere ad una normalizzazione del risultato.

Esempio. =Assumendo $b=10$, si voglia dividere il numero $0,572 \cdot 10^{-1}$ per il numero $0,27 \cdot 10^2$. Come caratteristica del quoziente si avrà -3.

Il quoziente delle mantisse è pari a 2,1185 e quindi il risultato in forma normalizzata è $0,21185 \cdot 10^{-2}$.

Concludiamo ricordando l'esistenza di un altro metodo di rappresentazione dei reali. Esso consiste nella rappresentazione in virgola fissa, in cui si stabilisce a priori la posizione della cella di memoria in cui cade la virgola, destinando un certo numero di bit alla parte intera ed il resto dei bit alla parte frazionaria.

In questo modo le operazioni di somma algebrica risultano molto semplici, dal momento che i numeri, così memorizzati, risultano già incolonnati con la virgola nella medesima posizione.

Esempio. Consideriamo un calcolatore con parole lunghe 32 bit. In questo caso una soluzione possibile è rappresentata dal supporre la virgola a destra del 16-esimo bit. Nei primi sedici bit viene rappresentata la parte intera con il segno, nei secondi sedici bit la parte decimale.

La somma dei due numeri $010,011_2 = 2,375_{10}$ e $101,101_2 = 5,625_{10}$ è data allora da:

$$\begin{array}{r} 010,011 + \\ 101,101 = \\ \hline 1\ 000,000 \end{array} \qquad \begin{array}{r} 2,375 + \\ 5,625 = \\ \hline 8,000 \end{array}$$

<i>Valore ottale</i>	<i>Carattere</i>	<i>Valore ottale</i>	<i>Carattere</i>
40	< spazio bianco >	106	F
41	I	107	G
42	"	110	H
43	#	111	I
44	\$	112	J
45	%	113	K
46	&	114	L
47	' (apice)	115	M
50	(116	N
51)	117	O
52	*	120	P
53	+	121	Q
54	, (virgola)	122	R
55	-	123	S
56	. (punto)	124	T
57	/	125	U
60	0	126	V
61	1	127	W
62	2	130	X
63	3	131	Y
64	4	132	Z
65	5	133	[
66	6	134	\
67	7	135]
70	8	136	^
71	9	137	-
72	:	140	,
73	;	141	a (¹)
74	<	fino a	fino a
75	=	172	z
76	>	173	{
77	?	174	:
100	@	175	}
101	A	176	~
102	B	177	DEL
103	C		
104	D		
105	E		

Fig. 10 - Codice ASCII

(¹) Alfabeto minuscolo

Lo svantaggio principale di questo metodo di memorizzazione è dato dalla limitata estensione dei valori rappresentabili. Nel caso dell'esempio precedente non possono essere rappresentati tutti i numeri maggiori ed uguali di $2^{15} = 32768$.

3.4. Rappresentazione dei caratteri

Ogni elaboratore riconosce un ben determinato alfabeto di simboli: lettere, cifre numeriche, simboli speciali (segni di interpunkzione, parentesi, simboli operazionali, ecc.). Generalmente l'alfabeto di una macchina differisce da quello di un'altra solo per i simboli speciali.

Ciascuno dei caratteri di un alfabeto viene rappresentato nella memoria dell'elaboratore con un byte e quindi con 6 o con 8 bit. Ad esso corrisponde, secondo un ben definito codice una ed una sola delle possibili configurazioni binarie ottenibili appunto con 6 o 8 bit.

Più caratteri possono essere rappresentati in una cella della memoria dell'elaboratore. Ad esempio, con celle di 36 bit si possono rappresentare 6 caratteri con codice a 6 bit per carattere (codice BCD, oppure codice FIELDATA). Con celle di 32 bit si ha, invece, la possibilità di memorizzare 4 caratteri, ciascuno con 8 bit, rappresentati ad esempio, secondo il codice EBCDIC oppure il codice ASCII (v. fig. 10). Nei vari casi in relazione alla codifica usata si parla semplicemente di caratteri BCD, FIELDATA, EBCDIC oppure ASCII.

Si noti infine che è sempre possibile rappresentare un numero, intero o reale, come una sequenza di cifre, ciascuna rappresentata come un particolare carattere di un fissato alfabeto. Sui numeri così rappresentati risulta molto complesso eseguire operazioni aritmetiche. Questa forma di rappresentazione può essere usata convenientemente per la sola memorizzazione e stampa di valori.

9.

Sistema di elaborazione: software di base

Un sistema di calcolo può essere visto come una struttura gerarchica composta di elementi hardware e software. Volendo schematizzare questa struttura si può far riferimento alla fig. 1 in cui si nota un primo strato hardware che rappresenta la struttura fisica dell'elaboratore: memoria, unità centrale, e unità periferiche.

Lo strato successivo, uno strato software, detto software di base, è costituito da quei programmi che svolgono funzioni di servizio per le comunicazioni tra le varie componenti del sistema hardware, e per la distribuzione e la gestione delle varie risorse dell'elaboratore (memoria, unità centrale, unità periferiche) in modo ottimale rispetto alle richieste dell'utente singolo o di più utenti. Inoltre essi provvedono all'utilizzazione dell'elaboratore mediante programmi in linguaggi ad alto livello, realizzando la traduzione di questi programmi nella corrispondente forma in linguaggio macchina.

L'ulteriore livello, quello del software applicativo, è quello più vicino ai problemi, ed è costituito dai programmi per il trattamento e la soluzione di fissate classi di applicazioni.

In questo capitolo esaminiamo il livello centrale, quello del software di base, partendo dal processo di traduzione dei programmi fino alla loro esecuzione, discutendo in particolare l'uso dell'elaboratore da parte di più utenti.

1. TRADUZIONE ED ESECUZIONE DI PROGRAMMI

In questo paragrafo introduciamo le problematiche relative alle trasformazioni che i programmi scritti in linguaggi ad alto livello devono subire per poter essere effettivamente eseguiti su un elaboratore. La prima trasformazione è

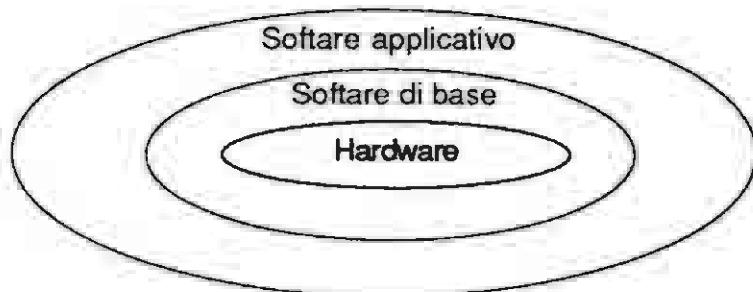


Fig. 1 - Struttura hardware-software del sistema di calcolo

quella di traduzione del programma in linguaggio ad alto livello nel corrispondente programma in linguaggio macchina.

Una seconda trasformazione del programma consiste nel collegarlo con programmi di supporto, come ad esempio programmi per l'uso delle unità periferiche oppure programmi per il calcolo di funzioni matematiche elementari, attraverso un processo noto come correlazione.

Infine per l'esecuzione effettiva è necessaria una fase di caricamento del programma in memoria, definendo l'indirizzo dell'istruzione iniziale del programma e, conseguentemente, di tutte le altre istruzioni.

I successivi paragrafi affrontano in maggiore dettaglio queste fasi di trasformazione, percorrendo l'intero passaggio dal programma in linguaggio ad alto livello alla sua completa esecuzione.

1.1. Compilatori ed interpreti

Nel cap. 1 sono state presentate le componenti principali di un linguaggio: sintassi e semantica; affrontiamo ora il problema della implementazione di un linguaggio, cioè la progettazione e la costruzione di quei programmi, detti compilatori ed interpreti, che consentono di eseguire, su un dato elaboratore programmi scritti in quel linguaggio.

Un *compilatore* è un programma che ha la funzione di tradurre programmi scritti in un linguaggio ad alto livello in corrispondenti programmi scritti in linguaggio macchina.

Questa funzione di traduzione viene realizzata mediante procedure distinte, con le quali il programma da tradurre, detto *programma sorgente*, viene analizzato al fine di controllare la sua correttezza lessicale, sintattica e semantica, e viene trasformato successivamente in forme sempre più vicine alla

versione finale, detta programma oggetto. È evidente lo stretto legame tra il processo di compilazione, e quindi il compilatore, e le caratteristiche del linguaggio ad alto livello in cui il programma sorgente è scritto. La rigorosa e non ambigua definizione della sintassi e della semantica di questo linguaggio può garantire una compilazione corretta dei programmi.

La prima funzione svolta dal compilatore è un'analisi lessicale, realizzata dal modulo che prende il nome di *analizzatore lessicale*: il programma sorgente viene esaminato, carattere per carattere come una stringa, al fine di individuare i simboli che lo compongono, classificando le parole chiave del linguaggio, gli operatori, le costanti, e gli identificatori. Durante questa fase inizia la costruzione di una tavola, detta *tavola dei simboli*, che conterrà le informazioni relative ai vari simboli, e si ha una prima trasformazione del programma sorgente dalla sua forma originale di sequenza di simboli ad una forma di sequenza di parole.

A valle dell'analisi lessicale si esegue l'analisi sintattica che ha lo scopo di individuare la struttura sintattica della stringa in esame. Ad esempio nella sequenza

ALFA1 := 5 + A * B

la sottostringa 5+A+B viene riconosciuta come <espressione> e la stringa nel suo complesso come <assegnazione>, in accordo con la regola sintattica dell'istruzione di assegnazione

<assegnazione> ::= <variabile> := <espressione>

L'analisi sintattica non prevede l'esame di tutti i caratteri dell'istruzione di assegnazione, già eseguito nella fase di analisi lessicale, e di fatto esamina una frase del tipo

id1 := id2 + id3 * id4

dove id1, id2, id3, id4, sono i nomi simbolici, cioè le chiavi della tavola dei simboli, generata nella fase precedente, nella quale sono stati memorizzati la costante 5 e gli identificatori ALFA1, A, e B.

L'analizzatore sintattico ha quindi accesso alla tavola dei simboli per verificare che l'uso che viene fatto dei vari identificatori sia coerente in ogni momento con le loro proprietà.

È importante notare che l'analisi sintattica non ha soltanto il compito di riconoscere la correttezza delle istruzioni, ma anche quello di costruire l'albero di derivazione delle stesse, poiché sarà proprio con l'aiuto di questo albero che si potrà generare il corrispondente codice oggetto.

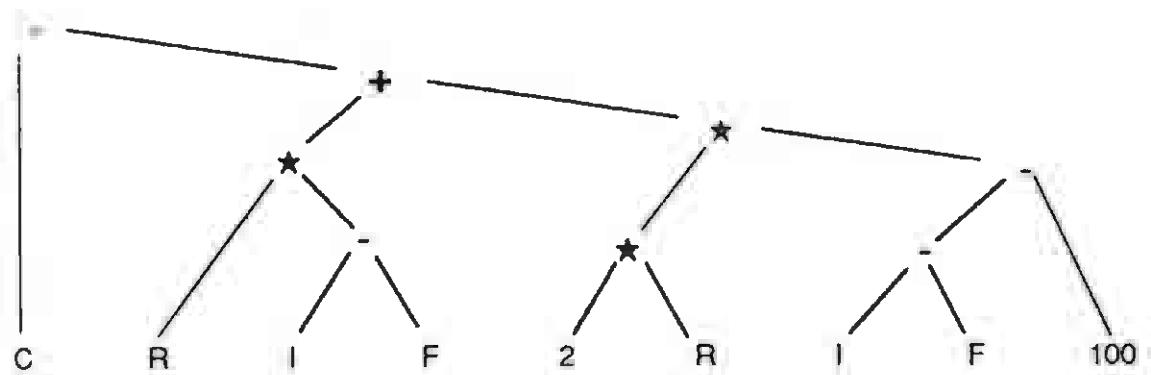


Fig. 2 - Esempio di albero sintattico

Dopo il riconoscimento del tipo di frase, il programma viene ulteriormente trasformato in una forma intermedia che facilita la realizzazione della codifica nella sua versione di programma oggetto.

Considerando ancora come esempio una istruzione di assegnazione, si può costruire una sequenza di terne, oppure una matrice, in cui per ogni operatore presente nell'espressione si riportano i due operandi relativi.

Così per l'espressione

$$C := R * (I - F) + 2 * R * (I - F - 100)$$

il cui albero sintattico è presentato in fig. 2, si ottiene la tabella di elementi, cioè un insieme di terne di oggetti (vedi fig. 3), ciascuno identificato con un nome simbolico che può essere utilizzato per riferirsi all'elemento stessoogniqualvolta sia necessario.

Si noti che nella tabella, al posto degli identificatori e delle costanti ci sono i nomi simbolici ad essi attribuiti durante la fase di analisi lessicale.

Il passo successivo sarà la generazione del codice oggetto a partire dalla forma di rappresentazione a tabella. Il codice così generato può essere soggetto ad ottimizzazione, per esempio eliminando le ripetizioni e migliorando l'utilizzazione dei registri di macchina. Si noti che alcune ottimizzazioni si possono già ottenere nelle fasi precedenti. Ad esempio si possono rimuovere dall'interno di un ciclo alcune istruzioni, come nel caso presentato in fig. 4, dove il risultato non cambia se l'istruzione $A := B + C + D$ viene portata fuori del ciclo **for**. Si possono eseguire anche operazioni numeriche durante la fase di analisi stessa per cui, ad esempio, l'espressione $A := (6+2)/3+B$ dà luogo alla tabella semplificata

M1	+	4	B
M2	\coloneqq	A	M1

	<i>codice</i>	<i>oper.1</i>	<i>oper.2</i>
M1	-	I	F
M2	*	R	M1
M3	*	2	R
M4	-	I	F
M5	-	M4	100
M6	*	M3	M5
M7	+	M2	M6
M8	\coloneqq	C	M7

Fig. 3 - Esempio di rappresentazione a tabella

```

for i  $\coloneqq$  1 to 100 do
begin
  j: = j + 1;
  a: = b + c + d;
  t: = i * a + j
end;

```

Fig. 4 - Esempio di passo di ottimizzazione

Inoltre si possono semplificare le tabelle eliminando elementi uguali, come nel caso dell'esempio precedente in cui gli elementi M1 ed M4 sono coincidenti.

Durante questi ultimi passi di generazione del codice e di ottimizzazione viene effettuata l'*analisi semantica* procedendo al controllo delle regole di semantica statica.

In ciascuna delle tre fasi di analisi descritte vengono generati eventuali messaggi di errore ed informazioni utili alla localizzazione degli errori e alla loro eliminazione.

La presentazione del processo di compilazione che abbiamo fin qui fatto, è stata riferita al solo caso delle istruzioni di assegnazione. Anche se per altri tipi di istruzioni la forma del codice generato nei vari passi può differire, con riferimento a strutture dati diverse, tuttavia lo schema logico generale del processo di compilazione può essere riassunto come in fig. 5.

<i>Funzione</i>	<i>Input</i>	<i>Output</i>
Analisi lessicale	Programma sorgente come sequenza di simboli	Codice intermedio, tavola dei simboli, errori lessicali
Analisi sintattica	Codice intermedio tavola simboli	Albero sintattico, codice intermedio, tavola dei simboli, errori sintattici
Analisi semantica	Codice intermedio tavola di simboli	Programma oggetto, tavole varie, errori semantici

Fig. 5 - Schema logico della compilazione

Si noti che l'efficienza di un compilatore è strettamente dipendente dall'efficienza degli algoritmi di gestione delle strutture di dati, come le tavole e gli alberi, che intervengono nel processo di traduzione. A questo proposito va anche ricordato che le varie fasi della compilazione possono essere svolte attraverso più passate, cioè attraverso esami ripetuti dello stesso programma sorgente. Ciò riduce l'efficienza del processo di compilazione, ma può aumentare la capacità di individuare le anomalie eventualmente presenti.

Infine bisogna tener presente che, dal punto di vista dell'utente, la capacità diagnostica del compilatore costituisce una delle caratteristiche più significative della sua qualità.

Un altro processo di esecuzione di un programma in un linguaggio ad alto livello, prevede la contemporanea traduzione ed esecuzione del programma sorgente, istruzione per istruzione. Questo processo è detto di interpretazione, ed è realizzato mediante un programma detto *interprete*. Ciascuna frase del

programma sorgente viene separatamente fornita in ingresso all'interprete, analizzata sintatticamente e, se corretta, tradotta in una o più istruzioni del linguaggio oggetto. Queste frasi vengono eseguite immediatamente, man mano che sono generate, prima di passare all'analisi-traduzione-esecuzione dell'istruzione successiva del programma sorgente.

Questo metodo di esecuzione dei programmi in linguaggi ad alto livello viene anche detto simulazione software, nel senso che l'interprete si comporta di fatto come un simulatore, poiché arricchisce la potenza dell'elaboratore capace di eseguire soltanto le istruzioni in linguaggio macchina, fornendogli la possibilità di eseguire direttamente le istruzioni di un programma scritto in un linguaggio simbolico ad alto livello. È da notare che, anche in questo caso l'affidabilità della traduzione-esecuzione si basa sul fatto che l'interprete utilizzato garantisca la correttezza semantica dell'esecuzione.

1.2. Correlazione e caricamento

Attraverso il processo di compilazione, descritto nel precedente paragrafo, si ottiene un codice oggetto, in linguaggio macchina, che tuttavia non è ancora eseguibile. In questo paragrafo descriviamo le ulteriori fasi del processo di trasformazione dei programmi compilati al fine di renderli eseguibili.

Il programma oggetto prodotto da un compilatore contiene in generale una serie di riferimenti esterni, ad esempio a programmi di servizio per l'uso delle unità periferiche, o a programmi per il calcolo di funzioni matematiche predefinite.

Questi riferimenti esterni, rappresentati mediante nomi simbolici, costituiscono alcuni dei cosiddetti legami o simboli irrisolti, per i quali cioè non è ancora noto il significato all'interno del programma compilato, e ai quali non corrisponde ancora un indirizzo.

Supponiamo, ad esempio, che il programma *padre* utilizzi n programmi *figlio₁*, *figlio₂*...*figlio_n*, e che questi, a loro volta, richiamino altri m programmi *nipote₁*, *nipote₂*....*nipote_m*, secondo una struttura di dipendenza del tipo indicato in fig. 6.

Perché sia possibile arrivare all'esecuzione del programma *padre* è necessario collegarli in un unico modulo oggetto in linguaggio macchina. Questa funzione di collegamento, o correlazione, viene svolta da un programma specifico che prende il nome di *collegatore* o *correlatore* (*linker*).

Per meglio capire il modo di operare del correlatore si consideri la struttura

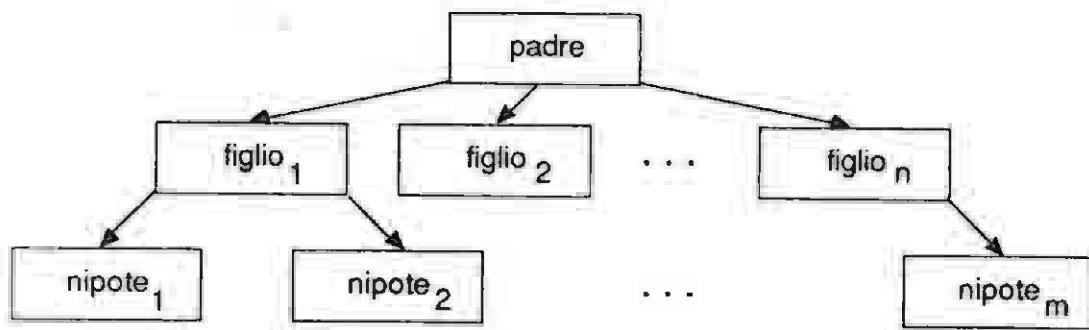


Fig. 6 - Struttura di un programma

semplificata formata soltanto dai programmi *padre*, *figlio₁* e *figlio₂*.

Supponiamo allora di avere a disposizione separatamente i moduli oggetto *padre*, *figlio₁* e *figlio₂*, e che le loro istruzioni facciano rispettivamente riferimento ad indirizzi relativi tra 0 e 300, tra 0 e 120, e tra 0 e 150.

Il correlatore riceve in input questi tre moduli, e genera un unico modulo con riferimento ad indirizzi contigui a partire da un indirizzo simbolico *ind*.

Inoltre, ogni volta che incontra un richiamo ad un modulo, sostituisce ad esso l'indirizzo della prima istruzione del modulo stesso.

Lo schema della fig. 7 fornisce una descrizione dello stato del modulo generato rispetto agli indirizzi di riferimento.

<i>Indirizzo</i>	<i>Contenuto</i>	<i>Commento</i>
<i>ind</i>	inizio <i>padre</i>	
.	
.	salta a <i>ind</i> + 301	attivazione <i>figlio₁</i>
.	
.	salta a <i>ind</i> + 421	attivazione <i>figlio₂</i>
<i>ind</i> + 300	fine <i>padre</i>	
<i>ind</i> + 301	inizio <i>figlio₁</i>	
.	
<i>ind</i> + 420	fine <i>figlio₁</i>	
<i>ind</i> + 421	inizio <i>figlio₂</i>	
.	
<i>ind</i> + 570	fine <i>figlio₂</i>	

Fig. 7 - Schema di programma correlato

Il modulo che è stato generato non ha ormai alcun simbolo irrisolto. Tuttavia non è ancora in una forma eseguibile poiché continua a far riferimento ad indirizzi relativi, e quindi senza specifico riferimento ad effettive zone di memoria. Una versione del programma in questa forma è detta programma *rilocabile*, proprio perché può essere allocato in una zona qualsiasi della memoria, semplicemente fissando il valore dell'indirizzo *ind*.

Una volta che è stato creato il modulo correlato, la fase successiva consiste nella sua trasformazione in un programma caricabile in memoria ed eseguibile. Questa fase, detta di caricamento, è compito del programma *caricatore (loader)* e consiste nello stabilire la zona di memoria in cui caricare il programma assegnando un valore numerico al simbolo *ind*.

Il programma, che fino a questo punto faceva riferimento ad indirizzi relativi, assume ora una forma definitiva ed è detto programma *assoluto* cioè legato ad indirizzi assoluti: esso è immediatamente eseguibile. Si noti che la posizione del programma in memoria dipende in generale dal meccanismo di gestione adottato.

2. SISTEMA OPERATIVO

Il sistema operativo, parte centrale del software di base, gestisce le risorse hardware e software del sistema di elaborazione: il funzionamento delle singole unità del sistema, la comunicazione tra le varie unità, l'assegnazione di ciascuna di esse alle diverse esigenze dei processi di elaborazione proposti.

Esso svolge la funzione di supervisione dell'intero funzionamento dell'elaboratore in modo da svincolare l'utente dalla conoscenza dettagliata e dalla gestione delle singole componenti, facilitando la comunicazione tra l'utente stesso ed il sistema di elaborazione.

Ciascuna delle funzioni del sistema operativo è svolta da uno specifico programma. Non tutti i programmi sono sempre presenti nella memoria centrale, ma vengono trasferiti in essa dalla memoria di massa su cui risiedono, ogniqualvolta è necessaria la loro esecuzione. Una parte del sistema operativo, detta *nucleo*, è generalmente sempre residente in memoria per poter essere attivata immediatamente. Di questo nucleo fa parte il programma *supervisore*, che controlla il funzionamento del processo di esecuzione dei programmi e gestisce l'unità centrale dell'elaboratore.

I sistemi operativi possono essere classificati sia rispetto alle funzionalità che rispetto alle caratteristiche della struttura interna. Nei successivi paragrafi presentiamo queste classificazioni.

2.1. Tipi di sistemi operativi

Rispetto alle funzionalità i sistemi operativi si dividono in *sistemi interattivi* e *sistemi a lotti (batch)*.

In un sistema interattivo l'utente ha a disposizione un terminale di accesso al sistema e può colloquiare utilizzando un linguaggio (detto di controllo), che gli permette di eseguire le diverse fasi della programmazione: creazione e modifica, compilazione, correlazione, caricamento ed esecuzione.

L'alternanza dei diversi passi viene decisa dall'utente in base ai risultati precedenti. Se in un dato passo si verificano degli errori, l'utente può decidere di ripetere alcune operazioni secondo il seguente schema:

1. creazione e/o modifica del programma (con un programma editore di testi);
2. compilazione;
3. se ci sono errori torna al passo 1;
4. correlazione;
5. se ci sono errori (ad esempio perché manca un modulo) torna al passo 1;
6. caricamento;
7. esecuzione (con l'eventuale immissione di dati);
8. se ci sono errori si torna ai passi precedenti, a seconda dei tipi di errori; si torna al passo 7 se si vuole semplicemente eseguire di nuovo lo stesso programma con altri dati.

In un sistema batch l'utente che vuole ad esempio compilare ed eseguire un programma, deve predisporre oltre al programma stesso, le varie richieste che vengono rivolte al sistema operativo, nell'ordine opportuno, inserendo anche i dati di ingresso al programma che si intende eseguire.

In questa modalità di accesso al sistema l'utente deve predefinire l'intera sequenza di operazioni che il sistema dovrà compiere e non ha, in generale, alcuna possibilità di interazione durante le varie fasi di elaborazione richieste. In particolare non può intervenire a fronte di eventuali errori che si presentino durante le fasi intermedie.

Confrontando le due modalità si può osservare che in un sistema interattivo l'accessibilità reale da parte dell'utente all'elaboratore è molto maggiore, quindi il modo di lavorare risulta più flessibile e comodo. In un sistema batch l'utente non può interagire con i processi di elaborazione ed i risultati gli arrivano in un tempo differito.

L'approccio interattivo è senz'altro preferibile durante tutto il periodo di messa a punto di un programma. L'approccio batch è vantaggioso specie per

elaborazioni che richiedono lunghi tempi di esecuzione e grandi quantità di dati in ingresso e di risultati da stampare.

2.2. La struttura dei sistemi operativi

I sistemi operativi possono anche essere classificati rispetto alla struttura interna, sulla base delle tecniche di gestione delle varie risorse. Si parla di sistemi operativi per *monoprogrammazione*, *multiprogrammazione* e *multielaborazione*.

Osserviamo innanzi tutto che la tipologia di un sistema operativo è strettamente legata alla struttura hardware dell'elaboratore e che, allo stesso tempo, esso ne vincola il modo di uso, rendendo più o meno efficiente l'impiego delle componenti hardware disponibili.

Inoltre va ricordato che il sistema operativo è esso stesso costituito da un insieme di programmi, originariamente scritti in un opportuno linguaggio ad alto livello, che vengono eseguiti con le stesse modalità di un qualsiasi altro programma applicativo. Ciò implica che, almeno per gli elaboratori dotati di un'unica unità centrale, in un determinato istante questa unità potrà eseguire una singola istruzione che di volta in volta sarà un'istruzione di un programma del sistema operativo o un'istruzione di un programma applicativo. Nel primo caso si dice che l'elaboratore è nello *stato supervisore*, nel secondo caso si dice che è nello *stato programma*.

Il passaggio da uno stato all'altro avviene sotto il controllo del supervisore, e viene determinato dal verificarsi di condizioni di stato delle unità componenti il sistema hardware. Al verificarsi di una di tali condizioni si determina una *interruzione (interrupt)* dell'esecuzione in corso, e il controllo passa al supervisore. Viene cioè eseguita qualche istruzione del supervisore, attraverso la quale si determineranno i passi successivi dell'elaborazione, come risposta conseguente al tipo di causa che ha generato l'interruzione.

Ad esempio quando un programma viene eseguito, si genera una richiesta d'uso di un'unità periferica ogniqualvolta si deve eseguire un'operazione di input/output. Questa richiesta è fisicamente manifestata assegnando un valore ad un registro. La verifica di questo valore genera il passaggio del controllo al programma supervisore che ha il compito di attivare l'unità relativa all'operazione da eseguire. Durante il tempo di questo trasferimento l'unità centrale è inattiva e sarà riattivata, proseguendo l'esecuzione del programma che ha generato la richiesta di input/output, solo quando quest'ultima operazione sarà

terminata.

Esaminiamo ora la struttura di un sistema operativo in relazione alle sue modalità di funzionamento.

La monoprogrammazione fa riferimento al modo più semplice di gestire un elaboratore. Un sistema monoprogrammato gestisce in modo sequenziale i diversi programmi: in ogni istante il sistema gestisce un solo programma applicativo. Nei sistemi monoprogrammati si ottiene solo un parziale sfruttamento delle risorse hardware, in quanto, a causa della differenza di velocità operativa fra unità centrale e unità periferiche, la prima resta necessariamente inattiva per lunghi intervalli di tempo allorché sono in corso operazioni di input/output.

Ad esempio, in relazione alla situazione mostrata nella fig. 8, l'attività dell'unità centrale resta sospesa negli intervalli di tempo dedicati alle operazioni di input/output.

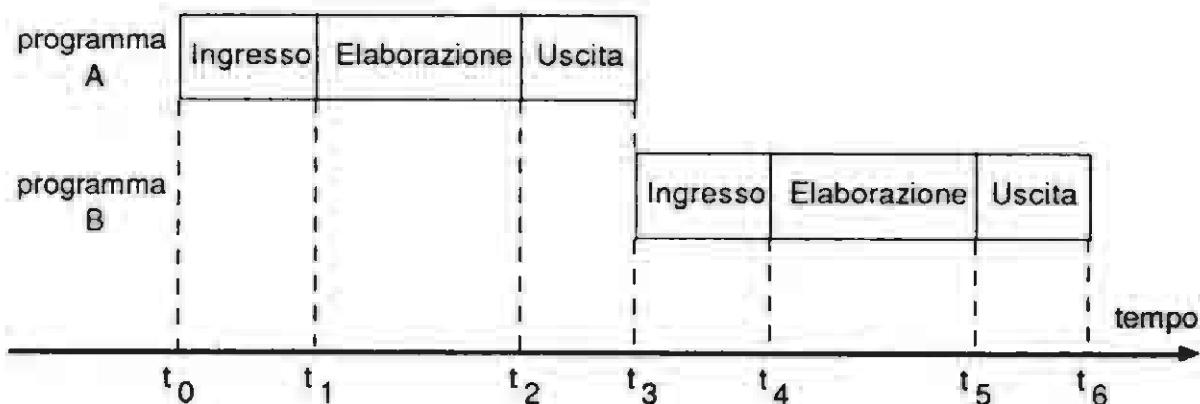


Fig. 8 - Monoprogrammazione

La natura strettamente sequenziale dell'esecuzione dell'unico programma gestito dal sistema non consente di utilizzare i tempi resisi disponibili per l'unità centrale, che per operare di nuovo dovrà necessariamente attendere il completamento dell'operazione precedente.

In un sistema in multiprogrammazione invece si gestiscono simultaneamente più programmi indipendenti, nel senso che ciascuno di essi può iniziare o proseguire l'elaborazione prima che un altro sia terminato.

La multiprogrammazione richiede un'attività molto più complessa da parte del sistema operativo. Ad esempio, ci si riferisca alla situazione in cui nella memoria sono presenti, oltre ad alcune parti essenziali del sistema operativo, i programmi A,B,C e D. Si supponga che A sia in esecuzione. Quando A termina l'esecuzione, oppure è in attesa dell'esecuzione di una operazione di lettura o scrittura, si può passare alla esecuzione di uno degli altri programmi

che sono in memoria per essere eseguiti.

Affinché sia possibile attivare l'esecuzione di un altro programma, mentre il primo è in attesa di completare un'operazione di input/output, occorre che questa sia eseguibile senza l'intervento dell'unità centrale. La struttura dell'elaboratore che abbiamo introdotto nei precedenti capitoli non consente questo tipo di funzionamento e pertanto dovremo analizzare quali modifiche si rendono necessarie da un punto di vista hardware. Affronteremo questo aspetto nel par. 2.4, mentre ora procediamo nella descrizione del funzionamento multiprogrammato di un elaboratore, prescindendo dagli aspetti hardware.

Sia che il programma A termini o che sia in attesa del completamento di un'istruzione di input/output, il controllo viene ceduto al sistema operativo, per decidere quale programma eseguire. Per far ciò il sistema operativo sceglie tra i programmi attivabili (e che non stanno, ad esempio, attendendo il completamento di operazioni di lettura o scrittura) quello con priorità più elevata.

Frequentemente, la priorità è più elevata per classi di programmi che prevedono breve durata, poca occupazione di memoria e con poche operazioni di input/output, e viene dinamicamente incrementata in funzione del tempo di attesa di esecuzione trascorso per un programma.

Supponiamo, nell'esempio di quattro programmi presenti in memoria A, B, C e D, che essi abbiano priorità diverse decrescenti da A a D. Supponiamo anche che le richieste di input/output dei quattro programmi siano riferite a unità diverse in modo da consentire la completa sovrapponibilità di tali operazioni. La fig. 9 mostra la situazione che si crea nell'esecuzione dei programmi con un meccanismo di multiprogrammazione.

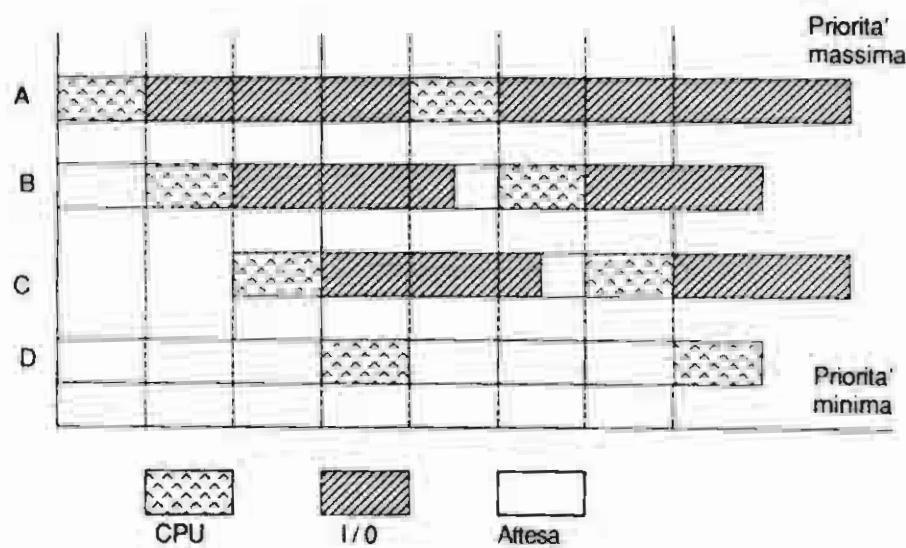


Fig. 9 - Multiprogrammazione

Quando si verifica un'interruzione si passa al programma con priorità maggiore tra i programmi restanti.

Un meccanismo di questo genere tuttavia, essendo strettamente dipendente dalle caratteristiche dei programmi in esecuzione, può creare situazioni di eccessiva penalizzazione per quei programmi che abbiano bassa priorità, come nel caso dell'esempio per il programma D che, peraltro, richiede solo l'uso di unità di calcolo.

Per ovviare a questo inconveniente sono stati individuati altri meccanismi di alteranza dei programmi. Il più importante di questi, noto come *partizione del tempo* (*time-sharing*), consente di assegnare ad ogni programma un quanto di tempo di esecuzione, terminato il quale il programma viene interrotto. In questo caso l'interruzione viene generata da parte del sistema operativo, che riprende il controllo ed attiva un altro programma. Si noti che i quanti di tempo possono essere uguali per tutti i programmi, oppure definiti in modo da tener conto delle priorità.

La multiprogrammazione consente dunque una gestione ottimale dell'unità centrale dell'elaboratore, garantendo un'efficienza molto maggiore nel funzionamento dell'intero sistema da parte di più utenti. Naturalmente ciò viene pagato in termini di un sistema operativo molto più potente, ma anche molto complesso, il quale oltretutto assorbe una buona parte del tempo di calcolo per la propria esecuzione.

Citiamo infine la tecnica della multielaborazione. Questa si ottiene quando più unità centrali (o addirittura più elaboratori) cooperano nella esecuzione dei processi e nella gestione delle risorse. Va notato che le problematiche sulla architettura hardware e software necessarie alla multielaborazione sono di estrema importanza nell'ambito dell'attuale evoluzione delle tecnologie informatiche; esse tuttavia esulano dagli scopi di questo testo.

2.3. La gestione della memoria

Quando un programma termina, oppure ne viene interrotta l'esecuzione, il sistema operativo deve decidere quale altro programma trasferire nella memoria centrale. Ciò dipende, oltreché dalla priorità, anche dalla relazione fra le dimensioni di memoria richieste dai programmi e quella disponibile.

Una soluzione semplice, ma poco flessibile, è quella che partiziona la memoria in aree di dimensioni fisse. Se un programma ha dimensioni maggiori di quella di un'area, è necessario procedere allo spostamento in altre aree di

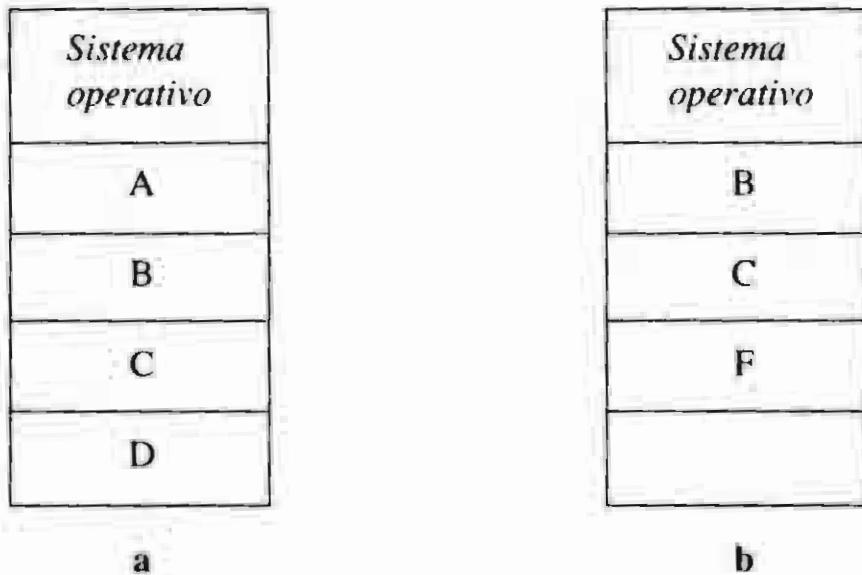


Fig. 10 - Gestione della memoria

programmi già presenti in memoria. Ad esempio con riferimento alla occupazione di memoria illustrata nella fig. 10a, una volta completati A e D, l'inserimento di F comporta la ristrutturazione descritta in fig. 10b.

La necessità di spostare programmi in memoria richiede di definire in modo opportuno le modalità di indirizzamento delle varie istruzioni dei programmi eseguibili.

Ogni spostamento di un programma comporta una sua rilocazione ottenuta con un nuovo processo di caricamento, dovendo ridefinire il valore dell'indirizzo iniziale, e conseguentemente di tutti gli altri indirizzi del programma. Questo processo può avvenire secondo diverse modalità di *rilocazione*. Ci limitiamo qui a fornire una descrizione generale di due classiche tipologie di rilocazione: *statica* e *dinamica*, tenendo conto che le effettive soluzioni adottate nei sistemi operativi esistenti variano da caso a caso e dipendono da numerosi parametri legati sia all'hardware delle macchine che al tipo di sistema operativo.

La rilocazione statica è quella in cui il carico somma al campo indirizzo di ogni istruzione il valore dell'indirizzo a partire dal quale il programma viene memorizzato.

La rilocazione dinamica è invece quella in cui la modifica degli indirizzi avviene all'atto della esecuzione di una istruzione utilizzando opportuni registri dell'unità centrale, ciascuno contenente l'indirizzo a partire dal quale il programma è memorizzato.

Un altro meccanismo di gestione della memoria è la cosiddetta paginazione. Secondo tale meccanismo, la memoria viene suddivisa in porzioni di dimensione fissa, dette pagine fisiche; ed i programmi vengono anch'essi partizionati in blocchi di istruzioni, detti pagine logiche, della stessa dimensione delle pagine fisiche. Quando il programma viene portato in memoria centrale non è necessario che le pagine fisiche da esso occupate siano contigue, bensì le pagine logiche possono disperdersi nella memoria in pagine fisiche secondo un ordine qualunque. In questo caso la memoria può essere gestita secondo un meccanismo analogo a quello della gestione delle strutture collegate.

2.4. Conseguenze della multiprogrammazione sull'architettura

Come abbiamo motivato precedentemente, lo schema funzionale di un elaboratore multiprogrammato presenta significativi cambiamenti rispetto allo schema funzionale a cui abbiamo fatto riferimento finora. La fig. 11 illustra questo nuovo schema, dove per semplicità si è rappresentata una sola unità periferica, per indicare memorie secondarie, organi di ingresso, organi di uscita.

Le componenti di questo schema sono:

- unità centrale (*uc*), che comprende governo centrale ed unità aritmetica e logica;
- memoria, con una propria unità di governo degli accessi alla memoria (*gam*), capace di controllarne l'accesso da più parti, ed eventualmente divisa in più banchi capaci di funzionare indipendentemente gli uni dagli altri;
- canali (*can*), che curano lo scambio di dati fra memoria e unità periferiche;
- unità di governo per unità periferiche (*gup*) che curano ciascuna l'esecuzione delle operazioni di una o di alcune unità periferiche (*up*), ciascuna dotata di una propria memoria locale di transito, detta buffer.

Il governo dell'unità centrale, quando decodifica un'istruzione che richiede l'uso di una unità periferica, delega al canale ed al governo dell'unità periferica cui è collegata l'unità periferica il compito di curarne l'esecuzione quando questa coinvolga lo scambio di dati fra memoria e unità periferica. In questo modo l'unità centrale resta libera per l'esecuzione contemporanea di altre istruzioni di un programma diverso da quello che conteneva la istruzione relativa alla unità periferica. Quando poi questa sarà terminata un apposito segnale di interruzione verrà inviato all'unità centrale che potrà riprendere il programma interrotto.

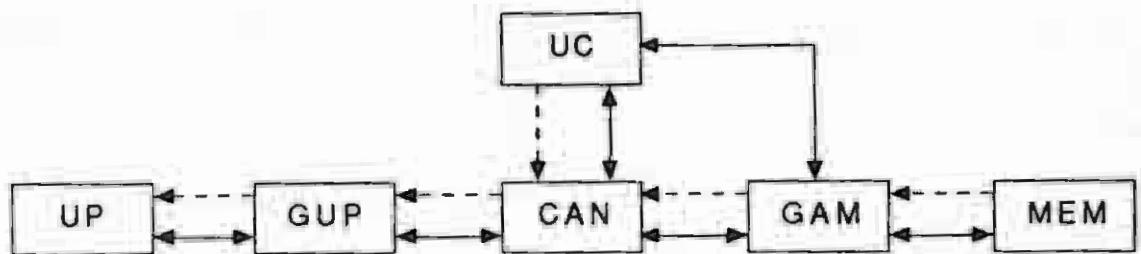


Fig. 11 - Architettura di un elaboratore multiprogrammato.

Risulta allora evidente che le modifiche introdotte permettono in modo effettivo la contemporanea attività dell'unità centrale e delle altre componenti del sistema di elaborazione. La gestione di questo tipo di funzionamento e la risoluzione dei relativi problemi di sincronizzazione dell'uso delle varie risorse sono effettuate dal sistema operativo.

3. AMBIENTE DI PROGRAMMAZIONE

Per poter aiutare l'utente a sviluppare software corretto, sono state messe a punto metodologie e strumenti di ausilio realizzati mediante un insieme di programmi tra loro connessi ed utilizzabili in modo uniforme in tutte le fasi del lavoro di progettazione ed esecuzione del software. Questo insieme prende il nome di *ambiente di sviluppo software* e comprende sia semplici programmi di utilità generale che sofisticati sistemi di controllo della qualità del prodotto.

In questo paragrafo esaminiamo le caratteristiche funzionali di alcuni specifici strumenti di ausilio alla progettazione e gestione dei programmi. In particolare consideriamo gli strumenti, editori di testi, per la creazione e modifica dei programmi, e quelli per la localizzazione e la correzione degli errori.

Ciascuno degli strumenti di un ambiente di programmazione tratta informazioni organizzate in file. È quindi importante esaminare preliminarmente i programmi, compresi nel sistema operativo, per la gestione dei file. L'insieme di questi programmi costituisce il cosiddetto *file system*.

I file sono stati introdotti per semplificare l'accesso alle informazioni contenute in memoria secondaria, consentendo un indirizzamento logico piuttosto che fisico. Nell'indirizzamento logico l'utente, ad esempio, non ha bisogno di far riferimento a indirizzi di settori durante l'uso di un disco, ma può interpretare il disco stesso come uno schedario su cui sono registrati i diversi file a cui è assegnato un nome simbolico.

L'utente può quindi riferirsi ad un file con il suo nome, operare agevolmente con inserimenti e cancellazioni, copiare file da una unità ad un'altra, e può in ogni istante avere a disposizione l'elenco dei file esistenti su una certa unità di memoria di massa, corredati di utili informazioni quali la lunghezza in caratteri, la data di creazione, le ultime modifiche apportate, ecc.

3.1. L'editore di testi

La creazione, memorizzazione e modifica di testi è una delle attività più comuni nell'utilizzo di un elaboratore, in particolare di uso personale. Per questo scopo l'utente ha a disposizione un programma specifico detto programma di redazione di testi (editor).

Un editor deve essere in grado di accettare in ingresso stringhe di caratteri e di memorizzarle in un file. Al file sarà assegnato, come identificatore, un nome scelto dall'utente. Inoltre il programma editor deve essere in grado di correggere testi già memorizzati, utilizzando specifici comandi, e di localizzare sequenze di caratteri.

L'importanza degli editor è andata sempre più aumentando sia perché l'uso di schede perforate, come forma di immagazzinamento di informazioni, è quasi scomparso, sia perché l'uso dell'elaboratore, ed in particolare di elaboratori di tipo personale, ha determinato e via via incrementato le applicazioni di automazione del lavoro di ufficio, di segreteria, ed in generale di composizione di testi.

Una delle caratteristiche principali di un editor sta nel fatto che le informazioni che si inseriscono vengono immediatamente controllate dall'utente che può intervenire con correzioni ove necessario. Le due fasi logicamente distinte di creazione e di correzione di un testo, diventano interconnesse ed alternate in modo interattivo.

Osserviamo infine che la valutazione della qualità di un editor riguarda principalmente la semplicità d'apprendimento e d'uso, la capacità di interagire con i programmi del file system e la velocità di reperimento delle informazioni.

3.2. Il debugger

Un programma *debugger*, è lo strumento software per la localizzazione e la correzione di errori in un programma. Un primo esempio di debugger è quello

basato sull'inserimento di *punti di arresto* (*breakpoint*). Questo metodo, particolarmente adatto per programmi in linguaggi a basso livello, consente di mandare in esecuzione il programma che si arresterà al primo breakpoint incontrato, mantenendo in memoria le informazioni elaborate fino a quel punto.

L'utente può allora analizzare il contenuto delle singole locazioni di memoria, del contatore di programma e dei registri dell'unità centrale, in modo da individuare eventuali errori. Può poi proseguire l'esecuzione che si interromperà al successivo breakpoint, iterando questo procedimento fino alla completa e corretta esecuzione del programma.

Una volta determinato un errore, l'utente può modificare direttamente i contenuti dei registri e della memoria, operando in linguaggio macchina, e procedere nell'esecuzione a partire da una situazione corretta. La correzione non viene riportata sul programma originario, in forma simbolica, per cui, una volta verificata l'esattezza del programma in esecuzione l'utente dovrà riportare le correzioni necessarie sul programma sorgente, ripetendo poi tutte le fasi di traduzione del programma.

Per linguaggi ad alto livello sono stati sviluppati dei debugger appropriati, detti simbolici, che consentono di operare sul formato sorgente simbolico del programma da trattare. Ma le funzioni dei debugger simbolici, si collocano allo stesso livello logico del programma che devono esaminare, e spesso sono scritti nello stesso linguaggio di programmazione. Essi consentono quindi una visibilità che, anziché alle locazioni di memoria e ai registri della macchina, si riferisce alle istruzioni del programma sorgente, alle sue variabili e alle sue strutture di controllo.

10.

Linguaggi di programmazione

I linguaggi di programmazione sono tradizionalmente classificati rispetto alle loro caratteristiche fondamentali che ne determinano il potere espressivo e che si riflettono sullo stile di programmazione.

Finora, in questo testo, ci si è basati sulla presentazione e sull'uso del linguaggio Pascal. In questo capitolo vengono descritte le caratteristiche principali di altri linguaggi di programmazione, secondo una classificazione ormai usuale che permette di confrontare i diversi stili di programmazione possibili. Lo scopo di questa descrizione è unicamente quello di ampliare la visione delle problematiche relative ai linguaggi di programmazione. La forma espositiva è necessariamente succinta e si basa su semplici esempi.

Come è già stato visto in altre parti di questo testo, il linguaggio base di un elaboratore, detto linguaggio macchina, è un linguaggio di programmazione che consente di specificare in un programma solo le operazioni che l'elaboratore può eseguire direttamente, attraverso una forma sintattica molto elementare basata sull'uso di un codice di tipo numerico, binario. Esso è diverso per ciascun elaboratore essendo dipendente dalle sue caratteristiche architettoniche, ed in particolare da quelle tipiche degli elaboratori basati sul modello di Von Neumann: elaboratori capaci di eseguire operazioni in modo sequenziale, utilizzando una memoria da cui prelevare la specifica di ogni operazione assieme ai suoi operandi, ed in cui memorizzare il risultato.

Questo tipo di linguaggio è quindi più fortemente orientato alla macchina, piuttosto che ai problemi trattati. Per questa ragione esso viene detto linguaggio a basso livello, cioè più vicino al livello hardware.

Una prima evoluzione dei linguaggi a basso livello si è avuta con l'introduzione di linguaggi, ancora orientati alla macchina nel senso detto, ma di tipo simbolico. Questi linguaggi, detti assemblativi, pur mantenendo le caratteristiche e gli inconvenienti dei linguaggi binari, consentono tuttavia, attraverso

l'uso di descrizioni simboliche delle variabili e dei codici delle istruzioni, un significativo passo in avanti verso l'introduzione di linguaggi più potenti ed espressivi.

Linguaggi più evoluti sono stati successivamente definiti verso la metà degli anni '50, come tentativo di astrazione dalla macchina, cioè con lo scopo di permettere di specificare operazioni di più alto livello rispetto a quelle elementari che l'elaboratore può eseguire direttamente. Questi linguaggi sono detti ad alto livello di tipo imperativo; esempi sono il Basic, il Fortran ed il Pascal.

Tuttavia, in questo tentativo di astrazione, mentre si è raggiunta una certa indipendenza dal particolare elaboratore, non si è raggiunta l'indipendenza dal modello di Von Neumann: i programmi sono una sequenza di istruzioni attraverso le quali si specificano le modifiche dei diversi contenuti delle aree di memoria, e l'evoluzione del calcolo è essenzialmente rappresentata da una variazione di stato della memoria ottenuta come *effetto* delle istruzioni eseguite.

Una svolta decisiva nel campo dei linguaggi di programmazione si è avuta quando ci si è posti il problema di definire il linguaggio prescindendo inizialmente dal fatto che i programmi che con esso si possono costruire debbano poi essere eseguiti su un elaboratore, e ponendosi direttamente come obiettivo quello di fornire un mezzo espressivo per specificare all'elaboratore il compito da eseguire in modo semplice e sintetico.

Sono stati così definiti una serie di linguaggi, di impostazione completamente diversa dalla precedente, per cui l'esecuzione di un programma può essere considerato, ad esempio, come il calcolo del valore di una funzione (come nei linguaggi funzionali, tipo il Lisp), oppure come la dimostrazione della veridicità di una asserzione (come nei linguaggi dichiarativi logici, tipo il Prolog).

Attualmente esistono diversi tipi di linguaggi, come vedremo nella breve classificazione che segue, ciascuno dei quali ha suoi propri paradigmi che garantiscono forme espressive appropriate solo per alcuni dei concetti e dei meccanismi fondamentali della programmazione. È infatti proprio questa specificità dei singoli linguaggi che ne ha favorito l'enorme proliferazione. Il fenomeno è inevitabile ed inarrestabile ed è intrinseco alla natura del linguaggio come strumento di comunicazione.

Nel resto di questo capitolo ci si propone, in particolare, di offrire una panoramica dei diversi tipi di linguaggi di programmazione ad alto livello esistenti. Pertanto in esso vengono largamente ignorati i problemi relativi alla implementazione, ed i riferimenti a linguaggi specifici saranno limitati a quelli più comunemente usati.

1. I LINGUAGGI IMPERATIVI

Alla classe dei linguaggi imperativi appartengono quelli basati sulla nozione di istruzione di macchina e di memorizzazione di valori in celle di memoria.

Un programma scritto in linguaggio macchina è costituito da un elenco di istruzioni che l'elaboratore esegue in sequenza e che realizzano tre diversi tipi di operazioni:

- il trasferimento di dati tra unità centrale e memoria;
- le operazioni aritmetiche eseguibili dall'unità di calcolo;
- l'alterazione della normale sequenza di esecuzione delle istruzioni, per mezzo di salti, condizionati o meno, o con l'arresto del programma.

Nei linguaggi imperativi ad alto livello ritroviamo i concetti su cui è basato il linguaggio macchina. Infatti le variabili sono astrazioni della cella di memoria e l'istruzione di assegnazione corrisponde al trasferimento di un valore in memoria.

Dal punto di vista della strutturazione dei dati in un linguaggio imperativo il programmatore ha a disposizione un insieme di tipi di dato primitivi e di meccanismi di composizione che gli consentono di organizzare in modo agevole i dati relativi al suo problema. Questa tematica è stata ampiamente trattata nei capp. 2 e 3.

Per quanto riguarda la strutturazione del controllo è stato individuato un insieme di primitive come l'iterazione e la selezione che, consentendo la realizzazione di qualsiasi flusso di controllo, definiscono in modo chiaro e leggibile la sequenza delle istruzioni da eseguire. Della strutturazione del controllo, e della sua rilevanza per la produzione di software di buona qualità, si è già parlato nei precedenti capitoli, ed in particolare nel cap. 4 sulle metodologie di programmazione.

Come già detto, esempi di linguaggi imperativi ad alto livello sono il Basic, il Fortran e il Pascal. Non è necessario riportare qui ulteriori esempi di programmi Pascal, mentre è utile fornire un'indicazione più precisa delle caratteristiche degli altri due linguaggi. Presentiamo quindi, nelle figg. 1 e 2, un programma Basic ed uno Fortran per la soluzione di un semplice problema quale quello della somma dei numeri dispari in un vettore di $N \leq 100$ elementi interi.

Si noti che il programma impiega un sottoprogramma per la somma dei termini dispari del vettore, che si richiama con l'istruzione GOSUB indicando semplicemente il numero della riga di inizio. I sottoprogrammi non hanno parametri e i valori sono passati attraverso variabili globali. L'istruzione di

```
100 DIM T(100)
200 READ N
300 FOR I = 1 TO N
400 READ T(I)
500 NEXT I
600 GOSUB 1100
700 PRINT S
800 GOTO 2000

1100 REM S = SOMMA DEGLI ELEMENTI DISPARI IN T(1..N)
1200 LET S = 0
1300 FOR I = 1 TO N
1400      IF NOT ODD(T(I)) THEN GOTO 1600
1500      LET S = S + T(I)
1600 NEXT I
1700 RETURN

2000 END
```

Fig. 1 - Programma Basic per la somma dei valori dispari di un vettore

```
INTEGER T(100), S, N
READ (5,10) N
DO100 I=1,N
    READ T(I)
100 CONTINUE
    S = SOMMA (N,T(N))
    WRITE (6,11) S
10 FORMAT(I5)
11 FORMAT(10X,'SOMMA DEGLI ELEMENTI DISPARI = ',I6)
STOP
END

INTEGER FUNCTION SOMMA(M, T(M))
INTEGER T
DO10 K=1,M
IF (REM(T(I),2).NE.2) GO TO 10
    SOMMA = SOMMA + T(K)
10 CONTINUE
RETURN
END
```

Fig. 2 - Programma Fortran per la somma dei valori dispari di un vettore

assegnazione è realizzata mediante il costrutto LET. Le dichiarazioni di tipo delle variabili sono implicite nella scelta degli identificatori, a meno che le variabili siano indicizzate come per il vettore T. In questo caso si fa uso della dichiarazione DIM. L'istruzione FOR...NEXT realizza una iterazione definita.

Si noti che il programma Fortran impiega un sottoprogramma di tipo funzione per il calcolo della somma dei termini dispari. Ad esso ci si riferisce mediante il nome e la lista degli argomenti. In Fortran il passaggio di parametri è sempre per riferimento. L'istruzione di assegnazione è realizzata mediante l'operatore =. La dichiarazione di tipo delle variabili e delle funzioni può essere implicita nella scelta degli identificatori (gli identificatori che iniziano con le lettere I, J, K, L, M, N sono usati per il tipo intero, tutti gli altri sono intesi riferiti al tipo reale, se non specificato altrimenti in modo esplicito). L'istruzione DO..CONTINUE realizza un'iterazione definita. Le istruzioni di ingresso e uscita sono guidate da una esplicita dichiarazione del formato che fa uso di identificatori chiave per indicare la tipologia dei dati trattati; ad esempio l'identificatore I è usato per i dati di tipo intero. Si noti anche che la funzione REM in Fortran calcola il resto della divisione tra valori interi.

Per meglio caratterizzare la classe dei linguaggi imperativi illustriamo, attraverso un esempio, come essi trattano l'astrazione funzionale.

Consideriamo la funzione che calcola il massimo tra due valori reali; essa può essere costruita, come in fig. 3, rispettivamente in Fortran e Pascal.

Queste definizioni realizzano funzioni senza effetti collaterali e sono pertanto vicine alla nozione matematica di funzione. Esse consentono, ad esempio in Fortran, di comporre la funzione *max* in un'istruzione di assegnazione del tipo

$$M = \max(\max(A,B),C)$$

per determinare il più grande di tre valori *A*, *B*, e *C*.

Volendo invece realizzare il calcolo del massimo mediante una procedura possiamo definirla in modo tale che l'attivazione $\max(A,B,M)$ calcoli il più grande tra i valori *A* e *B* ed assegni il risultato ad un terzo parametro *M*. Possibili soluzioni, rispettivamente in Fortran e Pascal sono riportate in fig. 4.

Il calcolo del più grande tra tre valori è allora richiesto con l'attivazione $\max(A, B, M)$ seguita dall'attivazione $\max(M,C,M)$.

A questo punto si potrebbe dire che abbiamo fatto una variazione soltanto di tipo sintattico, mentre in effetti esiste anche una significativa differenza semantica. Infatti, come già visto nel cap. 2, nell'uso delle procedure esiste la possibilità di occupare parti di memoria oltre quelle destinate ai valori di ingresso e ai risultati. Se per calcolare il massimo di tre valori si procede con

```
REAL FUNCTION MAX(X,Y)
REAL X,Y
IF(X.GE.Y) GO TO 10
MAX= Y
RETURN
10 MAX = X
RETURN
END

function max(x,y: real):real
begin
  if x >= y then max := x else max := y
end
```

Fig. 3 - Calcolo del massimo, prima versione

```
SUBROUTINE MAX(X,Y,Z)
REAL X,Y,Z
IF(X.GE.Y) GO TO 10
Z = Y
RETURN
10 Z = X
RETURN
END

procedure max(x,y: real; var z: real);
begin
  if x >= y then z := x else z := y
end
```

Fig. 4 - Calcolo del massimo, seconda versione

le successive attivazioni $\max(A,B,N)$ e $\max(N,C,M)$, si ottiene di nuovo il risultato come valore del parametro M, ma nella locazione di memoria identificata con N resta il valore del risultato intermedio.

Ciò evidenzia la profonda differenza semantica delle due soluzioni adottate, mediante rispettivamente funzioni e procedure, per la realizzazione di una funzione matematica.

Notiamo anche che l'uso di procedure può allontanare ulteriormente dalla nozione di funzione matematica. Ad esempio, se si definisce *max* come una procedura tale che l'attivazione *max(A,B)* calcoli il valore più grande tra *A* e *B* e lo assegna ad *A* come nel codice seguente. In fig. 5 abbiamo ancora rispettivamente in Fortran e Pascal, la nuova versione.

SUBROUTINE MAX(X,Y)

REAL X,Y

IF (X.GE.Y) GO TO 10

X = Y

10 RETURN

END

procedure max (var x: real; var y: real);

begin

if *x* < *y* **then** *x* := *y*

end

Fig. 5 - Calcolo del massimo, terza versione

A questo punto le attivazioni *max(A,B)* e *max(A,C)* in sequenza determinano il più grande tra tre valori lasciando il risultato in *A*, ma hanno sovrascritto il valore originale di *A*. Di nuovo potremmo dire che il significato di funzione non è perso, poiché il valore del risultato è determinato univocamente dai valori iniziali, ma le modifiche risultanti sulla memoria, come effetti collaterali, potrebbero determinare conseguenze indesiderabili circa la correttezza del programma.

Programmi costruiti componendo funzioni matematiche definiscono semplicemente come i nuovi valori debbano essere ottenuti a partire dai vecchi. Le procedure viste finora comunque, non solo calcolano valori, ma hanno anche l'effetto secondario di assegnare valori ad uno dei loro parametri. Usando queste procedure dobbiamo sempre tener conto dei successivi cambiamenti delle variabili determinati dalle successive assegnazioni. Ciò rappresenta il cosiddetto "calcolo per effetti" piuttosto che il "calcolo di valori". È comunque evidente che ci possono essere procedure che implementano funzioni matematiche, che si limitano cioè a calcolare valori, senza produrre altri effetti.

Per completare questo confronto esaminiamo un'ultima versione, vedi fig. 6, delle procedure viste in cui il calcolo procura effetti che alterano i valori di

```
SUBROUTINE MAX(X,Y)
REAL X,Y
COMMON M
REAL M
IF(X.GE.Y) GO TO 10
M = Y
RETURN
10 M = X
RETURN
END

procedure max(x,y: real);
begin
  if x >= y then m := x else m := y
end
```

Fig. 6 - Calcolo del massimo, quarta versione

variabili che non sono parametri della procedura, sempre rispettivamente in Fortran e Pascal.

In questo caso l'uso della variabile globale M crea una difficoltà di interpretazione dei programmi, come ad esempio nelle attivazioni successive *max(A,B)* e *max(M,C)* per determinare il più grande tra tre valori.

Pur con le loro limitazioni, tuttavia, i linguaggi imperativi sono di gran lunga i più usati e, sul mercato, si assiste a fenomeni come la diffusione del Basic su elaboratori personali. Il Basic, infatti, viene di solito considerato un linguaggio semplice da imparare, in quanto costituito da poche nozioni primitive. Ma, in questo caso semplice da imparare non vuole affatto dire semplice da usare: le primitive offerte dal linguaggio non consentono una interazione uomo-macchina di livello adeguato. Il Basic diventa uno strumento estremamente difficile da usare non appena il problema da risolvere si complica, rivelando così la propria carenza di espressività.

Nell'ambito della programmazione per applicazioni di tipo scientifico il linguaggio Fortran, introdotto nella metà degli anni '50, continua ad avere un ruolo prevalente, sia per le sue caratteristiche di efficienza e di aderenza al linguaggio scientifico, matematico, sia perché la disponibilità di raccolte di programmi applicativi, efficienti, robusti ed affidabili, costruiti in Fortran, condiziona spesso la scelta del linguaggio di programmazione.

Alla classe dei linguaggi imperativi appartengono inoltre i linguaggi della famiglia Algol che, ad eccezione del Pascal, sono poco noti, ma che hanno avuto una notevole influenza sull'evoluzione dei linguaggi di programmazione. Gli ultimi discendenti di questa famiglia sono Mesa ed Ada che offrono molte delle caratteristiche emerse nel corso degli anni e sono immersi in ambienti di programmazione ricchi di strumenti per lo sviluppo dei programmi.

2. I LINGUAGGI FUNZIONALI

Le differenze tra i vari linguaggi imperativi, prevalentemente riguardanti la diversa strutturazione dei dati e del controllo, che pur sono considerevoli, sono meno significative del fatto che tutti sono basati sullo stile di programmazione proprio del modello di Von Neumann. L'evoluzione del calcolo in questo modello è, come abbiamo visto, essenzialmente basato sugli effetti procurati dall'esecuzione di una sequenza di istruzioni.

Uno stile di programmazione completamente diverso, e non legato alla particolare architettura di macchina di Von Neumann, è quello offerto dai linguaggi per la programmazione funzionale. Questi linguaggi sono basati sul concetto di funzione matematica e su quello di applicazione di una funzione ad argomenti. Per questo motivo sono spesso detti anche linguaggi applicativi.

Alla fine degli anni '50 J. McCarthy diede la definizione del primo linguaggio funzionale effettivamente implementato e utilizzato: il Lisp (acronimo per List Processing). L'esigenza che questo linguaggio doveva soddisfare era quella di consentire di manipolare agevolmente informazioni di tipo simbolico.

L'esecuzione di un programma scritto in un linguaggio funzionale consiste nella computazione del valore di una espressione. Se tale espressione è una variabile o una costante il risultato del programma è semplicemente dato dal corrispondente valore. Se invece l'espressione è un'applicazione, cioè una coppia <funzione, lista di argomenti>, il risultato è il valore che la funzione assume, dati i valori degli argomenti.

Ad esempio la funzione *max*, che calcola il valore massimo tra due valori dati, può essere applicata ai valori interi *a* e *b* per determinarne il massimo, nel modo seguente

$$\max(a,b)$$

Inoltre si deve considerare che mediante l'applicazione di funzioni si possono scrivere programmi via via più significativi: il valore di un argomento

di una funzione può infatti essere a sua volta ottenuto dalla valutazione di una applicazione.

Ad esempio, si può applicare la funzione *max* ai valori ottenuti dall'applicazione della stessa funzione *max* a due coppie di valori (a, b) , e (c, d) , nel modo seguente

$$\max(\max(a,b),\max(c,d))$$

Si può allora definire una nuova funzione, di quattro argomenti, per calcolare il maggiore tra quattro valori dati. Se si assegna un nome a questa nuova funzione, *largest*, nel modo seguente

$$\text{largest}(x,y,w,z) <== \max(\max(x,y),\max(w,z))$$

dove il simbolo $<==$ significa "è definita come", la si può usare direttamente applicandola ai suoi argomenti, nel modo seguente

$$\text{largest}(a,b,c,d)$$

In tal modo si estende la classe di funzioni che si possono utilizzare, applicandole a valori fissati, oppure usandole per definire nuove funzioni.

L'insieme delle espressioni che caratterizzano un linguaggio funzionale è ampliato con il meccanismo di definizione di funzione per casi, mediante l'uso del costrutto condizionale, cioè di un'espressione del tipo

if *predicato*
 then *espressione1*
 else *espressione2*

oppure del tipo

cond
 predicato1 espressione1,
 predicato2 espressione2,

 predicaton espressionen

in cui i predicati sono espressioni che possono assumere i valori di verità vero e falso.

Il calcolo del valore di una funzione può anche richiedere la valutazione di una applicazione della funzione stessa: in tal caso si ha una definizione ricorsiva. Ad esempio possiamo definire nel modo ricorsivo usuale la funzione per il calcolo del fattoriale di un numero n , come segue:

```

fattoriale(n) <== if n = 0
    then 1
    else n * fattoriale(n - 1)

```

Nella definizione compaiono i simboli $=$ e $*$ che si riferiscono rispettivamente al predicato di uguaglianza ed alle funzioni di moltiplicazione e sottrazione definite nel tipo di dato dei numeri naturali.

La valutazione dell'applicazione $fattoriale(3)$ procede secondo il noto schema di valutazione di una funzione definita ricorsivamente, e pertanto richiede la valutazione di

$$3 * fattoriale(2).$$

che a sua volta richiede quella di

$$3 * 2 * fattoriale(1).$$

La valutazione termina quando l'argomento della applicazione della funzione è 0 e viene selezionata l'espressione del ramo **then**, la cui valutazione non richiede quella di altre applicazioni. Il risultato complessivo è dato dal prodotto

$$3 * 2 * 1 * 1.$$

L'evoluzione del calcolo, quindi, è quella tipica del calcolo per valori, in cui valori calcolati da una funzione vengono forniti come argomenti ad altre funzioni, e così di seguito fino a determinare il valore finale, soluzione di un dato problema. Questo meccanismo di applicazione funzionale evidenzia come lo stile di programmazione proprio dei linguaggi funzionali non faccia riferimento all'architettura di macchina tipica del modello di Von Neumann, ma al solo concetto di definizione e di calcolo di una funzione.

Un linguaggio di programmazione necessita di un tipo di dato sufficientemente potente da permettere la rappresentazione di altri tipi di dato. Usualmente i linguaggi funzionali sono definiti sul tipo di dato delle liste. Queste garantiscono la flessibilità necessaria per esprimere adeguatamente strutture anche molto complesse. Per mezzo di liste possono essere facilmente rappresentati i programmi scritti nel linguaggio stesso, offrendo in tal modo la possibilità di scrivere agevolmente programmi che a loro volta operano su programmi.

Possiamo concludere quindi che i linguaggi funzionali hanno la caratteristica di non calcolare per effetti, ma di calcolare valori. Il nucleo di un linguaggio funzionale è costituito dal meccanismo di applicazione di funzioni, dalla ricorsione e dal costrutto condizionale, come strumenti di controllo, e dalle liste

```
(defun sommadispari (termini)
  (cond
    ((null termini) 0)
    ((odd (car termini)) (plus (car termini)
                                 (sommadispari (cdr termini))))
    (t (sommadispari (cdr termini))))
```

Fig. 7 - Funzione sommadispari

come tipo di dato primitivo. Si noti anche che l'aderenza che questi linguaggi hanno con il concetto matematico di funzione favorisce una più semplice prova di correttezza dei programmi.

Di seguito è riportato il programma Lisp per il problema della somma dei valori dispari di un vettore, precedentemente presentato in Basic e Fortran, seguito dalla traccia di una sua esecuzione. Si noti che i programmi Lisp usano una notazione prefissa e che essi sono, come già detto, rappresentati mediante liste multiple.

Si definisce, mediante la funzione *defun*, la funzione *sommadispari* (v. fig. 7); essa opera sulla lista *termini*, che rappresenta il vettore dei dati.

Le funzioni *car* e *cdr* sono le usuali funzioni base delle strutture di lista, il predicato *odd* verifica se un oggetto è un numero dispari, mentre la funzione *plus* calcola la somma di interi, e la costante *T* denota il valore di verità vero.

Applicando la funzione *sommadispari* al vettore di quattro elementi 23 34 7 e 9, mediante l'attivazione seguente:

(sommadispari (23 34 7 9))

si ottiene la sequenza di richieste di valutazione di funzioni, riportata in fig. 8, in cui l'indentazione sottolinea il meccanismo di valutazione ricorsiva della funzione *sommadispari*.

Uno dei più significativi esempi di programmi scritti in un linguaggio funzionale è l'interprete per il linguaggio stesso. In generale, la costruzione degli strumenti dell'ambiente di programmazione sfrutta questa possibilità. Ciò ha fatto sì che proprio intorno ai linguaggi funzionali si siano sviluppati i primi ambienti di programmazione. Al riguardo si può ricordare il sistema Interlisp come una delle realizzazioni più significative.

I linguaggi funzionali si sono diffusi inizialmente negli ambienti in cui maggiormente era sentita la necessità di uno strumento adatto alla manipola-

zione di simboli ed in particolare nelle applicazioni di Intelligenza Artificiale. Attorno ai linguaggi funzionali sono stati costruiti sistemi di programmazione in grado di offrire al programmatore un'interazione di livello elevato e quindi di facilitare lo sviluppo di programmi di notevoli complessità e dimensioni.

La presenza di un ambiente di programmazione in grado di assistere il programmatore in tutte le fasi della messa a punto di un programma ha contribuito, al pari delle caratteristiche proprie del linguaggio, ad incentivare uno stile di programmazione noto col nome di raffinamento incrementale.

Questo si basa, oltre che sulla scomposizione del progetto in molte funzioni, sulla possibilità di mantenere un elevato livello di astrazione, ritardando le decisioni relative alla realizzazione delle strutture dati. In tal senso la flessibilità offerta dalle liste, sfruttata in modo opportuno, consente di rendere i programmi largamente indipendenti dalla rappresentazione usata per le strutture dati stesse.

Il processo di programmazione per raffinamento incrementale si avvale inoltre degli strumenti offerti dall'ambiente. In esso, oltre ai tradizionali interprete e compilatore per il linguaggio, sono spesso disponibili altri strumenti in grado di permettere l'esecuzione di programmi parzialmente definiti, di sospendere l'esecuzione di un programma e riprenderla dopo che esso è stato modificato ecc. In questo modo, programmare risulta essere un'attività altamente interattiva e gran parte delle inefficienze legate al ciclo redazione-compilazione-esecuzione vengono eliminate.

Il Lisp, in numerose versioni e dialetti proliferati grazie alla possibilità di estendere e modificare con facilità il linguaggio stesso, è senza dubbio il linguaggio funzionale più diffuso. Va tuttavia precisato che esso normalmente consente anche l'uso di istruzioni tipiche dei linguaggi imperativi, quali l'iterazione o l'assegnazione che, se da una parte portano ad una migliore efficienza, dall'altra, compromettono, in molti casi, la chiarezza e la leggibilità del programma. Tra gli altri linguaggi funzionali ricordiamo lo Scheme, il Pop-2 e l'Apl.

3. I LINGUAGGI DICHIARATIVI BASATI SULLA LOGICA

La logica è uno strumento di espressione formale che trae origine dall'esigenza di formalizzare il ragionamento umano. Fin dalla comparsa dei primi elaboratori è stato affrontato il problema della meccanizzazione dei processi deduttivi propri della logica matematica ed in particolare del calcolo dei

```
(sommadispari (23 34 7 9))
  (plus 23 (sommadispari (34 7 9)))
    (plus 23 (sommadispari (7 9)))
      (plus 23 (plus 7 (sommadispari (9))))
        (plus 23 (plus 7 (plus 9 (sommadispari ()))))
          (plus 23 (plus 7 (plus 9 0)))
            (plus 23 (plus 7 9))
              (plus 23 16)
```

39

Fig. 8 - Valutazione della funzione sommadispari

predicati (si veda anche il paragrafo 5 dell'appendice). Questo ha portato alla individuazione di sistemi logici equivalenti al calcolo dei predicati, caratterizzati da meccanismi deduttivi realizzabili su un elaboratore.

L'obiettivo inizialmente perseguito è stato quello di realizzare dimostratori automatici di teoremi, e successivamente questi studi hanno aperto la strada alla possibilità di usare la logica come linguaggio di programmazione.

Scrivere un programma in un linguaggio di programmazione basato sulla logica richiede i seguenti passi:

- definizione del problema da risolvere mediante formule del linguaggio;
- interrogazione del sistema automatico che è in grado di fare deduzioni sulla base della definizione fornita.

Il sistema automatico in questione è in genere costituito da un interprete in grado di dedurre, mediante l'applicazione di un principio di deduzione, altre verità a partire da quelle specificate.

Programmare in un linguaggio basato sulla logica risulta agevole in quanto tutto ciò che è richiesto al programmatore è di comunicare al sistema automatico (formalizzandole) le proprie conoscenze relative ad un certo problema. La programmazione segue uno stile puramente dichiarativo. Successivamente, per attivare il processo deduttivo, è sufficiente porre dei quesiti al sistema.

Linguaggi di programmazione di questo tipo sono molto utili specie in fase di progetto di programmi complessi, poiché permettono di procedere per raffinamenti successivi. Si noti anche che per i programmi in tali linguaggi è agevole effettuare una verifica di correttezza secondo metodi matematici.

Il linguaggio di programmazione Prolog costituisce un esempio di linguaggio basato su un sottoinsieme del calcolo dei predicati del primo ordine.

Un programma Prolog è costituito da un insieme di asserzioni (clausole) condizionate o meno. Una asserzione condizionata (regola) assume la forma

A IF B,C,...,D

(che si legge A è vero se lo sono B, C, ..., D), dove A, (conseguente) e B, C, e ... D (antecedenti) sono predicati applicati ad argomenti (letterali). A è anche detta conclusione, mentre B, C,..., D, dette premesse, costituiscono complessivamente l'ipotesi della asserzione considerata.

Se non esistono antecedenti si ha che la conclusione A è vera senza alcuna ipotesi, cioè A è un'asserzione incondizionata (fatto).

Una asserzione incondizionata esprime una relazione tra individui o oggetti di una fissata classe. Ad esempio, l'affermazione "Mario beve birra" è esprimibile applicando il predicato Beve alle costanti Mario e Birra, cioè mediante l'asserzione incondizionata Beve(Mario,Birra). Si possono anche avere asserzioni che esprimono relazioni riguardanti classi di individui. Ad esempio l'affermazione "Mario beve bibite" è esprimibile come Beve(Mario,x), dove x è un simbolo di variabile che indica un qualsiasi individuo, elemento, della classe Bibite.

Si noti che la notazione adottata prevede che simboli di costante siano denotati da identificatori con lettera iniziale maiuscola, mentre i simboli di variabile siano denotati da identificatori con lettere minuscole.

La forma assunta da una interrogazione del sistema (goal) è del tipo seguente:

? A,B,C,...,D.

dove A,B,C,...,D sono letterali. Il significato del goal è il seguente: dimostra, in base alle tue conoscenze, cioè in base alle asserzioni disponibili, che tutte le formule A,B,C,...,D sono vere, e ritorna come risultato i diversi valori assunti da ciascuna delle variabili che in essi compaiono al termine della dimostrazione. Nel caso in cui la dimostrazione non abbia un esito positivo, il risultato della computazione è un opportuno messaggio al programmatore.

Come esempio di programmazione in Prolog consideriamo il problema del calcolo del fattoriale di un numero, espresso con le due clausole seguenti:

Fatt(0,1)

Fatt(z,w) IF Sott1(z,z1),Fatt(z1,w1),Prod(z,w1,w)

La prima clausola, che è un'asserzione incondizionata, cioè un fatto, dice che il fattoriale di 0 è 1. La seconda dice che il fattoriale di un numero z è un numero w se sottraendo 1 a z si ottiene z1, e il fattoriale di z1 è w1 e w è il prodotto tra

z e w1.

Con questa specifica si possono risolvere due tipi di problemi:

- "calcola il fattoriale di un numero". Ad esempio per calcolare il fattoriale di 3, si richiede l'interrogazione ?Fatt(3,x). Il risultato sarà il valore di x che rende vera tale formula;
- "calcola quel numero (o quei numeri) il cui fattoriale è un numero dato". Ad esempio per calcolare il numero il cui fattoriale è 6, si richiede l'interrogazione ?Fatt(x,6). Il risultato sarà il valore della variabile x, se esiste, che rende vera l'interrogazione posta.

Come si vede dall'esempio precedente, una caratteristica molto utile dei programmi logici è quella di essere invertibili, potendo scambiare il ruolo dei parametri di ingresso con quelli di uscita.

Consideriamo ora un secondo esempio di programma Prolog, rappresentato da un insieme di clausole:

```
Padre(Enzo,Mario)
Padre(Andrea,Sara)
Madre(Sara,Laura)
Madre(Livia,Enzo)
Bella(Laura)
Nonno(x,z) IF Padre(x,y),Padre(y,z)
Nonno(x,z) IF Padre(x,y),Madre(y,z)
```

Comunicate queste conoscenze al sistema Prolog, l'utente può chiedere ad esso di individuare quella persona se esiste, (o quelle persone se esistono), che sia nipote di Andrea ed è bella, nella forma di un'interrogazione quale:

?Nonno(Andrea,nipote),Bella(nipote)

oppure può chiedere il nome di una bella ragazza, nella forma

?Bella(ragazza)

oppure ancora il nome di una persona che è nonno di una bella ragazza, nella forma

?Nonno(nonno,nipote),Bella(nipote).

Sfruttando alcune delle asserzioni che gli sono state comunicate, l'interprete individuerà i valori delle variabili nonno, nipote e ragazza, ritornando tali valori come risultato della computazione.

Concludiamo questo paragrafo proponendo un ultimo semplice esempio di

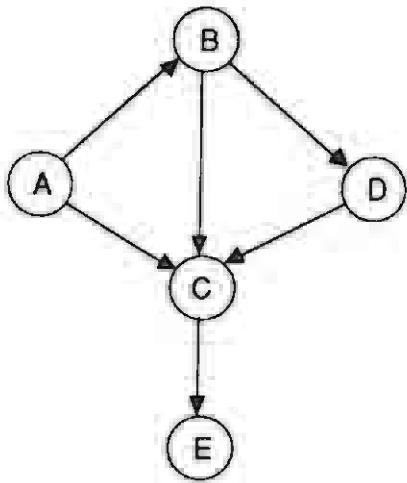


Fig. 9 - Grafo orientato

programma Prolog, che può essere confrontato facilmente con le corrispondenti versioni in linguaggi imperativi, al fine di sottolineare le differenze dello stile di programmazione. Il problema preso in esame è quello della ricerca di cammini nei grafi. Supponiamo di voler determinare i cammini dal nodo A al nodo E nel grafo orientato della fig. 1.

Un semplice programma potrebbe essere il seguente

```

go(A)
arc(A,B) arc(B,C) arc(D,C)
arc(A,C) arc(B,D) arc(C,E)
go(y) IF arc(x,y), go(x)

```

dove go(x) significa che il processo di ricerca può "andare" al nodo x , e arc(x,y) significa che esiste un arco da x a y . Si noti che go(A) afferma che A è raggiungibile, mentre gli altri fatti rappresentano la struttura del grafo dato. Ponendo l'interrogazione

?go(E)

si può verificare la raggiungibilità del nodo E , a partire dal nodo A . Questa interrogazione genera una serie di altre interrogazioni che portano tutte ad un esito positivo, e si determinano così i diversi cammini da A a E . Per esempio il cammino $A \rightarrow C \rightarrow E$ viene individuato in questo modo: dall'interrogazione iniziale

?go(E)

si genera, in base alla regola posta, in cui si è sostituita la variabile y con il valore

E, l'interrogazione

?arc(x,E),go(x)

e quindi successivamente le interrogazioni

?go(C)

?arc(x,C),go(x)

?go(A)

che ha esito positivo in quanto go(A) è un'asserzione incondizionata. In tal modo il percorso A C E è individuato sulla base dei successivi valori assunti dalla variabile x.

Appendice

1. INSIEMI, RELAZIONI E FUNZIONI

1.1. Insiemi

Un *insieme* è una collezione di oggetti. Ciascuno di essi è detto elemento dell'insieme. Indichiamo mediante il simbolo \in l'appartenenza dell'elemento x all'insieme A . Cioè,

$$x \in A$$

indica che x è elemento di (appartiene ad) A , mentre con

$$x \notin A$$

indichiamo che x non è elemento di A .

Gli insiemi possono essere definiti elencandone gli elementi tra parentesi graffe. Ad esempio:

$$\{ \text{Ada, Pascal, Fortan, Lisp, Prolog, Coral} \}$$

è l'insieme costituito da sei elementi (ciascuno dei quali è il nome di un linguaggio di programmazione). Talvolta gli insiemi sono anche definiti mediante la seguente notazione:

$$\{ x \mid P(x) \}$$

che indica l'insieme costituito da tutti e soli gli oggetti x per cui vale la proprietà P .

Esempio:

$$\{ x \mid (x \bmod 5) = 0 \}$$

definisce l'insieme costituito da tutti e soli i multipli di 5, cioè da tutti e soli i numeri che divisi per 5 danno resto 0.

Sugli insiemi si definisce l'*uguaglianza* nel seguente modo:

$A = B$ se e solo se la seguente frase è vera:

“per ogni x , $x \in A$ se e solo se $x \in B$ ”

quindi: se $A = \{a,b\}$, $B = \{b,a\}$ e $C = \{b,b,a\}$ si avrà che $A=B=C$.

Notare che in un insieme non sono significative le occorrenze multiple di uno stesso elemento e l'ordine degli elementi è irrilevante.

Si definisce invece *multiinsieme* una collezione di elementi in cui occorrenze diverse di uno stesso elemento sono considerate distinte. Per cui se $A = \{a,a,b\}$, A è un multiinsieme costituito da tre elementi: due occorrenze di a e una di b . Il numero di occorrenze di un elemento a in un multiinsieme è detto la *molteplicità* di a .

Se $A = \{a,a,b\}$, $B = \{a,b\}$ e $C = \{a,b,a\}$ sono multiinsiemi, si avrà che $A = C$ e $A \neq B$.

Diciamo che un insieme A è *sottoinsieme* di un insieme B se la seguente frase è vera

“per ogni x , se $x \in A$ allora $x \in B$ ”

In tal caso scriviamo $A \subseteq B$, altrimenti scriviamo $A \not\subseteq B$, indicando così che esiste almeno un elemento $x \in A$ tale che $x \notin B$.

Se $A \subseteq B$ ed esiste $x \in B$ tale che $x \notin A$, allora A è detto *sottoinsieme proprio* di B e scriveremo $A \subset B$.

Denotiamo con \emptyset l'insieme vuoto, cioè l'insieme cui non appartiene alcun elemento. Naturalmente $\emptyset \subseteq A$ per qualunque insieme A . Infatti non esistono elementi di \emptyset che non siano anche elementi di A .

Se A e B sono due insiemi il *prodotto cartesiano* di A e B , denotato con $A \times B$ è l'insieme di tutte le coppie ordinate $\langle a, b \rangle$ tali che $a \in A$ e $b \in B$. Scriviamo pertanto:

$$A \times B = \{\langle a, b \rangle \mid a \in A, b \in B\}$$

Per esempio se $A = \{a,b\}$ e $B = \{1,2,3\}$

$$A \times B = \{\langle a, 1 \rangle, \langle a, 2 \rangle, \langle a, 3 \rangle, \langle b, 1 \rangle, \langle b, 2 \rangle, \langle b, 3 \rangle\}$$

Il prodotto cartesiano dei tre insiemi A , B e C è l'insieme delle terne ordinate di elementi di A , B e C :

$$A \times B \times C = \{ \langle a, b, c \rangle \mid a \in A, b \in B \text{ e } c \in C \}$$

In generale, dati n insiemi A_1, \dots, A_n il prodotto cartesiano tra essi è definito come segue:

$$A_1 \times A_2 \times \dots \times A_n = \{ \langle a_1, \dots, a_n \rangle \mid a_i \in A_i, i=1, \dots, n \}$$

gli elementi $\langle a_1, \dots, a_n \rangle$ sono detti n -uple.

Nel caso in cui $A=B$ il prodotto cartesiano $A \times B$, cioè $A \times A$, è indicato con A^2 . Similmente si indica con A^n il prodotto cartesiano di n insiemi tutti uguali ad A :

$$A^n = A \times \dots \times A = \{ \langle a_1, \dots, a_n \rangle \mid a_i \in A, i=1, \dots, n \}$$

n volte

Sugli insiemi definiamo l'operazione di *unione* e di *intersezione* come segue:

$$\text{unione: } A \cup B = \{x \mid x \in A \text{ oppure } x \in B\}$$

$$\text{intersezione: } A \cap B = \{x \mid x \in A \text{ ed } x \in B\}$$

Se $A \cap B = \emptyset$ diciamo che A e B sono *disgiunti*. Se A e B sono due insiemi, la loro differenza è definita come

$$A - B = \{x \mid x \in A \text{ e } x \notin B\}$$

Se A è un insieme, l'insieme di tutti i sottoinsiemi di A , detto *insieme potenza* di A , o *insieme delle parti* di A , e denotato con $P(A)$, è definito come segue:

$$P(A) = \{B \mid B \subseteq A\}$$

ad esempio, se $A = \{a, b\}$

$$P(A) = \{ \emptyset, \{a\}, \{b\}, \{a, b\} \}$$

Esempio 1. I numeri naturali sono un insieme, denotato con N

$$N = \{0, 1, 2, 3, \dots\}$$

I numeri interi sono un insieme, denotato con Z

$$Z = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

I numeri interi modulo n sono un insieme denotato con Z_n

$$Z_n = \{0, 1, 2, \dots, n-1\}$$

Diamo qui di seguito alcuni esempi di prodotti cartesiani e di insiemi potenza costruiti su N e Z_2 .

$$N^2 = \{<0,0>, <0,1>, \dots <1,0>, <1,1>, \dots, <2,0> \dots\}$$

$$N^3 = \{<0,0,0>, <0,0,1>, <0,0,2> \dots\}$$

$$P(N) = \{\emptyset, \{0\}, \{1\}, \dots \{0,1\}, \dots \{0,1,2\}, \dots\}$$

$$Z_2^2 = \{<0,0>, <0,1>, <1,0>, <1,1>\}$$

$$P(Z_2) = \{\emptyset, \{0\}, \{1\}, \{0,1\}\}$$

Esempio 2. Sia V un insieme finito non vuoto. Si pensi a V come un alfabeto di *caratteri* (o di lettere). Indichiamo con V^* l'insieme di tutte le sequenze finite costruite giustapponendo lettere di V . Per esempio se $V=\{a,b,c\}$

$$V^* = \{a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}$$

Gli elementi di V^* sono chiamati *parole* o *stringhe* su V .

Insiemi di questo tipo sono spesso usati in informatica. Per esempio se

$$T = \{\text{tutti i simboli che si trovano sulla tastiera di un terminale}\}$$

allora

$$T^* = \{\text{tutte le sequenze di simboli che si possono immettere in un calcolatore tramite quel terminale}\}.$$

Si possono definire sottoinsiemi particolarmente significativi di T^* , ad esempio:

$$\{w \in T^* \mid w \text{ è un programma Fortran}\},$$

oppure

$$\{w \in T^* \mid w \text{ è una espressione aritmetica Pascal}\}.$$

Esempio 3. Un insieme che spesso ricorre in informatica è l'insieme $\{T, F\}$ o anche *true, false* o anche *vero, falso*. Esso è detto l'insieme dei *valori di verità*.

1.2. Relazioni e funzioni

Una relazione R tra due insiemi A e B (o *relazione binaria*) è un sottoinsieme di $A \times B$.

$$R \subseteq A \times B$$

Per esempio se $A = \{\text{Carlo, Giulio}\}$ e $B = \{\text{Maria, Giulia, Ilaria}\}$, l'insieme

$$R = \{<\text{Carlo, Maria}>, <\text{Giulio, Ilaria}>\}$$

è un sottoinsieme di $A \times B$. Se esso mette in relazione gli elementi di A che sono

sposati con elementi di B, R è la relazione “esser sposato con”.

Vedremo nel seguito altri esempi di relazioni e di loro proprietà.

Una *funzione* da un insieme A ad un insieme B è una relazione f tra A e B tale che, per ogni $a \in A$ esiste un unico $b \in B$ tale che $\langle a, b \rangle \in f$.

Se per ogni $a \in A$ esiste al più un $b \in B$ tale che $\langle a, b \rangle \in f$, allora f è detta *funzione parziale* da A a B.

In entrambi i casi A è detto *dominio* di f e B è detto *codominio* di f.

Di solito si scrive $f: A \rightarrow B$ per indicare che f è una funzione da un insieme A ad un insieme B.

Se $\langle a, b \rangle \in f$ diciamo che b è il valore associato ad a da f o che b è il valore di f in a e scriviamo $f(a) = b$.

Per esempio, siano $A = \{a_1, a_2\}$ e $B = \{b_1, b_2, b_3\}$; se $f = \{\langle a_1, b_1 \rangle, \langle a_1, b_2 \rangle, \langle a_2, b_3 \rangle\}$, f è una relazione, ma non una funzione, perché ad a_1 vengono associati due elementi distinti di B. Se f invece è $f = \{\langle a_1, b_1 \rangle, \langle a_2, b_1 \rangle\}$, f è una funzione.

Se infine $f = \{\langle a_1, b_2 \rangle\}$, f è una funzione parziale da A a B, in quanto nessun valore di B è associato ad $a_2 \in A$.

Consideriamo ora alcune importanti proprietà di funzioni.

- Se $f: A \rightarrow B$ ha la proprietà che, ogni $b \in B$ è un valore possibile per f, allora f è detta *surgettiva*. Più formalmente, f è surgettiva se per ogni $b \in B$ esiste un $a \in A$ tale che $f(a) = b$.
- Se f applica elementi distinti di A in elementi distinti di B, f si dice *iniettiva*. Più formalmente, f è iniettiva, o uno-a-uno, se per ogni $a, a' \in A$ con $a \neq a'$ si ha che $f(a) \neq f(a')$. Equivalentemente possiamo dire che $f(a) = f(a')$ implica $a = a'$.
- Se f è surgettiva ed iniettiva, allora f è detta *biunivoca*.
- Una funzione $P: A \rightarrow \{T, F\}$ è detta un *predicato* su A. Per esempio, sia $P: N \rightarrow \{T, F\}$ così definita

$$P(n) = \begin{cases} T & \text{se } n \text{ è multiplo di 2} \\ F & \text{altrimenti} \end{cases}$$

P è il predicato “esser pari”.

Ogni predicato P su un insieme A definisce un sottoinsieme di A nel seguente modo:

$$\{x \mid x \in A \text{ e } P(x) = T\}$$

spesso abbreviato con

$$\{x \in A \mid P(x)\}$$

Similmente, data $f:A \rightarrow B$ essa definisce un sottoinsieme di B (detto *immagine* di A in B secondo f) nel seguente modo:

$$\{f(x) \mid x \in A\}$$

Due insiemi si dicono in *corrispondenza biunivoca* se esiste una funzione biunivoca tra essi. Se due insiemi finiti hanno la stessa cardinalità, essi sono in corrispondenza biunivoca. Se un insieme infinito è in corrispondenza biunivoca con l'insieme dei naturali, esso si dice costituito da una *infinità numerabile* di elementi.

Le funzioni fin qui viste sono funzioni di *un argomento*, o *monadiche*.

Se $f:A \times B \rightarrow C$ allora f è un *funzione a due argomenti*, il primo in A , il secondo in B e a valori in C . Tale f è una relazione tra $A \times B$ e C , cioè un sottoinsieme di $(A \times B) \times C$. Perciò un elemento di f è del tipo $\langle\langle a, b \rangle, c \rangle$ e, seguendo la notazione usata precedentemente, dovremmo scrivere $f(\langle a, b \rangle) = c$. Per brevità scriveremo $f(a, b) = c$.

Le stesse considerazioni si applicano a funzioni di n argomenti: scriveremo $f(a_1, \dots, a_n) = b$ quando $f:A_1 \times \dots \times A_n \rightarrow B$.

Agli interi sono associate alcune interessanti funzioni a due argomenti, per esempio

$$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \text{ (addizione)}$$

$$\cdot : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \text{ (moltiplicazione)}$$

All'insieme V^* delle stringhe su V è associata una funzione anch'essa interessante:

$$\cdot : V^* \times V^* \rightarrow V^*$$

dove $\cdot(v, w)$ è la parola o stringa ottenuta giustapponendo tutte le lettere di v e poi quelle di w . Questa operazione è detta *concatenazione* e spesso si scrive vw invece di $\cdot(v, w)$. È facile convincersi che $(vw)z = v(wz)$, cioè per la funzione \cdot sulle stringhe vale la proprietà associativa come per le funzioni $+$ e \cdot sugli interi. Diversamente da queste ultime si può notare che \cdot non è commutativa, infatti in genere $vw \neq wv$.

Come esempio dell'operazione di concatenazione, si consideri una istruzione in un linguaggio di programmazione: essa è una concatenazione di lettere.

L'addizione sugli interi è, come già detto una funzione

$$+ : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$$

quindi $+(3,2)=5$. Molto più spesso scriveremo però $3+2=5$. La prima notazione è detta *prefissa*, la seconda *infissa* ed è spesso usata nel caso di funzioni a due argomenti.

Per alcune funzioni * a due argomenti possono valere le seguenti proprietà:

a) *commutatività*: $* : A \times A \rightarrow A$ è commutativa se per ogni $a, b \in A$

$$(a * b) = (b * a)$$

b) *associatività*: $* : A \times A \times A \rightarrow A$ è associativa se per ogni $a, b, c \in A$

$$(a * b) * c = a * (b * c); \text{ in tal caso scriviamo anche } a * b * c.$$

c) *esistenza dell'identità*: $e \in A$ è detto un'identità o elemento unità per * se $a * e = e * a = a$ per ogni $a \in A$.

La addizione e la moltiplicazione sugli interi sono commutative e associative ed hanno lo 0 e l'1 rispettivamente come identità. La sottrazione non è né commutativa né associativa e non ha identità.

La concatenazione su V^* è associativa, ma non commutativa e non ha identità.

Se invece di considerare V^* consideriamo V^* con

$$V^* = \{\epsilon\} \cup V^+$$

dove ϵ denota la stringa vuota, avremo che su V^* è associativa, non commutativa ed ha la come identità, infatti $\epsilon v = v \epsilon$ per ogni $v \in V^*$. V^* è di solito detto universo linguistico sull'alfabeto V .

1.3. Funzione inversa, composizione di funzioni

L'*inversa* di una funzione $f: A \rightarrow B$ è l'insieme di coppie $\{(b, a) \mid (a, b) \in f\}$ essa è denotata con f^{-1} .

L'inversa di una funzione è quindi una relazione $f^{-1} \subseteq B \times A$. Essa è una funzione se e solo se f è biunivoca. Se f è iniettiva ma non surgettiva, f^{-1} è una funzione parziale.

Siano $f: A \rightarrow B$ e $g: B \rightarrow C$. Siccome il codominio di f coincide col dominio di g , ogni elemento di A può essere trasformato da f in un elemento di B , il quale è poi trasformato da g in un elemento di C . Possiamo cioè associare ad f e g una funzione $h: A \rightarrow C$ così definita:

$$h(a) = g(f(a))$$

Definiamo *composizione (a sinistra)* di una funzione $f: A \rightarrow B$ la funzione $g \circ f: A \rightarrow C$ che trasforma $a \in A$ in $g(f(a)) \in C$. Una funzione $g: B \rightarrow C$ è detta componibile a sinistra con una funzione $f: A \rightarrow B$ se definendo $f(A) = \{y \in B \mid y \in f(x)\}$,

$x \in A\}$, l'immagine di A sotto f , si ha che $f(A) \subseteq B$.

Sulla composizione di funzioni si possono dimostrare interessanti proprietà:

- la composizione di due funzioni è una funzione
- siano f e g come sopra. Se $(g \cdot f)^{-1}$ è una funzione (eventualmente parziale) su tutto A allora f e g sono iniettive e

$$(g \cdot f)^{-1} = f^{-1} \cdot g^{-1}$$

- la composizione di funzioni è associativa, cioè data $f:A \rightarrow B$, $g:B \rightarrow C$ e $h:C \rightarrow D$ si ha che

$$(h \cdot g) \cdot f = h \cdot (g \cdot f)$$

1.4. Cardinalità di un insieme

Dato un insieme A il numero di elementi che lo costituiscono rappresenta la sua *cardinalità*, che viene denotata con $|A|$.

La cardinalità di un insieme può essere finita o infinita.

Dati due insiemi A e B si ha che

- $|A| \leq |B|$ se esiste una funzione iniettiva $f:A \rightarrow B$
- $|A| = |B|$ se esiste una corrispondenza biunivoca $f:A \rightarrow B$.

Un insieme A è *enumerabile* se può essere messo in corrispondenza biunivoca con l'insieme dei numeri naturali $N = \{0, 1, 2, \dots\}$.

Se la cardinalità di un insieme A è finita, allora essa è sempre maggiore della cardinalità di un suo sottoinsieme proprio.

Si noti che questo non è sempre vero nel caso di insiemi infiniti come mostra il seguente esempio.

Esempio. Consideriamo l'insieme $N = \{0, 1, 2, \dots\}$ dei numeri naturali e l'insieme $P = \{0, 2, 4, \dots\}$ dei numeri naturali divisibili per due.

È facile vedere che la seguente funzione $f:A \rightarrow B$, $f(i)=2i$ stabilisce una corrispondenza biunivoca tra N e P ; quindi P è un sottoinsieme di N con la stessa cardinalità di N .

Un fondamentale risultato di Cantor permette di stabilire l'esistenza di insiemi con cardinalità maggiore della cardinalità di N .

Teorema. Non esiste una corrispondenza biunivoca tra N e $P(N)$, l'insieme delle parti di N . Quindi $P(N)$ ha cardinalità strettamente maggiore di N .

2. ELEMENTI DI ALGEBRA

2.1. Strutture algebriche

Una struttura algebrica, detta anche *tipo di dato matematico* o *tipo di dato astratto* è definita come:

$$\langle D, f_1, \dots f_n, P_1, \dots P_m, \dots x, y, \dots \rangle$$

in cui

- (i) D è un insieme (dominio)
- (ii) f_i è una funzione a n_i argomenti in D e a valori in D
- (iii) P_i è un predicato a n_i argomenti in D
- (iv) x e y sono elementi distinti di D .

Esempi

$\langle Z, \leq \rangle$ è un ordinamento totale su Z

$\langle Z, = \rangle$ è un'equivalenza su Z

$\langle R, +, *, \leq, 0, 1 \rangle$ è la struttura dei reali ordinata,

con due operazioni $+$ e $*$ commutative, associative, aventi 0 e 1 rispettivamente come identità.

Nota: la definizione di ordinamento totale ed equivalenza verranno fornite nel par. 3.1 di questa appendice.

Vedremo nel seguito altre strutture algebriche di interesse in informatica.

2.2. Omomorfismi

In molti casi due diverse strutture algebriche hanno delle caratteristiche comuni. Se una struttura ha come dominio D_1 e l'altra ha come dominio D_2 possiamo mostrare la similitudine tra loro mediante una funzione $f: D_1 \rightarrow D_2$ che preserva alcune caratteristiche della struttura.

Definizione. Siano $\langle D_1, * \rangle$ e $\langle D_2, o \rangle$ due strutture algebriche dove $*$ è una funzione binaria infissa su D_1 e, analogamente, o è una funzione binaria infissa su D_2 . Una funzione $f: D_1 \rightarrow D_2$ è detta omomorfismo da $\langle D_1, * \rangle$ a $\langle D_2, o \rangle$ se per ogni $x_1, x_2 \in D_1$ si ha che

$$f(x_1) o f(x_2) = f(x_1 * x_2)$$

Graficamente un omomorfismo può essere visualizzato come:

$$\begin{array}{ccc} D_1 & \times & D_1 \\ f \downarrow & f \downarrow & f \downarrow \\ D_2 & \times & D_2 \end{array} \xrightarrow{*} \begin{array}{c} D_1 \\ \xrightarrow{o} D_2 \end{array}$$

f è detto *epimorfismo* se è una funzione surgettiva, *monomorfismo* se è una funzione iniettiva, *isomorfismo* se è una funzione biunivoca, *endomorfismo* se $D_1 = D_2$, *automorfismo* se è un endomorfismo biunivoco.

2.3. Operazioni su Z_n

Come già visto in precedenza, indichiamo con Z_n l'insieme dei numeri interi $\{0, 1, \dots, n-1\}$, cioè l'insieme in cui sono stati "resi identici" tutti gli interi che, divisi per n , hanno uguale resto.

Quindi

$$Z_n = \{x \bmod n \mid x \in Z\}$$

Per esempio se $n=3$

$$Z_3 = \{x \bmod 3 \mid x \in Z\} = \{0, 1, 2\}$$

infatti

$$0 \bmod 3 = 3 \bmod 3 = 6 \bmod 3 = \dots = 0$$

$$1 \bmod 3 = 4 \bmod 3 = 7 \bmod 3 = \dots = 1$$

$$2 \bmod 3 = 5 \bmod 3 = 8 \bmod 3 = \dots = 2$$

Siano $a, b, c, d, x, y, n \in Z$ con $n \geq 2$ allora avremo che:

1. se $a \bmod n = b \bmod n$ allora $(a-b) \bmod n = 0$
2. se $a \bmod n = b \bmod n$ e $b \bmod n = c \bmod n$ allora $a \bmod n = c \bmod n$
3. se $a \bmod n = b \bmod n$ e $c \bmod n = d \bmod n$ allora

$$(ax + cy) \bmod n = (bx + dy) \bmod n$$

4. se $a \bmod n = b \bmod n$ e $c \bmod n = d \bmod n$ allora $ac \bmod n = bd \bmod n$

Per ogni $n \geq 2$ l'*addizione modulo n*, indicata con $+_n$ è una funzione (binaria infissa) su Z_n che trasforma $x_1, x_2 \in Z_n$ in $x_3 \in Z_n$ dove $x_3 = (x_1 + x_2) \bmod n$. Si può dimostrare che $+_n$ è commutativa e associativa.

A titolo di esempio mostriamo la tabella che definisce $+_3$.

$+_3$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

La *moltiplicazione modulo n*, denotata con $*_n$ è una funzione (binaria infissa) su Z_n che trasforma $x_1, x_2 \in Z_n$ in $x_3 \in Z_n$ dove $x_3 = (x_1 * x_2) \text{ mod } n$. Per il caso $n=3$ la moltiplicazione è definita dalla seguente tabella

$*_3$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Anche per $*_n$ valgono le proprietà commutativa ed associativa.

Come esempio di omomorfismo mostriamo che la funzione $f: Z_4 \rightarrow Z_2$ definita come

$$\begin{aligned} f(0) &= f(2) = 0 \\ f(1) &= f(3) = 1 \end{aligned}$$

cioè la funzione f che vale 0 sui pari e 1 sui dispari è un omomorfismo di $\langle Z_4, +_4 \rangle$ in $\langle Z_2, +_2 \rangle$. Riportiamo le tabelle che definiscono $+_4$ e $+_2$:

$+_4$	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$+_2$	0	1
0	0	1
1	1	0

Per mostrare che f è un omomorfismo basta riscrivere $+_4$ "identificando" i numeri pari P e i dispari D

$+_4$	P	D
P	P	D
D	D	P

Come si può constatare $+_4$ scritta per P e D è identica a $+_2$, se si identifica P con 0 e D con 1.

3. GRAFI E ALBERI

3.1. Relazioni e grafi

Ricordando che una relazione tra due insiemi A e B è un sottoinsieme R di $A \times B$ possiamo dare una descrizione di R in forma diagrammatica scrivendo tutti gli elementi di A, tutti gli elementi di B e congiungendo gli elementi di A mediante una freccia agli elementi di B ad essi associati dalla relazione R. Questa descrizione di R viene chiamata rappresentazione in forma di *grafo bipartito*. Un esempio di grafo bipartito è mostrato in fig. 1.

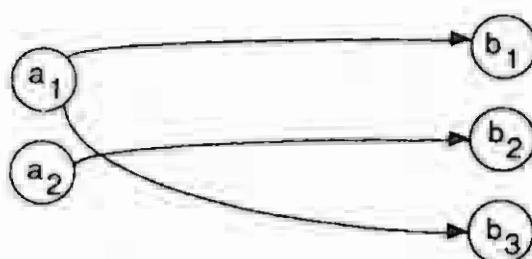


Fig. 1 - Grafo bipartito su $A=\{a_1, a_2\}$ e $B=\{b_1, b_2, b_3\}$

Se $A=B$, invece di rappresentare $R \subseteq A \times A$ in forma di grafo bipartito possiamo rappresentarlo in forma di *grafo orientato*, scrivendo gli elementi di A una sola volta e scrivendo una freccia da a_i ad a_j per quelle coppie $\langle a_i, a_j \rangle \in R$. Gli elementi $a \in A$ sono detti *nodi* o *vertici* del grafo, le frecce sono anche dette *archi orientati*. Alcuni esempi di grafo sono mostrati in fig. 2.

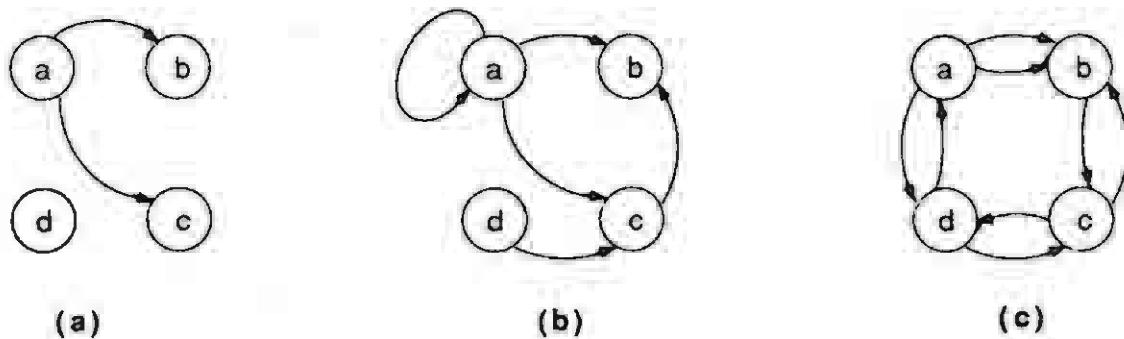


Fig. 2 - Esempi di grafi orientati su $A=\{a,b,c,d\}$

Possiamo dare una definizione di grafo orientato in termini insiemistici come segue:

Un *grafo orientato* G è una coppia $\langle A, R \rangle$ dove A è un insieme finito non vuoto detto insieme dei nodi di G e $R \subseteq A \times A$, detto insieme degli *archi orientati* di G , è un insieme di coppie ordinate. Se $\langle a_i, a_j \rangle \in R$ allora diremo che nel grafo G c'è un arco da a_i ad a_j , cioè un arco che ha come *nodo iniziale* a_i e come *nodo finale* a_j .

Se in un grafo orientato G esiste un arco da a_i ad a_j diremo che a_i è un *predecessore* di a_j nel grafo G e che a_j è un *successore* di a_i nel grafo G .

Se a è un nodo in un grafo orientato G chiameremo *grado d'ingresso* di a in G il numero di archi di G che hanno a come nodo finale, chiameremo invece *grado di uscita* di a in G il numero di archi di G che hanno a come nodo iniziale.

Nella fig. 2a il grado di ingresso del nodo a è 0, il grado di uscita è 2, il grado di ingresso del nodo b è 1, il grado d'uscita è 1, eccetera.

In altre parole, il grado di ingresso di un nodo a in un grafo orientato G è il numero dei suoi predecessori in G , il grado di uscita è il numero dei suoi successori. Un nodo con un grado d'uscita nullo è detto *pozzo*, un nodo con grado di ingresso nullo è detto *sorgente*. Un nodo con un grado di ingresso e d'uscita entrambi nulli è detto *isolato*.

Un grafo orientato $G = \langle A, R \rangle$ è detto *completo* se per ogni coppia di nodi $a_i, a_j \in A$ esiste un arco che va da a_i ad a_j in G . In altre parole G è completo se e solo se $R = A \times A$.

I grafi vengono spesso usati in informatica per rappresentare relazioni. Vediamo ora alcune proprietà delle relazioni e come queste si riflettono nella loro rappresentazione in forma di grafo.

Una relazione $R \subseteq A \times A$ è detta *riflessiva* se per ogni $a \in A$, $\langle a, a \rangle \in R$; un grafo orientato $G = \langle A, R \rangle$ è detto *riflessivo* se ogni nodo ha un *cappio*, cioè un arco orientato che ha quel nodo come nodo iniziale e finale.

Una relazione R è detta *simmetrica* se ogni volta che la coppia $\langle a_i, a_j \rangle \in R$

allora la coppia $\langle a_i, a_j \rangle \in R$. Un grafo orientato $G = \langle A, R \rangle$ è detto *simmetrico* se tra due nodi a_i e a_j non ci sono archi o ce ne sono due orientati in senso opposto.

Se una relazione è simmetrica, oltre a rappresentarla con forma di grafo orientato possiamo anche rappresentarla in forma di *grafo non orientato*.

Intuitivamente possiamo dire che, se la relazione è simmetrica, possiamo "fondere" ogni coppia di archi orientati in senso opposto tra due nodi in un solo arco non orientato.

In fig. 3 sono mostrati alcuni esempi di grafi (non orientati).

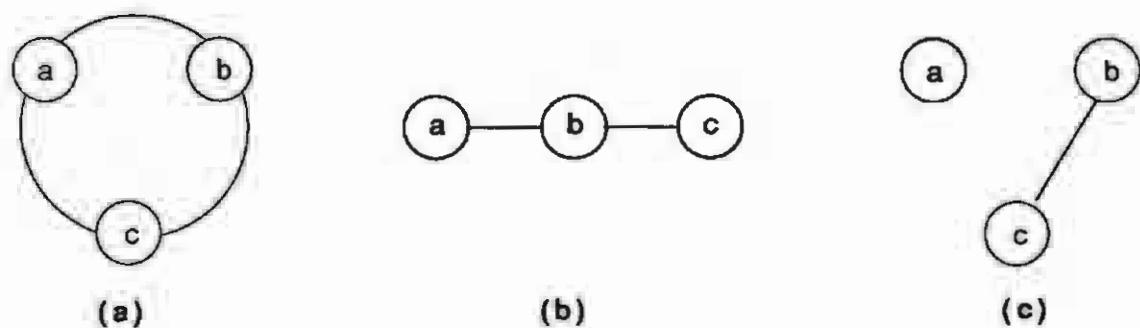


Fig. 3 - Esempi di grafi non orientati sull'insieme $A = \{a, b, c\}$

- Una relazione R è detta transitiva se ognialvolta $\langle a_i, a_j \rangle \in R$ e $\langle a_j, a_k \rangle \in R$ allora $\langle a_i, a_k \rangle \in R$. Nella terminologia dei grafi orientati possiamo dire che un grafo orientato è *transitivo* se ognialvolta c'è una arco dal nodo a_i al nodo a_j e c'è un arco dal nodo a_j al nodo a_k , c'è un arco che va dal nodo a_i al nodo a_k .

- Se una relazione R è riflessiva, simmetrica e transitiva, essa si chiama *relazione di equivalenza*. Un grafo associato ad una relazione di equivalenza è mostrato in fig. 4.

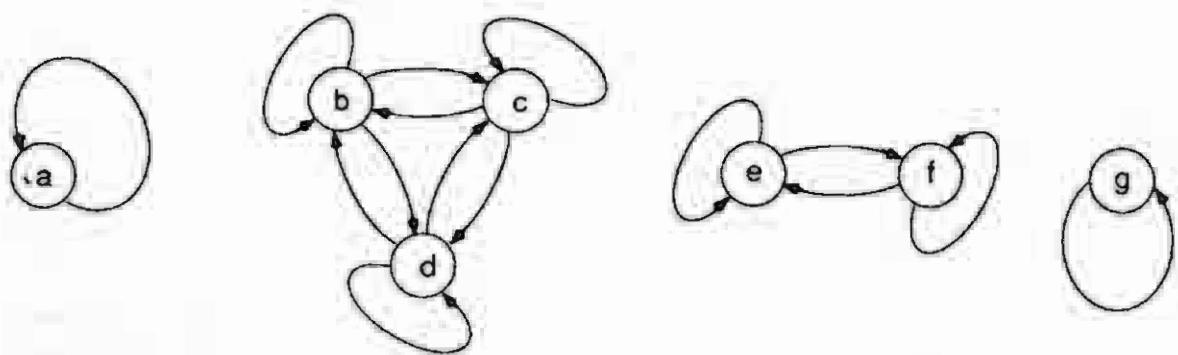


Fig. 4 - Esempio di grafo associato ad una relazione di equivalenza su $A = \{a, b, c, d, e, f, g\}$

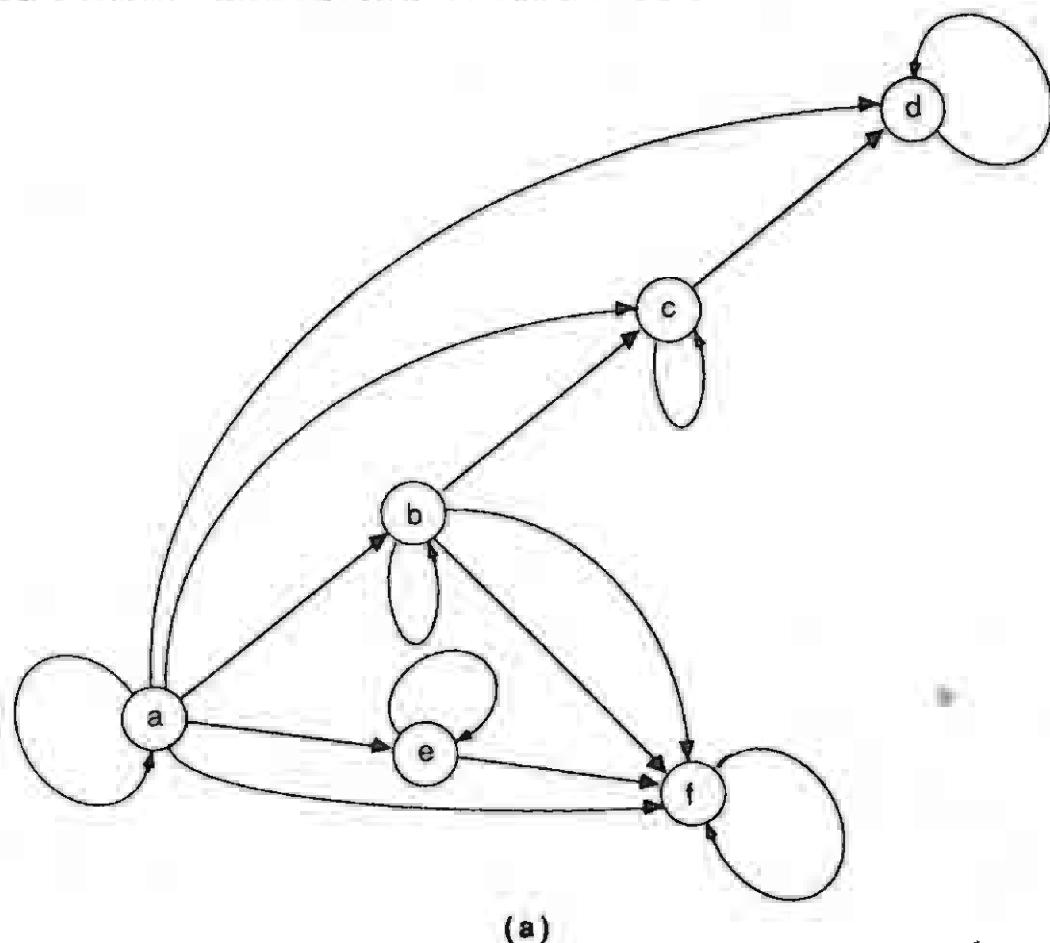
Dalla figura si nota come i nodi sono partizionati in "isole". In ogni isola ci sono archi che congiungono due coppie qualsiasi di nodi, cioè ogni isola è un grafo completo, ma tra le varie isole non ci sono archi. Queste isole corrispon-

dono alle classi di equivalenza della relazione, cioè a quei sottoinsiemi della relazione costituiti da tutti gli elementi equivalenti fra di loro.

- Una relazione R è detta antisimmetrica (quindi $G = \langle A, R \rangle$ è detto *antisimmetrico*), dati due nodi $a_i, a_j \in A$ se $\langle a_i, a_j \rangle \in R$, allora $\langle a_j, a_i \rangle \notin R$.
- Una relazione R (e il relativo grafo) è detta *asimmetrica* se non è né simmetrica né antisimmetrica.
- Una relazione R è detta irriflessiva se per ogni $a \in A$, $\langle a, a \rangle \notin R$. Quindi un grafo è *irriflessivo* se per nessuno dei suoi nodi esiste un cappio.
- Una relazione che sia riflessiva, antisimmetrica e transitiva è detta *ordinamento parziale*, così è detto il grafo che la rappresenta.
- Se in un ordinamento parziale $R \subseteq A \times A$ si ha che per ogni $a_i, a_j \in A$ o $\langle a_i, a_j \rangle \in R$ o $\langle a_j, a_i \rangle \in R$ (cioè tutti gli elementi di A sono “confrontabili in R”) allora R si dice “ordinamento totale”.

Molto spesso nel disegnare un grafo associato ad un ordinamento (parziale o totale) si omettono tutti gli archi che sono implicati dalla riflessività (cioè tutti i cappi) e quelli implicati dalla transitività, per cui la visualizzazione grafica ne risulta notevolmente semplificata.

Un esempio è mostrato in fig. 5. La fig. 6 è un ordinamento totale; un grafo con questa forma viene spesso detto una *catena*.



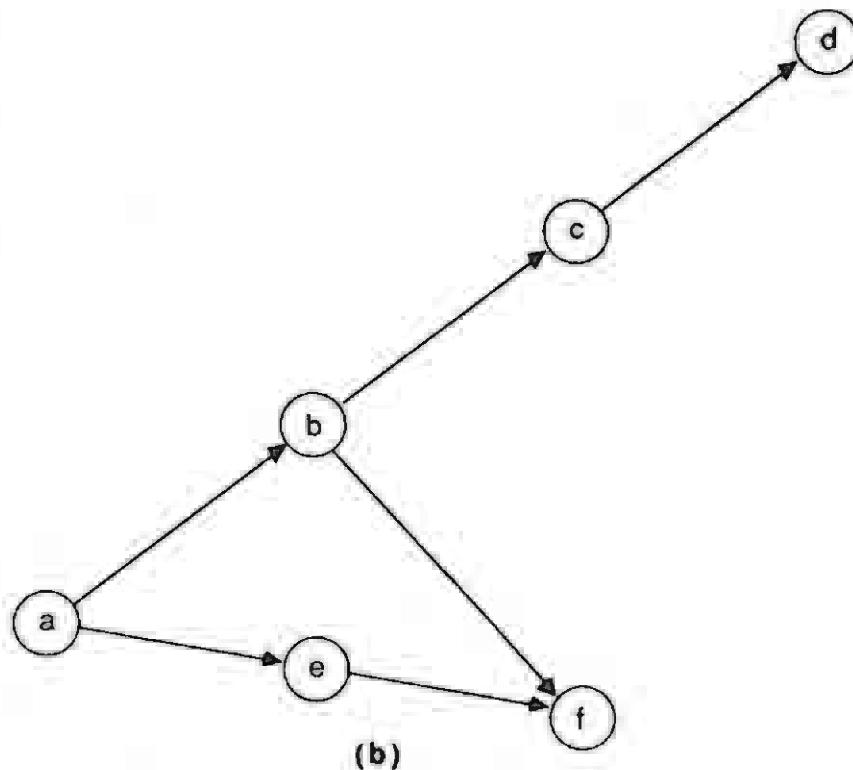


Fig. 5 - Grafo (a) e grafo semplificato (b) rappresentanti un ordinamento parziale

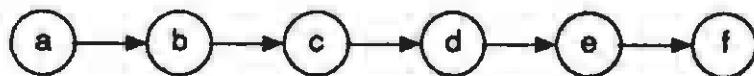


Fig. 6 - Grafo semplificato rappresentante un ordinamento totale

In un grafo $G = \langle A, R \rangle$ dati due nodi $a, b \in A$ dicesi *cammino* da a a b una sequenza di nodi, elementi di A :

tali che

$$a_0, a_1, \dots, a_n$$

$$(1) n \geq 1$$

$$(2) a = a_0, a_n = b$$

$$(3) \text{ per ogni } i, 0 \leq i \leq n, \langle a_i, a_{i+1} \rangle \in R.$$

Il numero n è detto *lunghezza* del cammino. Se i nodi che compaiono nel cammino sono distinti tra loro, il *cammino* è detto *semplice*.

Se $a = b$ chiamiamo il cammino un *ciclo*; se tutti gli elementi che compaiono nel ciclo sono distinti, il *ciclo* è detto *semplice*. Un cappio è quindi un ciclo di lunghezza 1. Un grafo senza cicli è detto *aciclico*.

Per esempio, nella fig. 2b intorno al nodo a c'è un cappio, la sequenza a, c, b è un cammino da a a b di lunghezza 2, nella fig. 2.c la sequenza a, b, c, d, a è un ciclo.

Si può dimostrare (vedere l'esempio del par. 4) che un grafo è transitivo se e solo se ogni volta che esiste un cammino che va da un nodo a ad un nodo b, allora esiste un arco che va da a a b.

In un grafo $G = \langle A, R \rangle$ dicesi *semicammino* da un nodo a ad un nodo b una sequenza di nodi di A

$$a_0, a_1, \dots, a_n$$

tali che

- (1) $n \geq 1$
- (2) $a_0 = a, a_n = b$
- (3) per ogni $i, 0 \leq i \leq n, \langle a_i, a_{i+1} \rangle \in R$ oppure $\langle a_{i+1}, a_i \rangle \in R$

n è detto lunghezza del semicammino. Se i nodi che compaiono in un *semicammino* sono distinti, esso è detto *semplice*.

Notare che mentre un cammino da a a b non è un cammino da b ad a, un semicammino da a a b è anche un semicammino da b ad a. Per esempio nella fig. 2b la sequenza a, c, d è un semicammino da a a d, la sequenza a, c, b, a è un semiciclo.

Definiremo *grafo connesso* un grafo $G = \langle A, R \rangle$ in cui, dati $a, b \in A$ esiste un cammino da a a b o un cammino da b ad a. G è detto *fortemente connesso* se dati due nodi $a, b \in A$ esiste un cammino da a a b. G è detto *debolmente connesso* se dati due nodi $a \neq b \in A$ esiste un semicammino tra di essi.

Se $G = \langle A, R \rangle$ è un grafo e V_1, V_2 sono due alfabeti (di solito disgiunti), la terza $\langle G, f_1, f_2 \rangle$, dove $f_1: A \rightarrow V_1$ e $f_2: R \rightarrow V_2$ è detto grafo etichettato su V_1 e V_2 . Gli elementi di V_1 sono le etichette dei nodi, quelli di V_2 sono le etichette degli archi.

Esempi di grafi etichettati sono mostrati in fig. 7.

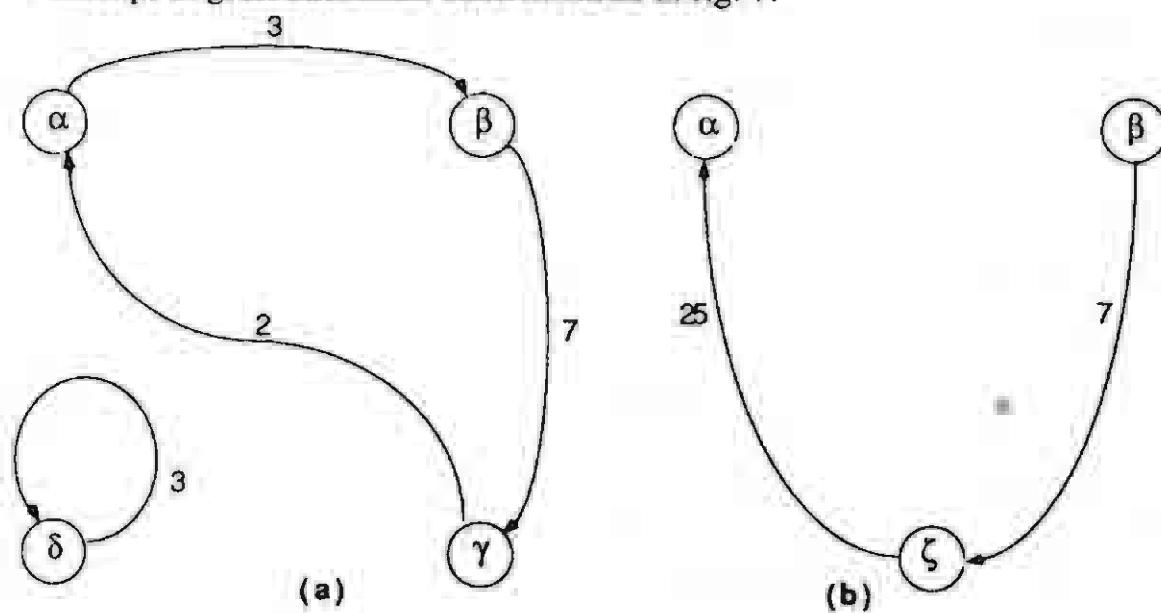


Fig. 7 - Esempi di grafi etichettati su $V_1 = \{\alpha, \beta, \gamma, \delta, \dots\}$ e $V_2 = \mathbb{N}$

Se due nodi di un grafo sono collegati da più di un arco (nello stesso verso) si parla di *multigrafo*.

Un esempio di multigrafo è mostrato in fig. 8.

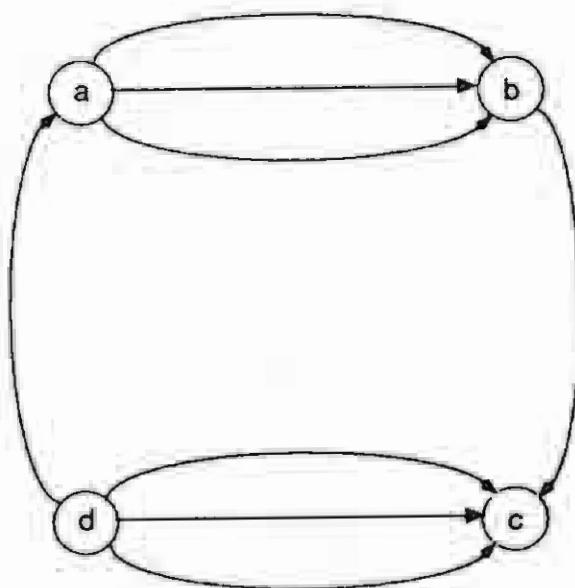


Fig. 8 - Multigrafo su $A=\{a,b,c,d\}$

Un multigrafo può essere quindi definito come una coppia $I = \langle A, M \rangle$ dove A è un insieme di nodi e M è un multiinsieme di archi, cioè un multiinsieme di coppie $\langle a_1, a_2 \rangle \in A \times A$.

Un multigrafo in cui il massimo numero di archi che collegano la stessa coppia di nodi è k può anche essere definito come una $k+1$ -upla $I = \langle A, R_1, \dots, R_k \rangle$ dove A è un multinsieme e R_1, \dots, R_k sono k relazioni.

Anche nel caso di multigrafi si possono introdurre etichette e si può quindi parlare di multigrafi etichettati.

3.2. Alberi

Un particolare tipo di grafi orientati ricorre molto spesso in informatica: gli alberi.

Un *albero* (finito) è un grafo orientato aciclico in cui esiste un nodo, detto *radice*, con grado di ingresso 0, ogni altro nodo ha grado di ingresso 1.

Esempi di alberi sono mostrati nella fig. 9.

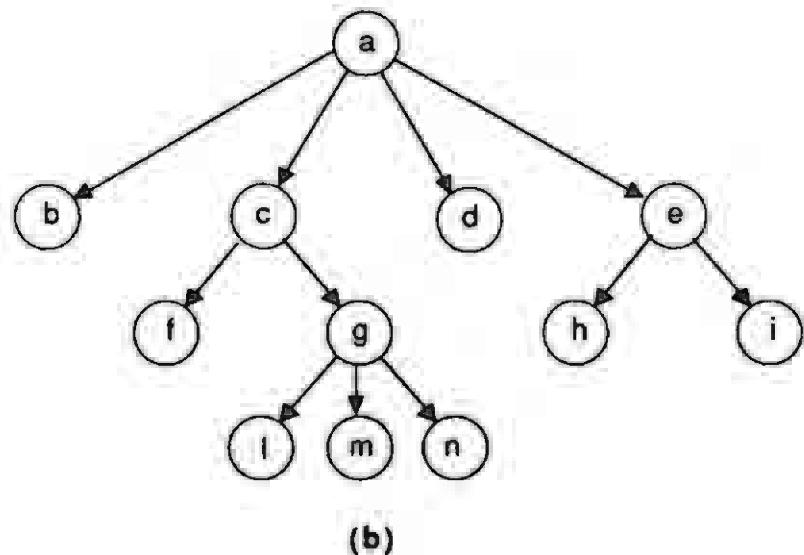
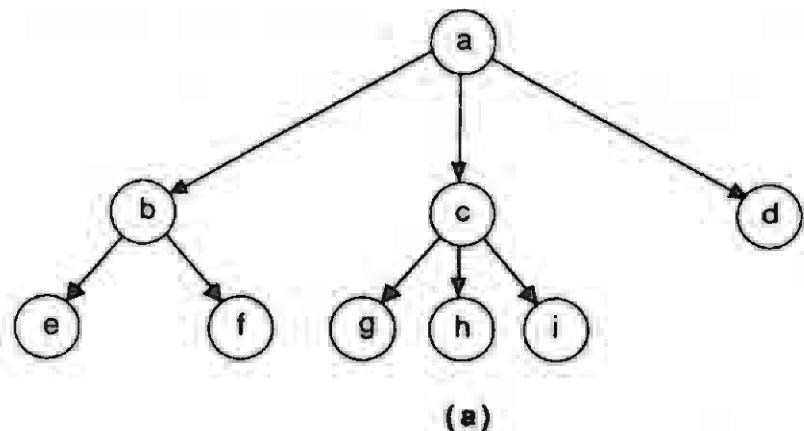


Fig. 9- Esempi di alberi

I nodi di un albero hanno grado di uscita qualunque, ma finito. Quei nodi che hanno grado di uscita 0 vengono detti *foglie*. Gli alberi come qui definiti vengono talvolta detti *alberi sorgente*, in quanto la radice è un nodo sorgente. Si può parlare di *albero pozzo* se si cambia la definizione data sopra sostituendo “grado di uscita” al posto di “grado di ingresso”. Un *albero simmetrico* è un grafo simmetrico, senza cicli e connesso. In un albero simmetrico qualunque nodo può essere considerato nodo radice.

Per gli alberi valgono alcune importanti proprietà, che ci limitiamo ad enunciare.

1. Un albero con n nodi ha esattamente $n-1$ archi.
2. Un albero è un grafo debolmente connesso.
3. In un albero esiste esattamente un cammino che va dalla radice a qualunque altro nodo.
4. Un albero è un ordinamento parziale (in cui sono stati tralasciati gli archi implicati dalla riflessività e dalla transitività).

Nella letteratura esistono molte definizioni alternative di alberi. Ne diamo qui di seguito una che risulta particolarmente utile in quanto mette in luce la natura induttiva della costruzione della "struttura" albero.

Un albero T è un grafo in cui:

1. esiste un nodo r con n ($n \geq 0$) nodi successori a_1, \dots, a_n ; r è detto radice dell'albero;
2. tutti i nodi di T tranne r possono essere ripartiti in n insiemi disgiunti, ciascuno dei quali individua in T un albero T_1, \dots, T_n con radice a_1, \dots, a_n , rispettivamente.

T_1, \dots, T_n sono detti *sottoalberi immediati* della radice r di T . Il numero di sottoalberi immediati di un nodo costituisce il grado di quel nodo. Le foglie hanno grado 0. Per analogia con gli alberi genealogici, le radici dei sottoalberi T_1, \dots, T_n di un nodo a sono detti *figli* di a , e *fratelli* tra loro. Il nodo a è detto il loro *padre*.

In un albero T si definisce la nozione di *livello di un nodo* nel seguente modo:

1. la radice ha livello 0;
2. se un nodo di T ha livello k , tutti i suoi figli hanno livello $k+1$.

Si definisce *profondità* di un albero il massimo livello dei suoi nodi.

Nell'albero di fig. 9b il nodo b ha livello 1, il nodo m ha livello 3. Ancora nella fig. 9, l'albero (a) ha profondità 2, l'albero (b) ha profondità 3, l'albero (c) ha profondità 0.

Negli alberi definiti finora non è stato dato alcun ordinamento tra i figli dei vari nodi, per cui gli alberi in fig. 10 sono due diverse visualizzazioni grafiche dello stesso albero.

Se si impone un ordine tra i figli di ogni nodo di un albero si parla di *albero ordinato*. Nella visualizzazione grafica degli alberi ordinati si conviene di partire da sinistra e si scrivono ordinatamente i figli verso destra.

Particolari esempi di alberi ordinati sono gli alberi binari.

Un *albero binario* è un albero ordinato in cui ogni nodo ha al massimo due figli detti *figlio sinistro* e *figlio destro*, rispettivamente.

La fig. 11 mostra esempi di alberi binari. Notare che quelli mostrati in (b) e (c) sono diversi tra loro.

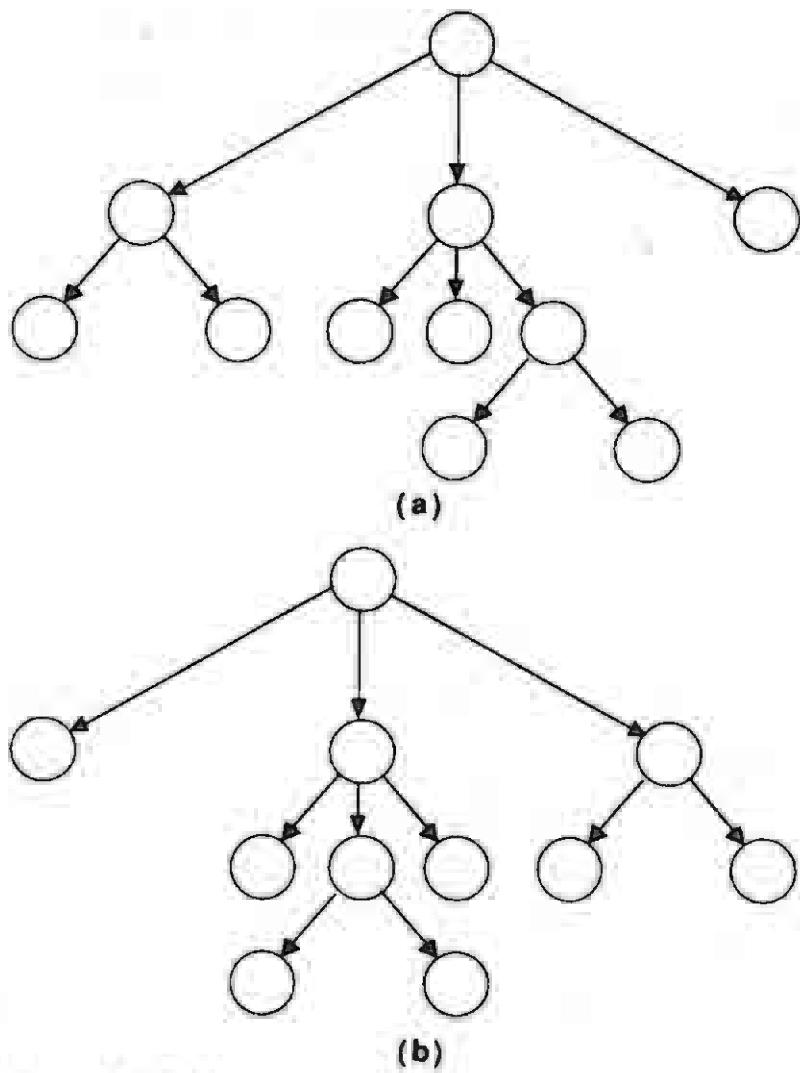


Fig. 10 - Alberi uguali tra di loro

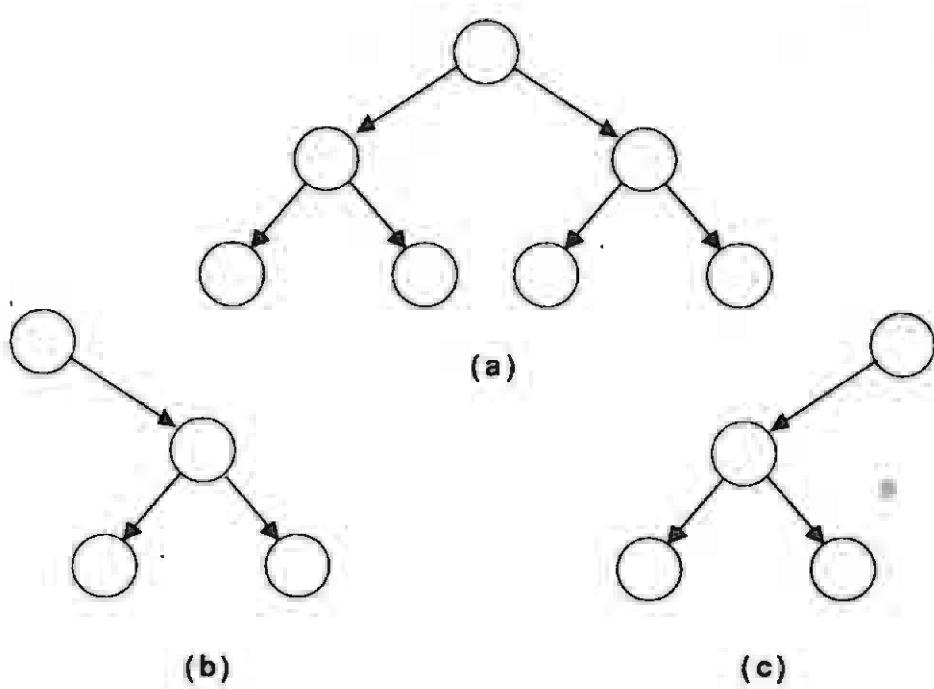


Fig. 11 - Alberi binari

Un albero binario è detto *completo* se ogni nodo che non sia una foglia ha esattamente due figli e tutte le foglie hanno lo stesso livello. Un albero binario completo di profondità k avrà quindi 2^k foglie.

4. INDUZIONE MATEMATICA

In matematica si fanno spesso costruzioni del tipo “immagina di avere x_k , ora costruisci x_{k+1} ”. Il principio che giustifica le costruzioni di questo tipo è il principio della *induzione matematica*. Esso viene ricordato qui di seguito in quanto trova molte applicazioni in informatica e in particolare in programmazione.

Principio di induzione. Sia P un predicato su N . Se

1. $P(0)$ è vero e
2. per ogni k , se $P(k)$ è vero allora anche $P(k+1)$ è vero, allora $P(n)$ è vero per ogni n .

Cerchiamo di convincerci della “ragionevolezza” di un tale principio. Supponiamo che 1. e 2. valgano.

1. dice che $P(0)$ è vero
2. dice che, per $k=0$, essendo vero $P(0)$ posso concludere che è vero $P(1)$
2. dice che, per $k=1$, essendo vero $P(1)$ posso concludere che è vero $P(2)$ e ancora
2. dice che, per $k=2$, essendo vero $P(2)$ posso concludere $P(3)$, e così via.

Quindi, basandoci sul principio di induzione, se vogliamo dimostrare che un predicato P vale per tutti gli n , possiamo limitarci a dimostrare che

- 1) $P(0)$ è vero (*Passo base* dell’induzione)
- 2) se $P(k)$ è supposto vero allora anche $P(k+1)$ è vero (*Passo induttivo*).

$P(k)$ si chiama l’ipotesi induttiva.

- A titolo di esempio riportiamo due dimostrazioni costruite usando il principio di induzione.

Esempio 1. Per tutti gli $n \in N$ vale la proprietà che chiameremo P

$$P(n) = 0 + 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

Dimostrazione

Passo base: $0 = 0(0+1)$ (P è verificato al passo 0)

Passo induttivo: (P(k) implica P(k+1))

assumiamo P(k) vero, allora

$$P(k+1) = 0+1+\dots+k+(k+1) = (0+\dots+k)+(k+1) = P(k)+(k+1)$$

ipotesi induttiva

$$= \frac{k(k+1)}{2} + (k+1)$$

$$= \frac{k(k+1)}{2} + \frac{2(k+1)}{2}$$

$$= \frac{(k+1)(k+2)}{2}$$

quindi $P(k+1)$ è vero.

Esempio 2. Sia $G=<A,R>$ un grafo transitivo.

Se esiste un cammino tra a e b , nodi di A , allora esiste un arco $(a,b) \in R$.

La dimostrazione viene fatta per induzione sulla lunghezza k del cammino tra a e b (ricordare che per definizione $k \geq 1$, quindi il passo base dell'induzione verrà fatto per $k=1$).

Passo base. Se $k=1$ il cammino tra a e b è un arco tra a e b , quindi il passo base è verificato.

Passo induttivo. Supponiamo che l'ipotesi induttiva sia vera per k .

Prendiamo in G un cammino di lunghezza $k+1$: $\langle a_0, \dots, a_k, a_{k+1} \rangle$.

Allora $\langle a_0, a_k \rangle$ è un cammino da a_0 ad a_k di lunghezza k .

Per l'ipotesi induttiva $\langle a_0, a_k \rangle$ è un arco di R . Per definizione di cammino $\langle a_0, a_k+1 \rangle \in R$. Quindi per transitività $\langle a_0, a_{k+1} \rangle \in R$. Questo conclude la dimostrazione.

4.1. Definizione induttiva di insiemi

Spesso gli insiemi di interesse nella programmazione possono essere definiti come unione di insiemi

$$S = \bigcup_{i \in I} S_i$$

dove $I \subseteq N$ è un insieme di indici e S_i sono insiemi.

Esempio 1. Sia V un insieme finito non vuoto di caratteri detto alfabeto. Possiamo esprimere l'insieme di tutte le stringhe o parole di lunghezza qualsiasi $n \geq 0$ formate dai simboli di V nel seguente modo

$$V^* = \bigcup_{n \in N} V_n$$

dove

$$V_0 = \{\epsilon\} \text{ l'insieme che contiene solo la stringa vuota}$$

$$V_1 = V_0 \cup V \text{ l'insieme delle stringhe di lunghezza al più 1.}$$

$$V_n = \{v \cdot w \mid v \in V, w \in V_{n-1}\} \text{ l'insieme delle stringhe di lunghezza al più n.}$$

Esempio 2. Costruiamo ora in maniera analoga alla precedente l'insieme di tutte le espressioni aritmetiche che contengono variabili, costanti e operazioni di addizione e che sono esprimibili in un linguaggio di programmazione L.

Sia $A_0 = \{a \mid a \text{ è un simbolo legale in } L \text{ per denotare una variabile o una costante}\}$

A_0 è l'insieme delle espressioni che consistono di un solo simbolo, questi simboli saranno, in generale, identificatori di variabili e costanti numeriche.

Possiamo definire A_1 come l'insieme delle espressioni che contengono al più un simbolo di addizione.

$$A_1 = A_0 \cup \{a_1 \oplus a_2 \mid a_1, a_2 \in A_0\}$$

notare che abbiamo scritto \oplus e non $+$ per indicare che \oplus è un simbolo dell'alfabeto di L e *non* la funzione di addizione sugli interi.

Possiamo continuare la costruzione precedente

$$A_n = A_{n-1} \cup \{e_1 \oplus e_2 \mid e_1, e_2 \in A_{n-1}\}$$

ed esprimere l'insieme di tutte le espressioni aritmetiche contenenti un numero qualunque di simboli di variabili, costante e di addizione come:

$$A = \bigcup_{n \in N} A_n$$

Esempio 3. Sia A un insieme di oggetti qualunque detti atomi. Sia *cons* (dall'inglese "construct") un simbolo che useremo per denotare l'operazione di costruzione di liste, cioè sequenze ordinate di lunghezza qualunque di elementi di A. L, l'insieme delle liste di lunghezza n qualunque, $n \geq 0$, di elementi di A, può essere costruito come segue:

$$L_0 = \{nil\} \text{ dove } nil \text{ è un simbolo speciale che denota la lista vuota}$$

$$L_1 = L_0 \cup \{cons(a, nil) \mid a \in A\}$$

$$L_2 = L_1 \cup \{cons(a, l) \mid a \in A, l \in L_1\}$$

$$L_n = L_{n-1} \cup \{cons(a, l) \mid a \in A, l \in L_{n-1}\}$$

La costruzione che stiamo facendo va "letta" così: una lista di lunghezza al massimo n è un oggetto *cons* (a, l) con una *testa* che è un elemento $a \in A$ e una *coda* l che è una lista lunga al massimo $n-1$. Quindi l'insieme di tutte le liste di lunghezza finita su A risulta essere

$$L = \bigcup_{n \in \mathbb{N}} L_n$$

Quello che abbiamo fatto nei tre esempi precedenti può essere così descritto: abbiamo fissato un insieme B , base della costruzione, a partire dal quale, mediante l'uso ripetuto di una funzione f (costruttore) abbiamo costruito un insieme S . Tale insieme gode delle due seguenti importanti proprietà:

- a) S è chiuso sotto f , cioè se applichiamo f ad un elemento di S otteniamo elementi di S ;
- b) S è il minimo insieme chiuso sotto f che contiene B .

Questo fatto ci è garantito dal seguente teorema:

Teorema fondamentale sulla induzione:

Sia U un insieme, tale che $B \subseteq U$ e $f: U \rightarrow U$

se

$$A_0 = B$$

$$A_{k+1} = A_k \cup \{f(x) \mid x \in A_k\}$$

$$A = \bigcup_k A_k$$

allora

- (1) A è chiuso sotto f
- (2) se $U \supseteq A' \supseteq B$ e A' è chiuso sotto f allora $A' \supseteq A$.

Dimostrazione

(1) A è chiuso sotto f . Se $x \in A$ allora esiste un k per cui $x \in A_k$. Quindi $f(x) \in A_{k+1}$ e quindi $f(x) \in A$

(2) A è il minimo.

Sia A' tale che $U \supseteq A' \supseteq B$ e chiuso sotto f .

Per dimostrare che $A' \supseteq A$ dimostriamo che per ogni k , $A' \supseteq A_k$.

Ne seguirà che $A = \bigcup_k A_k \subseteq A'$.

Lo dimostriamo per induzione su K .

Passo base $A' \supseteq A_0$ per ipotesi

Passo induttivo. Supponiamo $A' \supseteq A_k$ dobbiamo dimostrare che $A' \supseteq A_{k+1}$, cioè che se $x \in A_{k+1}$, allora $x \in A'$. Se $x \in A_{k+1}$ si hanno due casi

- a) $x \in A_k$, ma allora $x \in A'$ per l'ipotesi induttiva;
- b) $x \in \{f(z) \mid z \in A_k\}$, ma se $z \in A_k$ per l'ipotesi induttiva $z \in A'$ ed essendo A' chiuso sotto f si ha che $f(z) \in A'$ quindi $x \in A'$.

Un disegno che visualizza questa costruzione è riportato in figura 12.

Sulla base di questo teorema possiamo introdurre un nuovo schema di definizione di insiemi.

Schema di definizione (induttivo)

Sia U un insieme, $B \subseteq U$, $f: U \rightarrow U$. La seguente regola di costruzione

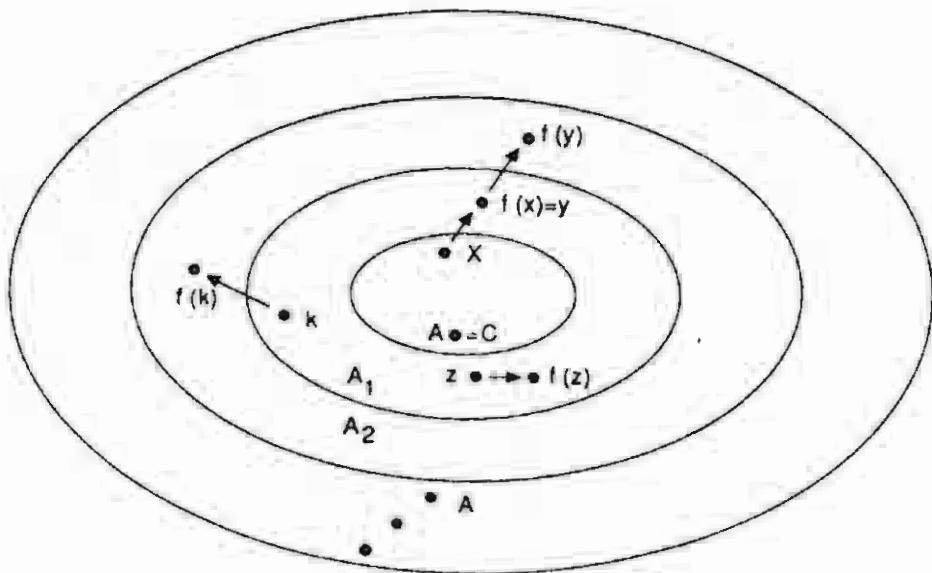


Fig. 12 · Costruzione induttiva di insiemi

1. $A \supseteq B$
2. se $x \in A$ allora $f(x) \in A$
3. nient'altro appartiene ad A

definisce A come il più piccolo insieme contenente B e chiuso sotto f , cioè

$$A = \bigcup_{k \in \mathbb{N}} A_k$$

Nota: in effetti gli insiemi costruiti precedentemente possono essere definiti usando uno schema leggermente più generale:

nell'esempio 1 f non è unaria, ma binaria

$$f = . \quad : V \times V \rightarrow V^*$$

nell'esempio 2 f è binaria

$$f = \oplus \quad \oplus : A \times A \rightarrow A$$

nell'esempio 3 f è binaria

$$f = cons \quad cons : A \times L \rightarrow L$$

Anche la definizione di alberi data nel par. 3.2 ricalca lo schema induttivo qui presentato: l'insieme A degli alberi (di profondità qualunque purché finita) può essere definito come segue. Se B è un insieme (di nodi):

1. ogni elemento di B è un elemento di A : (cioè ogni nodo è un albero che consiste della sola radice);
2. se a_1, \dots, a_n sono elementi di A allora l'oggetto che ha un elemento r di B come radice e a_1, \dots, a_n come (sotto) alberi (immediati) è un albero;
3. nient'altro è un albero.

In questo caso esprimere la funzione di costruzione f è piuttosto intricato, ci limitiamo a farlo nel caso di alberi binari

$$f = albin$$

$$albin : B \times A^0 \times A^0 \rightarrow A^0$$

dove $A^0 = A \cup \{\epsilon\}$ ed ϵ rappresenta l'albero binario vuoto.

5. LOGICA MATEMATICA

5.1. Il calcolo proposizionale

In questi ultimi due paragrafi daremo alcune nozioni di logica matematica. La logica studia i *sistemi formali*. Essi servono a dare l'intelaiatura deduttiva rigorosa entro cui derivare asserzioni su una qualche realtà. Il primo e più semplice sistema formale che presentiamo è il *calcolo proposizionale*. Esso formalizza ragionamenti basati su *proposizioni* che sono vere o false (*logica a due valori di verità*) e studia i modi in cui proposizioni possono essere combinate a costituire una nuova proposizione e come la verità o falsità di quest'ultima dipenda dalla verità o falsità delle proposizioni costituenti.

La combinazione di proposizioni avviene mediante l'uso di *operatori* detti *connettivi logici*; essi sono:

la negazione	indicata con \neg
la congiunzione	indicata con \wedge
la disgiunzione	indicata con \vee
la implicazione	indicata con \rightarrow

Il significato dei connettivi logici corrisponde a quello attribuito loro dall'uso quotidiano:

- negazione "non": $\neg A$ è vera se A è falsa e viceversa;
- congiunzione "e": $A \wedge B$ è vera se A e B sono contemporaneamente vere;
- disgiunzione (semplice) "oppure", (cioè vel latino): $A \vee B$ è vera se almeno uno tra A e B sono veri;
- implicazione (materiale) "se allora": $A \rightarrow B$ è vera se, essendo vera A anche B è vera, essendo invece A falsa B può essere sia vera che falsa.

Si noti come talvolta vengano usati anche altri connettivi, per esempio la disgiunzione esclusiva, che corrisponde all'aut latino (indicata con \vee : $A \vee B$ è vera se almeno una tra A e B son vere, ma non entrambe contemporaneamente), o l'equivalenza logica (indicata con \equiv : $A \equiv B$ è vera se A e B sono o contemporaneamente vere o contemporaneamente false). Nel seguito non tratteremo i connettivi \vee e \equiv , in quanto essi possono esser facilmente definiti in termini degli altri quattro già introdotti sopra.

I connettivi logici $\wedge, \vee, \rightarrow$ sono connettivi binari perché agiscono su due proposizioni; il connettivo \neg è invece *unario*.

Per definire il calcolo proposizionale dobbiamo dapprima definire il linguaggio, cioè l'insieme delle *formule proposizionali* e poi vedremo nel seguito come a queste formule viene attribuito un significato. L'insieme delle formule proposizionali viene qui definito mediante una costruzione induttiva, ricalcando lo schema di definizione mostrato nel par. 4.1.

Sia $E = \{A, B, C, \dots, P, Q, R, \dots\}$ un insieme finito non vuoto (*alfabeto*) di simboli che chiamiamo *simboli di proposizione*. L'insieme F delle *formule ben formate* (o più brevemente, *formule*, o *f.b.f.*) del calcolo proposizionale è così definito:

1. se $A \in E$ allora $A \in F$
- 2a. se $A \in F$ allora $\neg A \in F$
- 2b. se $A \in F$ e $B \in F$ e b è un connettivo binario allora $(A \ b \ B) \in F$
3. nient'altro appartiene a F .

Notare la differenza tra l'uso dei simboli A e B, \dots e dei simboli a e b, \dots nella precedente definizione: A e B, \dots sono simboli proposizionali, cioè elementi di E , mentre a , b, \dots sono variabili che stanno ad indicare generici elementi di E e F .

Diamo ora alcune definizioni: il *connettivo principale* e le *sottoformule immediate* di una f.b.f. si definiscono nel seguente modo:

1. le f.b.f. di E *non* hanno sottoformule immediate, né connettivo principale, esse sono perciò dette *atomiche*;
2. $\neg A$ ha come unica sottoformula immediata A e come connettivo principale \neg ;
3. $A \ b \ B$ (dove b è un qualsiasi connettivo binario) hanno due sottoformule immediate A e B (rispettivamente sinistra e destra) e b come connettivo principale.

Descriviamo ora una maniera di dare un significato alle formule di F , definiamo cioè una funzione $V: F \rightarrow \{T, F\}$ che per ogni formula di F ci dica se essa è vera o falsa.

A questo scopo iniziamo col definire il comportamento dei connettivi logici, come illustrato informalmente all'inizio di questo paragrafo. Faremo questo attraverso le cosiddette *tabelle dei valori di verità*

Connettivo \neg

A	$\neg A$
T	F
F	T

La tabella dei valori di verità del connettivo unario dice che se $A \in F$ è vera allora $\neg A$ è falsa, e viceversa.

<i>Connettivo</i> \wedge	<i>A</i>	<i>B</i>	<i>A \wedge B</i>
	T	T	T
	T	F	F
	F	T	F
	F	F	F

<i>Connettivo</i> \vee	<i>A</i>	<i>B</i>	<i>A \vee B</i>
	T	T	T
	T	F	T
	F	T	T
	F	F	F

<i>Connettivo</i> \rightarrow	<i>A</i>	<i>B</i>	<i>A \rightarrow B</i>
	T	T	T
	T	F	F
	F	T	T
	F	F	T

Notare che la tabella dei valori di verità per il connettivo \rightarrow fa sì che quando la premessa dell'implicazione è falsa allora tutta l'implicazione è vera, indipendentemente dal valore di verità della conseguenza dell'implicazione. Per cui, in base a questa definizione, la frase "se il numero 3 è pari allora l'Italia è una monarchia" risulta essere un'implicazione vera.

Di fatto la definizione appena data per \rightarrow rende la formula $A \rightarrow B$ equivalente (nel senso che ha la stessa tabella di verità) alla formula $(\neg A) \vee B$. (Il lettore lo verifichi costruendo la relativa tabella dei valori di verità).

Le tabelle dei valori di verità sono delle funzioni, da F (caso del \neg) o da $F \times F$ (caso dei connettivi binari) a valori in {T,F}. Chiamiamo H queste funzioni.

Definiamo interpretazione una funzione da E in {T,F}

$$I : E \rightarrow \{T, F\}$$

Definiamo valutazione booleana di f la funzione

$$V : F \rightarrow \{T, F\}$$

così definita

$$V(A) = I(A) \quad \text{se } A \in E$$

$$V(\neg A) = H_{\neg}(V(A)) \quad \text{se } A \in F$$

$$V(A \mathbf{b} B) = H_b(V(A), V(B)) \quad \text{se } A, B \in F \text{ e } b \text{ è connettivo binario}$$

Notare che una volta fissate le funzioni H, la funzione I determina unicamente la funzione V.

Diremo che una formula A del calcolo proposizionale è *vera secondo V* (o secondo I) se $V(A)=T$, che essa è *falsa secondo V* (o secondo I) se $V(A)=F$.

Diremo che A è *soddisfacibile* se esiste V (cioè I) tale che $V(A)$ è vera; diremo che A è una *tautologia* (o formula *valida*) se $V(A)$ è vera per ogni V. Diremo che A è *contraddittoria* se nessuna V rende vera A.

Data una formula di F il suo valore di verità si può calcolare mediante V, eventualmente visualizzando il calcolo mediante una tabella di verità. Ad esempio:

se $A = (A \wedge B) \vee \neg(C \wedge A)$

V di	A	B	C	$(A \wedge B)$	$(C \wedge A)$	$\neg(C \wedge A)$	$(A \wedge B) \vee \neg(C \wedge A)$
T	T	T	T	T	T	F	T
T	T	F	T	F	F	T	T
T	F	T	F	F	T	F	F
T	F	F	F	F	F	T	T
F	T	T	F	F	F	T	T
F	T	F	F	F	F	T	T
F	F	T	F	F	F	T	T
F	F	F	F	F	F	T	T

Un modo alternativo di disegnare la tabella dei valori di verità è scrivendo l'interpretazione dei simboli di proposizione sotto ai medesimi e il valore di verità della proposizione composta sotto il relativo connettivo (questo metodo è detto "tabella dei valori di verità ridotta"). Il valore di verità dell'intera formula si troverà nella colonna corrispondente al suo connettivo principale.

Esempio:

(A)	\wedge	(B)	\vee	\neg	(C)	\wedge	(A)
T	T	T	T	F	T	T	T
T	T	T	T	T	F	F	T
T	F	F	F	F	T	T	T
T	F	F	T	T	F	F	T
F	F	T	T	T	T	F	F
F	F	T	T	T	T	F	F
F	F	F	T	T	T	F	F
F	F	F	T	T	F	F	F

La formula in esempio è soddisfacibile in quanto esistono sette interpretazioni che la rendono vera. Non è una tautologia perché essa non è vera per ogni interpretazione. La formula $A \vee \neg A$ è un esempio di contraddizione.

Per valutare una f.b.f. possiamo anche basarci sulla sua struttura e sulla interpretazione dei simboli di proposizione che in essa compaiono. Più precisamente possiamo:

1. dare la f.b.f. A e l'interpretazione I per i simboli di proposizione che in essa compaiono;
2. costruire l'albero strutturale per A ;
3. sostituire ai nodi terminali (foglie) dell'albero i valori di verità dati da I ;
4. far "scorrere" lungo i nodi dell'albero i valori di verità desunti mediante le H , fino ad arrivare alla radice dell'albero. Tale valore di verità è quello associato alla f.b.f. A .

Per le formule del calcolo proposizionale valgono le seguenti importanti proprietà, che il lettore potrà verificare:

per ogni f.b.f. A, B e C :

$$\begin{array}{ll} \neg\neg A & \text{equivale ad } A \text{ (legge della doppia negazione)} \\ \neg(A \wedge B) & \text{equivale a } \neg A \vee \neg B \\ \neg(A \vee B) & \text{equivale a } \neg A \wedge \neg B \\ A \wedge (B \vee C) & \text{equivale a } (A \wedge B) \vee (A \wedge C) \text{ (distributività di } \wedge \text{ rispetto a } \vee) \\ A \vee (B \wedge C) & \text{equivale a } (A \vee B) \wedge (A \vee C) \text{ (distributività di } \vee \text{ rispetto a } \wedge) \end{array} \quad \left. \begin{array}{l} \neg(A \wedge B) \text{ e } \neg(A \vee B) \\ \neg A \vee \neg B \end{array} \right\} \text{ (leggi di De Morgan)}$$

5.2. Il calcolo dei predicati

Il *calcolo dei predicati* o *logica del primo ordine* è un linguaggio formale con un potere espressivo maggiore rispetto al calcolo proposizionale. Infatti, nel calcolo proposizionale non potremmo esprimere frasi del tipo "esiste un numero infinito di numeri primi" oppure "tutti i multipli di 4 e 5 sono multipli di 4 e multipli di 5".

Queste due frasi possono essere espresse con formule del calcolo dei predicati come segue:

$$\forall x \ y. (x < y \wedge P(y))$$

dove P è il predicato "esser primo" (esso può essere espresso dalla formula:

$$1 < x \wedge \forall y. \forall z. [(y * z) = x \rightarrow (y = 1 \vee z = 1)]$$

e

$$\forall x. [x \bmod (4 * 5) = 0] \rightarrow [x \bmod 4 = 0 \wedge x \bmod 5 = 0]$$

Il calcolo dei predicati tratta formule del tipo visto sopra, formule cioè in cui compaiono simboli di costanti, variabili e funzioni, oltre ai simboli di predicato e permette di quantificare esistenzialmente e universalmente (\exists , \forall) simboli di variabili. Questo è il motivo per cui si chiama logica del primo ordine, se si potessero quantificare anche simboli di funzioni e predicato si parlerebbe di logica del secondo ordine.

Passiamo ora alla definizione del linguaggio del calcolo dei predicati. Faremo questo in maniera induttiva introducendo prima l'alfabeto, poi l'insieme dei *termini* e infine l'insieme delle formule ben formate.

L'alfabeto A del calcolo dei predicati è costituito dalla unione dei seguenti insiemi:

- a. un'infinità numerabile di simboli di variabile;
- b. un insieme finito di simboli di costante;
- c. un insieme finito di simboli di funzione a ciascuno dei quali è associato un numero che denota la sua arità (cioè il numero dei suoi argomenti);
- d. un insieme finito non vuoto di simboli di predicato a ciascuno dei quali è associato un numero che denota la sua arità (cioè il numero dei suoi argomenti);
- e. l'insieme dei simboli $\{\wedge, \vee, \neg, \rightarrow, \forall, \exists, (,), , , .\}$.

Notare che gli insiemi descritti in b. e c. possono essere vuoti, gli altri no.

Definiamo ora l'insieme T dei termini su A , in maniera induttiva:

1. se x è un simbolo di variabile o di costante allora x è un termine;
2. se t_1, \dots, t_n sono termini e f è un simbolo di funzione n -ario allora $f(t_1, \dots, t_n)$ è un termine;
3. nient'altro è un termine.

Una volta costruito l'insieme dei termini definiamo l'insieme delle formule ben formate del calcolo dei predicati, anche questo in modo induttivo:

1. se t_1, \dots, t_n sono termini e P è un simbolo di predicato n -ario, allora $P(t_1, \dots, t_n)$ è una f.b.f. (detta atomica);
2. se A, B sono f.b.f. e x è un simbolo di variabile allora
 - 2.a) $\neg A$ è una f.b.f.,
 - 2.b) $A \vee B$ è una f.b.f.,
 - 2.c) $A \wedge B$ è una f.b.f.,
 - 2.d) $A \rightarrow B$ è una f.b.f.,
 - 2.e) $\forall x.A$ è una f.b.f.,
 - 2.f) $\exists x.A$ è una f.b.f.;
3. nient'altro è una f.b.f.

Anche per le f.b.f. del calcolo dei predicati si può costruire un albero strutturale, analogamente alle f.b.f. del calcolo proposizionale.

Nel calcolo proposizionale il significato viene associato alle formule dalle funzioni di verità, mediante una interpretazione di simboli proposizionali nell'insieme {T,F}. Nel calcolo dei predicati, l'associazione di un significato alle formule è più complessa: dobbiamo specificare un tipo di dato astratto (o struttura algebrica, come definito nel par. 2 di questa appendice) cioè un dominio di valori, un insieme di funzioni e un insieme di predicati.

Una *interpretazione* per le formule del calcolo dei predicati consiste quindi in una struttura $S = \langle S, f_1, \dots, f_n, P_1, \dots, P_m \rangle$ e delle seguenti funzioni di denotazione $\sigma_c, \sigma_f, \sigma_p$:

1. una funzione σ_c che ad ogni simbolo di costante di A, alfabeto di F associa un valore di dominio S;
2. una funzione σ_f che ad ogni simbolo di funzione (n-aria) di A associa una funzione n-aria di S;
3. una funzione σ_p che ad ogni simbolo di predicato n-ario di A associa un predicato n-ario di S.

Vediamo ora come un'interpretazione determina un valore in S per ciascun termine e un valore di verità per ciascuna formula. Sia τ una funzione che ad ogni simbolo di variabile di A associa un valore di S; la valutazione dei termini, cioè la funzione σ che associa ad ogni termine un valore in S viene definita in modo ricorsivo come segue:

1. se il termine è un simbolo di costante c , $\sigma(c) = \sigma_c(c)$;
2. se il termine è un simbolo di variabile x , $\sigma(x) = \tau(x)$;
3. se il termine è $f(t_1, \dots, t_n)$ allora

$$\sigma(f(t_1, \dots, t_n)) = \sigma_f(f)(\sigma(t_1), \dots, \sigma(t_n))$$

Possiamo ora definire il predicato di uguaglianza sui termini:

$$t_1 = t_2 \text{ se e solo se } \sigma(t_1) = \sigma(t_2)$$

Passiamo ora a definire la valutazione booleana delle formule del calcolo dei predicati, ancora una volta faremo questo definendo una funzione ricorsiva che ricalca la struttura delle definizioni delle formule stesse.

La funzione V che associa valori di verità a f.b.f. nel calcolo dei predicati può essere definita come:

1. $V(P(t_1, \dots, t_n)) = \sigma_p(P)(\sigma(t_1), \dots, \sigma(t_n))$
2. a) $V(\neg A) = H_{\neg}(V(A))$, cioè

- $V(\neg A)$ è vero se $V(A)$ è falso, e viceversa
- $V(A \vee B) = H_{\vee}(V(A), V(B))$
 - $V(A \wedge C) = H_{\wedge}(V(A), V(C))$
 - $V(A \rightarrow B) = H_{\rightarrow}(V(A), V(B))$
 - $V(\forall x.A) = T$ se $V_{s/x}(A) = T$ per ogni $s \in S$
 - $V(\exists x.A) = T$ se $V_{s/x}(A) = T$ per almeno un $s \in S$

notare che col simbolo $V_{s/x}$ è stata indicata la valutazione che si ottiene da V modificando τ , funzione di valutazione delle variabili, in modo che $\tau(x)=s$. Quindi, la riga e) della definizione dice che $\forall x.A$ è vera se A è vera, qualunque sia il valore s che si attribuisce alla sua variabile x ; la riga f) dice invece che la formula $\exists x.A$ è vera se esiste un valore s che, attribuito a x rende vera A .

Notare come la funzione di valutazione V dipende da S , σ_c , σ_r , σ_p , e dal particolare τ .

Per le formule del calcolo dei predicati si possono dare le seguenti definizioni:

- una interpretazione $\langle S, \sigma_c, \sigma_r, \sigma_p \rangle$ soddisfa una formula A se esiste un τ tale che $V(A) = T$.
- una formula A è soddisfacibile se esiste una interpretazione $\langle S, \sigma_c, \sigma_r, \sigma_p \rangle$ e un τ tale che $V(A) = T$.
- una formula è insoddisfacibile se per nessun $\langle S, \sigma_c, \sigma_r, \sigma_p \rangle$ e τ essa è soddisfatta.
- una formula A è valida se ogni $\langle S, \sigma_c, \sigma_r, \sigma_p \rangle$ e τ la soddisfano.

Notare che A è valida se (e solo se) $\neg A$ è non soddisfacibile.

Verificare la validità di una formula ben formata richiederebbe di verificarne la verità in ogni possibile interpretazione. Per evitare ciò si introduce la nozione di *regola di inferenza*, *derivazione* e *teorema*.

Una *regola di inferenza* è una relazione tra f.b.f. che si scrive di solito come segue

$$\frac{A_1 \dots A_n}{B}$$

$A_1 \dots A_n$ sono dette le *premesse*, B la *conseguenza*. In una regola di inferenza se $A_1 \dots A_n$ sono f.b.f. valide, B è una f.b.f. valida.

Data una f.b.f. A del calcolo dei predicati una derivazione per A è una sequenza di f.b.f. ognuna delle quali o è una f.b.f. valida o è ottenuta da alcune delle precedenti mediante una regola di inferenza.

Dicesi un *teorema* una f.b.f. per la quale esiste una derivazione.

Un *apparato deduttivo* è un insieme di regole di inferenza. Un apparato deduttivo si dice *completo* per il calcolo dei predicati se l'insieme dei teoremi derivabili con esso coincide con l'insieme delle f.b.f. valide.

Per la formulazione di apparati deduttivi completi per il calcolo dei predicati si rimanda ai libri di logica.

Concludiamo questa appendice dando soltanto la definizione di *teoria* e *modello*.

Dicesi *teoria* (del primo ordine) un insieme di f.b.f. detti *assiomi* e l'insieme di tutte le f.b.f. (*teoremi*) che da questi possono derivare usando un apparato deduttivo completo per il calcolo dei predicati. Dicesi *modello* per una teoria del primo ordine una struttura algebrica in cui tutti gli assiomi sono veri (e di conseguenza tutti i teoremi).



FONDAMENTI DI PROGRAMMAZIONE DEI CALCOLATORI ELETTRONICI

Questo libro affronta lo studio dei concetti fondamentali relativi alla programmazione dei calcolatori elettronici. Il testo presenta un insieme di argomenti riguardanti i linguaggi programmativi, le strutture di dati, le metodologie di programmazione, con una impostazione che integra in modo opportuno fondamenti teorici e metodologie di base.

Il libro è diviso in quattro parti.

Nella prima parte vengono presentate le nozioni teoriche sui linguaggi di programmazione e le tecniche di implementazione, con particolare riguardo ai linguaggi imperativi.

Nella seconda parte vengono introdotte le problematiche connesse alla rappresentazione dei dati: partendo dalla definizione astratta dei diversi tipi di dato, si illustrano e si confrontano varie tecniche per la loro rappresentazione.

La terza parte è dedicata alle metodologie di progetto e di analisi dei programmi. In particolare, vengono presentati i metodi per la costruzione di programmi e vengono illustrate le tecniche utilizzate per dimostrare la correttezza dei programmi e per valutarne l'efficienza.

Nella quarta parte, vengono presentate le caratteristiche fondamentali di un sistema di calcolo: vengono descritti i diversi componenti dell'architettura di un elaboratore elettronico e i diversi componenti del software di base. Vengono inoltre presentate le caratteristiche principali di alcuni linguaggi di programmazione.

Concepito come testo universitario, il volume contiene gli elementi di base per essere compreso anche da altri lettori, a cui fornisce un agile ed aggiornato compendio su diversi argomenti riguardanti la programmazione.

ISBN 88-204-3819-4



L. 40.000, iva inclusa

9 788820 438197