Università degli Studi di Napoli Federico II
Dipartimento di Informatica e Sistemisitica

Giulio Iannello

Dispense di

Fondamenti di Informatica

Dicembre 2003

Indice

Elenco delle figure

Note introduttive

Queste dispense sono un tentativo di raccogliere in un documento unitario e coerente il materiale didattico messo a punto per un corso di base di informatica adatto ad un Corso di Laurea a livello universitario di natura tecnico-scientifica.

Nella loro forma attuale le dispense presentano ancora parti mancanti o lacunose. Tuttavia è stato fatto uno sforzo per rendere omogenea la presentazione dei vari argomenti ed è pertanto opportuno segnalare al lettore i principali accorgimenti impiegati per agevolare la loro assimilazione.

1. I termini impiegati per identificare i concetti via via introdotti sono sempre indicati in corsivo quando compaiono per la prima volta. In alcuni casi tali termini sono riportati in corsivo anche nelle occorrenze successive quando i concetti loro associati giocano un ruolo importante nel contesto di cui si tratta.

   Si raccomanda pertanto al lettore di porre particolare cura nell'assimilare i concetti associati a tutti i termini riportati in corsivo.

2. Il testo è frazionato in una parte principale, in osservazioni e in esempi. Le osservazioni possono essere omesse in una prima lettura del testo, mentre gli esempi possono essere omessi nelle fasi di ripasso finali e sostituiti con lo svolgimento di esercizi. Le osservazioni contribuiscono peraltro in modo sostanziale alla presentazione dei vari argomenti e approfondiscono aspetti importanti o comunque utili all'assimilazione dei concetti.

3. Anche le figure hanno un ruolo rilevante nella comprensione dei concetti tipici delle discipline informatiche, Si raccomanda pertanto al lettore di fare particolare attenzione alle figure che illustrano caratteristiche strutturali dei vari concetti presentati o che cercano di rendere in forma grafica quelli tra essi che risultano più astratti.

4. Si raccomanda infine di svolgere gli esercizi riportati alla fine dei paragrafi a breve distanza dalla lettura del testo. Essi, grazie anche alla loro ripetitività, hanno lo scopo di aiutare l'assimilazione dei concetti e di mettere in evidenza possibili difficoltà "nascoste" dietro un'apparente semplicità.

Il testo ha comunque natura provvisoria e contiene inevitabilmente imprecisioni ed errori. L'autore è grato di qualsiasi commento o segnalazione di errore che i lettori vogliano inviare. Per rendere tali segnalazioni più efficaci si raccomanda di redigerle in forma sintetica e puntuale facendo riferimento alla versione delle bozze (riportata in altro su ogni pagina), al capitolo, al numero di pagina e alla riga all'interno della pagina. Inviare le segnalazioni all'indirizzo di posta elettronica `iannello@unina.it` indicando sempre come *subject*: SEGNALAZIONE DISPENSE FI.

Capitolo 1

Algoritmi ed esecutori (bozze, v. 2.0)

1.1 Algoritmi ed esecutori

Poiché l'informatica è lo studio sistematico degli algoritmi, è opportuno introdurre subito una definizione per questo termine.

Definizione 1.1 *Un algoritmo è una sequenza finita di passi che portano alla realizzazione di un compito*

In questa definizione sono presenti due termini il cui significato non è competamente univoco. Tali termini sono: *passi* e *compito*. Con il termine passo intendiamo un'*operazione* di natura qualsiasi, ma ben definita, i cui effetti siano cioè completamente noti. Con il termine compito intendiamo un *risultato* di natura qualsiasi, ma anch'esso chiaramente e completamente definito.

La definizione di algoritmo implica la possibilità di fornire una sua descrizione ben definita. A tal fine si assume di disporre di un opportuno *linguaggio* che consenta di descrivere in modo non ambiguo tutti i singoli passi che formano l'algoritmo e l'ordine (la sequenza) con cui tali passi portano alla realizzazione del compito cui l'algoritmo è finalizzato. Il linguaggio usato per fornire una descrizione degli algoritmi verrà indicato nel seguito con il termine *linguaggio di programmazione*, l'attività di formulazione di un algoritmo attraverso un linguaggio di programmazione verrà indicata con il termine *programmazione* e il risultato di tale attività, cioè la descrizione dell'algoritmo verrà indicata coni il termine *programma*.

La definizione di algoritmo implica la presenza di un *esecutore* che può essere definito nel modo seguente.

Definizione 1.2 *L'esecutore di un algoritmo è l'entità che deve realizzare il compito, attuando la sequenza di passi che compongono l'algoritmo stesso.*

È evidente che l'esecutore può far fronte al suo compito se e solo se è in grado di *eseguire* tutti i passi della sequenza. Questo implica a sua volta che un algoritmo *presuppone* sempre la disponibilità di un opportuno esecutore, o, in altri termini, che un algoritmo dipende sia dal compito che si vuole realizzare, sia dall'esecutore per il quale è stato formulato.

È opportuno infine mettere in rilievo le relazioni tra l'esecutore e il linguaggio usato per formulare l'algoritmo. Per quanto si è detto, il linguaggio serve a descrivere in modo non ambiguo *tutte e sole* le operazioni che l'esecutore è in grado di eseguire. Il linguaggio pertanto descrive indirettamente tutte le potenzialità dell'esecutore, e sebbene non dia ragione della natura dell'esecutore stesso o di come esso possa eseguire le operazioni di cui è capace, esso descrive *tutto ciò che occorre sapere dell'esecutore per poter formulare algoritmi*. Per questo motivo, se si è interessati solo alla realizzazione di compiti tramite la formulazione di algoritmi, *si può identificare l'esecutore con il linguaggio usato per formulare gli algoritmi*, ignorando il suo funzionamento interno.

Esempio 1.1

Disponendo di un esecutore in grado di contare e spostare le palline di un pallottoliere, per effettuare la somma tra due numeri interi si può usare il seguente algoritmo:

1. Si inizializzi il pallottoliere: si spostino sulla sinistra della prima riga un numero di palline pari al primo addendo, si spostino sulla sinistra della seconda riga un numero di palline pari al secondo addendo, si spostino tutte le palline della terza riga a destra.
2. Si sposti una pallina dalla sinistra alla destra della prima riga e contestualmente se ne sposti una dalla destra alla sinistra della terza riga.
3. Si ripeta il passo 2 finché non si è svuotata la parte sinistra della prima riga.
4. Si sposti una pallina dalla sinistra alla destra della seconda riga e contestualmente se ne sposti una dalla destra alla sinistra della terza riga.
5. Si ripeta il passo 4 precedente finché non si è svuotata la parte sinistra della seconda riga.
6. Lettura del risultato: il numero di palline che si viene a trovare alla sinistra della terza riga al termine delle operazioni è il risultato.

Esempio 1.2

Disponendo di un esecutore in grado di comprendere il significato delle cifre decimali, di leggere e scrivere cifre su una lavagna, e di calcolare la somma e il riporto della somma di due cifre decimali e di un riporto, per effettuare la somma tra due numeri interi si può usare il seguente algoritmo:

1. Si inizializzi la lavagna: si pulisca la lavagna e si scrivano i due numeri uno sotto l'altro incolonnati a destra.
2. Si consideri la coppia costituita dalle cifre più a destra dei due numeri e si consideri come riporto iniziale 0.
3. Si calcoli la somma e il riporto della coppia di cifre e del riporto considerati.
4. Si scriva la somma sotto le due cifre considerate.
5. Finché vi sono cifre a sinistra di quelle appena considerate, si torni al passo 3 considerando la coppia costituita da tali cifre e il riporto appena calcolato.
6. Se l'ultimo riporto calcolato è diverso da 0, scriverlo a sinistra dell'ultima somma scritta sulla lavagna.
7. Lettura del risultato: il risultato è scritto sulla lavagna sotto i due addendi.

Nella figura 1.1 è mostrata la successione di passi che si ha quando i numeri da sommare sono 1309 e 3616. Nella figura, che mostra il contenuto della lavagna man mano che l'esecuzione dell'algoritmo procede, è indicato come si susseguono i passi appena elencati.

Osservazione 1.1

Se si confronta la natura dei passi dei due algoritmi presentati negli esempi 1.1 e 1.2, si possono mettere in evidenza le differenti capacità dei due esecutori.

Il primo esecutore è costituito dal pallottoliere e da un'entità in grado di:

- distinguere tra le tre righe del pallottoliere;

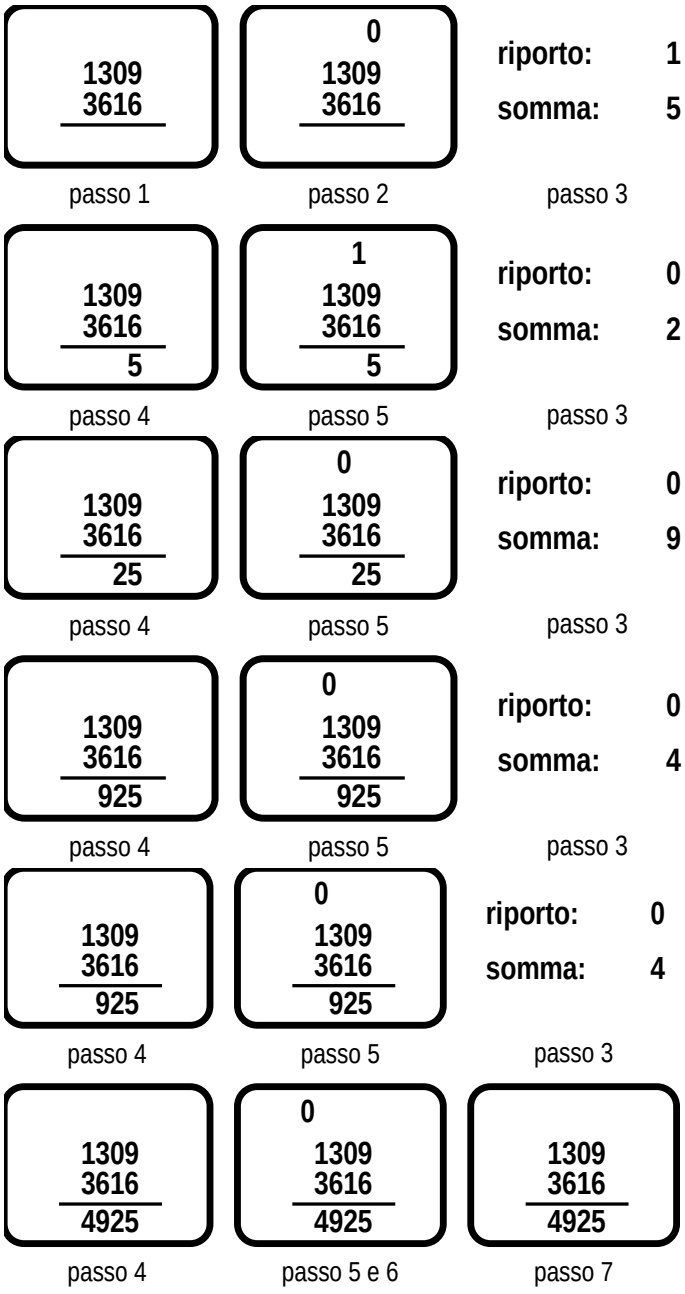


Figura 1.1: Esempio di esecuzione dell'algoritmo dell'esempio 1.2.

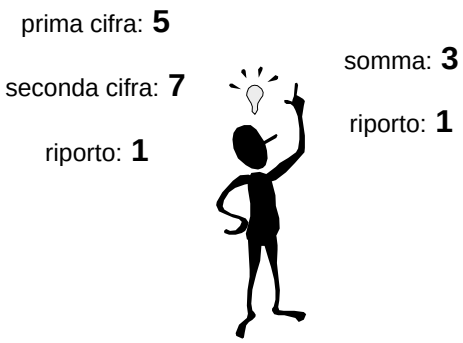


Figura 1.2: Operazione fondamentale di cui è capace l'esecutore dell'esempio 1.2.

- spostare da sinistra a destra (e viceversa) una pallina su una riga assegnata;
- contare le palline contigue;
- prendere decisioni sul prossimo passo da eseguire.

Il secondo esecutore è costituito dalla lavagna, da un gesso o altro strumento idoneo e da un'entità in grado di:

- fare tutte le cose descritte all'inizio dell'esempio 1.2, e in particolare sapere compiere l'operazione di somma tra due cifre e un riporto, esemplificata graficamente nella figura 1.2;
- prendere decisioni sul prossimo passo da eseguire.

Si noti che, in entrambi i casi, strumenti come il pallottoliere, la lavagna, il gesso *fanno parte dell'esecutore* (l'entità che opera non potrebbe fare nulla senza questi strumenti) e che l'entità che opera non deve essere necessariamente un uomo, ma può anche essere una macchina (per esempio un robot) con opportune caratteristiche.

È abbastanza evidente comunque che il secondo esecutore è in grado di eseguire operazioni più complesse del primo. Per rendersene conto può essere utile osservare, ad esempio, che l'operazione fondamentale del secondo algoritmo mostrata nella figura 1.2 implica che l'esecutore conosca le tabelle riportate in figura 1.3. Tali tabelle sono quelle che definiscono la somma tra cifre con riporto e che abbiamo imparato a usare (generalmente non sotto forma di tabelle) quando ci è stato insegnato a fare le somme tra numeri rappresentati con cifre decimali.

Per illustrare in modo sommario come potrebbe funzionare internamente un esecutore per l'algoritmo dell'esempio 1.2, nel caso in cui le cifre e il riporto da sommare siano quelle mostrate nella figura 1.2, l'esecutore usando la seconda tabella (somma, con riporto = 1) troverebbe la cifra risultato 3 all'incrocio della riga 5 e della colonna 7, e usando la quarta tabella (riporto successivo, sempre con riporto = 1) troverebbe come riporto successivo 1, sempre all'incrocio della riga 5 e della colonna 7.

Osservazione 1.2

Dopo aver confrontato gli esecutori, è opportuno confrontare anche gli algoritmi per vedere come le differenze tra gli esecutori si ripercuotono sull'algoritmo (si ricordi che il compito dei due algoritmi è lo stesso). Intuitivamente ci aspettiamo che l'algoritmo dell'esempio 1.1, che fa riferimento a un esecutore con minori capacità, sia in qualche modo più complesso e/o più limitato dell'algoritmo dell'esempio 1.2, che fa riferimento ad un esecutore capace di operazioni più potenti.

In effetti, confrontando la formulazione dei due algoritmi, non appaiono particolari vantaggi del secondo algoritmo. Anzi per certi versi le maggiori capacità dell'esecutore si traducono in passi più "ricchi" di contenuto (scrivere cifre

somma

riporto = 0

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

riporto = 1

	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3	4	5
6	7	8	9	0	1	2	3	4	5	6
7	8	9	0	1	2	3	4	5	6	7
8	9	0	1	2	3	4	5	6	7	8
9	0	1	2	3	4	5	6	7	8	9

riporto successivo

riporto = 0

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	1	1
3	0	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	1	1	1	1
6	0	0	0	0	1	1	1	1	1	1
7	0	0	0	1	1	1	1	1	1	1
8	0	0	1	1	1	1	1	1	1	1
9	0	1	1	1	1	1	1	1	1	1

riporto = 1

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	0	1	1	1
3	0	0	0	0	0	0	1	1	1	1
4	0	0	0	0	0	1	1	1	1	1
5	0	0	0	0	1	1	1	1	1	1
6	0	0	0	1	1	1	1	1	1	1
7	0	0	1	1	1	1	1	1	1	1
8	0	1	1	1	1	1	1	1	1	1
9	1	1	1	1	1	1	1	1	1	1

Figura 1.3: Tabelle che forniscono la somma e il riporto successivo date due cifre decimali e un riporto (0 o 1).

sulla lavagna sembra cosa più complessa che spostare palline se si pensa ad esempio di farlo fare a un robot). Dove è allora il vantaggio di usare un esecutore più potente? La risposta è duplice e mette in evidenza due aspetti molto importanti nella valutazione degli algoritmi:

1. Da un punto di vista pratico la coppia algoritmo-esecutore dell'esempio 1.1 ha probabilmente molti limiti perché non è ragionevole supporre che il pallottolire possa avere più di qualche decina di palline per riga e questo limita la possibilità di fare somme al caso di numeri piccoli; la coppia algoritmo-esecutore dell'esempio 1.2 non ha tali limiti dal momento che si può presumere di scrivere sulla lavagna praticamente qualunque numero di interesse pratico.

2. Supponiamo ottimisticamente che il pallottoliere abbia 100 palline su ogni riga, se vogliamo sommare i numeri 45 e 37, dovremo spostare complessivamente 45 palline sulla prima riga, 37 sulla seconda e 82 sulla terza per un totale di 164 operazioni elementari. Se eseguiamo la stessa somma sulla seconda coppia algoritmo-esecutore dobiamo invece scrivere sulla lavagna le quattro cifre dei due addendi, due riporti e due cifre del risultato per un totale di 8 scritture! Se assumiamo che il tempo necessario per spostare una pallina e per scrivere una cifra sulla lavagna sia confrontabile, è evidente l'enorme risparmio di tempo che si ha nel caso dell'esempio 1.2.

Queste osservazioni confermano l'intuizione che la seconda coppia algoritmo-esecutore sia in qualche modo migliore della prima, anche se con riferimento ad aspetti non immediatamente evidenti.

Sebbene il confronto tra le due coppie algoritmo-esecutore abbia messo in evidenza molti punti interessanti, tuttavia lo scopo della discussione è stato solo quello di chiarire meglio i concetti di algoritmo e di esecutore, mettendone in luce la natura con degli esempi. Non è opportuno per ora sviluppare ulteriormente questo tipo di argomenti in quanto il nostro primo obiettivo è quello di introdurre i principi di costruzione degli algoritmi, piuttosto che la loro analisi.

Osservazione 1.3

È opportuno che, sulla base di questi primi due esempi di algoritmi con inclusa descrizione di massima dell'esecutore, vengano fatte alcune osservazioni sul linguaggio usato per descriverli.

In entrambi i casi si tratta del linguaggio naturale, arricchito da una numerazione dei passi. La numerazione ha lo scopo di evidenziare quali sono i passi dell'algoritmo e di permettere di esprimere in modo chiaro la successione con cui tali passi devono essere considerati.

Un secondo punto, particolarmente importante da evidenziare, è la presenza di passi di tipo decisionale, oltre a passi puramente esecutivi. Normalmente infatti, i passi vanno eseguiti nell'ordine con cui vengono elencati e numerati. Tuttavia in alcuni casi è possibile che la sequenza di esecuzione subisca delle variazioni sulla base della verità o della falsità di opportune affermazioni Ad esempio, nel primo algoritmo, i passi 3 e 5 richiedono di decidere se ripetere i passi precedenti o se continuare con il passo successivo.

Una terza osservazione riguarda la presenza di uno o più passi iniziali (detti di *inizializzazione*) che hanno lo scopo di "preparare il terreno" all'algoritmo vero e proprio, e uno o più passi finali, volti alla presentazione del risultato.

Esempio 1.3

A conclusione del paragrafo, illustriamo un ulteriore esempio di algoritmo.

Disponendo di un esecutore in grado di utilizzare una calcolatrice tascabile e di leggere una lista di numeri scritti su un foglio di carta, per calcolare la somma della lista di numeri si può usare il seguente algoritmo:

1. Si inizializzi la calcolatrice: si prema il tasto di cancellazione.

2. Si consideri come numero corrente il primo numero della lista.

3. Si digiti il numero corrente.

4. Se la lista non è terminata si prema il tasto '+', si consideri come numero corrente il numero successivo e si ripeta il passo 3, altrimenti si prema il tasto '='.

5. Lettura del risultato: il risultato appare sul display della calcolatrice.

L'algoritmo precedente assume di utilizzare una calcolatrice tascabile di tipo scientifico. Se al suo posto l'esecutore utilizza una calcolatrice per contabilità[1], l'algoritmo deve essere modificato come segue:

1. Si inizializzi la calcolatrice: si prema il tasto di cancellazione.

2. Si consideri come numero corrente il primo numero della lista.

3. Si digiti il numero corrente e si prema il tasto '+'.

4. Se la lista non è terminata si consideri come numero corrente il numero successivo e si ripeta il passo 3, altrimenti si vada al passo successivo.

5. Lettura del risultato: il risultato appare sul display della calcolatrice.

Osservazione 1.4

L'esempio, considerando due esecutori che differiscono solo per un aspetto di dettaglio, mette bene in evidenza la dipendenza dell'algoritmo dalle capacità dell'esecutore. In entrambi i casi l'esecutore è un individuo dotato di una calcolatrice, ma cambiando il tipo di calcolatrice cambiano (nelle modalità operative) le capacità dell'esecutore (individuo+calcolatrice) e cambia di conseguenza l'algoritmo. Si noti che la modifica non riguarda necessariamente la natura delle operazioni specificate, ma può riguradare i passi decisionali e l'ordine con cui devono essere eseguite le operazioni.

Esercizi proposti

1. Con riferimento agli esempi 1.1–1.3 indicare quali passi sono di natura esecutiva e quali di natura decisionale (nel caso dell'esempio 1.1 la risposta e già stata data nell'osservazione 1.3).

2. Disponendo di un esecutore in grado di leggere, scrivere e cancellare numeri su una lavagna e di effettuare somme e sottrazioni tra numeri qualsiasi, formulare un algoritmo per calcolare il prodotto di due numeri interi positivi (*suggerimento*: il prodotto si può calcolare attraverso somme ripetute).

3. Disponendo di un esecutore in grado di leggere, scrivere e cancellare numeri su una lavagna, di effettuare somme e sottrazioni tra numeri qualsiasi, e di stabilire se un numero è minore di un altro numero, formulare un algoritmo per calcolare la divisione intera di due numeri interi positivi (*suggerimento*: la divisione intera si può calcolare attraverso differenze ripetute).

4. Disponendo dello stesso esecutore dell'esercizio 2, formulare un algoritmo per calcolare il prodotto di due numeri interi relativi.

5. Descrivere in modo sintetico, ma preciso, le capcità di un esecutore in grado di eseguire il seguente algoritmo:

[1] Nelle calcolatrici scientifiche la pressione del tasto '+' ha come effetto l'esecuzione dell'operazione impostata precedentemente (se esiste) e imposta una nuova operazione di somma che ha come operandi il numero che compare sul display dopo la pressione del tasto e il numero che viene digitato successivamente. Nelle calcolatrici per contabilità invece, la pressione del tasto '+' ha come effetto l'esecuzione di una somma che ha come operandi l'ultimo numero digitato e il numero che compariva sul display prima di tale digitazione. In altre parole, nel primo caso l'ultimo operando della somma deve essere immesso dopo aver premuto il tasto '+' e la somma viene effettuata quando si richiede l'operazione successiva, nel secondo caso l'ultimo operando della somma deve essere immesso prima di premere il tasto '+' e la somma viene effettuata immediatamente. Di conseguenza nelle calcolatrici scientifiche occorre premere il tasto '=' per visualizzare il risultato dell'ultima somma impostata.

(a) riempire una pentola d'acqua
(b) versare nella pentola 3 cucchiai di sale grosso
(c) accendere un fornello e metterci sopra la pentola
(d) attendere che l'acqua bolla
(e) mettere nella pentola 500 gr di pasta
(f) attendere 10 minuti
(g) assagiare un pezzo di pasta
(h) se la pasta non è completamente cotta attendere 30 secondi e poi ripetere il passo 5g
(i) spegnere il fornello e scolare la pasta
(j) condire la pasta in un recipiente opportuno

6. Rispondere alle seguenti domande sull'esecutore dell'algoritmo descritto nell'esercizio 5:
   - l'esecutore deve saper misurare il tempo e se si con quale precisione?
   - l'esecutore deve saper stabilire se l'acqua è sufficientemente salata?
   - l'esecutore deve saper contare?
   - l'esecutore deve disporre di un dispositivo per scolare la pasta o deve andarselo a cercare?
   - l'esecutore deve disporre di una bilancia e se si quanto precisa?

1.2 Risoluzione di problemi

Da quanto detto nel paragrafo 1.1 risulta che una conoscenza adeguata del linguaggio di programmzione è condizione necessaria per formulare e comprendere gli algoritmi. Ad essa deve pertanto essere dedicata una gran parte dell'attenzione di chi si avvicina allo studio dell'informatica. In particolare, la conoscenza del linguaggio di programmazione fornisce una conoscenza adeguata delle possibilità dell'esecutore, rendendo possibile la formulazione di algoritmi effettivamente eseguibili, cioè di programmi.

Tuttavia la conoscenza del linguaggio di programmazione non è tutto. In particolare essa fornisce informazioni sulle singole operazioni che l'esecutore è in grado di compiere, ma non dice nulla su come esse possano essere organizzate in una sequenza che porti al risultato finale voluto. Tale attività di programmazione è di natura creativa e richiede in generale capacità che si acquistano solo con l'esperienza, non riconducibili a un insieme di norme perfettamente definite. La programmazione di un algoritmo è pertanto un procedimento complesso, in cui svolge un ruolo importante l'intuizione e che solo in parte può essere descritto in modo sistematico.

Tuttavia, è possibile fin da ora mettere in evidenza alcuni elementi che svolgono un ruolo particolarmente importante in questo processo e che saranno utili in seguito per fornire un quadro di riferimento a cui ricondurre le nozioni e le tecniche illustrate nei prossimi capitoli.

Gli elementi che riteniamo utile introdurre sin d'ora sono quattro:

- la specifica del compito
- i dati di ingresso e i dati di uscita
- i casi di test
- le nozioni di sequenza statica e sequenza dinamica di un algoritmo.

A ciascuno di essi dedichiamo uno dei seguenti paragrafi.

1.2.1 Specifica

Si assuma di aver fissato l'esecutore e di volere che una persona (o un gruppo di persone) formuli un algoritmo per tale esecutore. Indichiamo con il termine *programmatore* tale persona.

Poiché un algoritmo ha come obiettivo la realizzazione di un compito, prima della formulazione di un algoritmo, il programmatore deve disporre di una descrizione sufficientemente chiara del compito da risolvere. Tale descrizione, che nel caso di semplici problemi viene solitamente denominata traccia del problema, viene indicata nel caso più generale con il temine *specifica*.

Si noti che la specifica di un problema deve definire in modo completo e non ambiguo in *che cosa* consiste il compito da svolgere e non *come* esso deve essere svolto. La specifica inoltre non deve fare nessun riferimento esplicito all'esecutore. La specifica infatti descrive un problema che, in principio, può essere risolto usando esecutori diversi. È compito dell'algoritmo descrivere come un particolare esecutore debba risolvere il problema posto dalla specifica.

Negli esempi precedenti le specifiche sono contenute nella frase iniziale che precede la formulazione dell'algoritmo:

- Esempi 1.1 e 1.2: effettuare la somma tra due numeri interi;
- Esempio 1.3: calcolare la somma di una lista di numeri.

Come abbiamo detto, quando un algoritmo è diretto ad un esecutore automatico, esso viene formulato in un apposito linguaggio che permette di descrivere esattamente il comportamento dell'esecutore. La specifica, invece è diretta normalmente ad un operatore umano ed è pertanto espressa (interamente o in parte) in linguaggio naturale. Se da un lato questo consente una grande flessibilità e concisione, tuttavia introduce la possibilità di *incompletezza* e *ambiguità*.

Per questo motivo, prima di formulare un algoritmo è opportuno effettuare alcune operazioni preliminari sulla specifica volte a colmare eventuali incompletezze e a chiarire possibili ambiguità. In particolare, la specifica di un problema *deve permettere l'individuazione dei dati di ingresso e di uscita, e la generazione di casi di test*.

1.2.2 Dati di ingresso e dati di uscita

Ogni problema da risolvere presuppone delle informazioni da cui partire e delle informazioni da produrre. Le prime vengono dette comunemente *dati di ingresso* e devono essere fornite all'esecutore prima o durante l'esecuzione dell'algoritmo nelle modalità previste dall'algoritmo stesso. Le seconde vengono dette comunemente *dati di uscita* o *risultati* e vengono prodotte dall'esecutore durante o al termine dell'esecuzione dell'algoritmo nelle modalità previste dall'algoritmo stesso.

Ad esempio si consideri il seguente problema:

> **Specifica**: Calcolare l'area di un cerchio, noto il raggio.

Da un rapido esame del testo della specifica si deduce immediatamente che l'unico dato di ingresso è il raggio del cerchio e che l'unico dato di uscita è l'area del cerchio.

Analogamente, con riferimento agli esempi 1.1, 1.2 e 1.3 abbiamo i seguenti dati di ingresso e di uscita:

- Esempi 1.1 e 1.2: i dati di ingresso sono due numeri da sommare, il dato di uscita è la somma;
- Esempio 1.3: i dati di ingresso sono una lista di numeri da sommare, il dato di uscita è la somma.

Sebbene possa sembrare banale, è opportuno osservare che deve essere sempre possibile determinare i dati di ingresso e di uscita a partire dalla specifica. In caso contrario, la specifica sarebbe evidentemente incompleta o ambigua: come potremmo infatti formulare un algoritmo se non fosse possibile stabilire con precisione quali sono i dati di partenza e quali sono quelli di arrivo?

Anche se nella maggior parte degli esercizi proposti in questa introduzione all'informatica i dati di ingresso e di uscita sono ovvi, è opportuno abituarsi a esplicitarli nell'analisi della specifica che precede la formulazione di un algoritmo, perchè questo non è vero per i problemi reali, dove i dati di ingresso e di uscita possono essere numerosi e ammettere varianti.

1.2.3 Casi di test

Oltre a permettere l'identificazione dei dati di ingresso e di uscita, la specifica deve consentire di individuare esempi concreti di dati di ingresso e di derivare da ciascun esempio i corrispondenti dati di uscita attesi. Tali esempi vengono indicati con il termine *casi di test*.

Ad esempio, con riferimento al problema dell'area del cerchio si possono scegliere i seguenti due casi di test:

Esempio di dati di ingresso	`1`
Uscita attesa	`3.14`

Esempio di dati di ingresso	`5`
Uscita attesa	`78.5`

Analogamente, per gli esempi 1.1 e 1.2, sono due casi di test distinti i seguenti:

Esempio di dati di ingresso	`57 71`
Uscita attesa	`128`

Esempio di dati di ingresso	`0 13`
Uscita attesa	`13`

e per l'esempio 1.3 sono due casi di test distinti i seguenti:

Esempio di dati di ingresso	`56 73 18 2 10`
Uscita attesa	`159`

Esempio di dati di ingresso	`27`
Uscita attesa	`27`

Si noti che non è stato indicato come ricavare l'uscita attesa dai dati di ingresso. In questa fase le uscite attese vengono derivate con un qualunque metodo già disponibile (un esperto del problema, un documento scritto, una tabella, un esecutore diverso da quello fissato, ecc). Nella fase in cui si analizza la specifica, infatti, non si dispone ancora di un algoritmo per l'esecutore fissato, essendo proprio questo l'obiettivo da raggiungere.

In alcuni casi particolarmente complessi tuttavia, pur essendo possibile in linea di principio derivare le uscite attese dai casi di test (in caso contrario il problema sarebbe mal posto), le uscite potrebbero non essere di fatto derivabili senza l'aiuto dell'esecutore[2]. Questi casi tuttavia superano gli obiettivi di questa trattazione e assumeremo quindi nel seguito che sia possibile per il programmatore derivare in qualche modo le uscite attese di un problema da una specifica adeguata, a prescindere dalla disponibilità dell'algoritmo risolutivo per l'esecutore fissato.

L'utilità di identificare un certo numero di casi di test e di calcolare le corrispondenti uscite prima di formulare l'algoritmo per l'esecutore fissato è ancora una volta legata al problema della possibile ambiguità e/o incompletezza della specifica: i casi di test aiutano a scoprire ambiguità e incompletezze nella specifica prima di iniziare a formulare l'algoritmo, o a scoprire errori nell'algoritmo dovuti ad ambiguità e incompletezze della specifica non rilevate.

[2] Si pensi ad esempio ad un moderno programma con interfaccia visuale: gli input sono generalmente i clic e le coordinate del mouse che assumono siginificato in dipendenza di ciò che è presente sullo schermo, e gli output includono elementi grafici come forme, colore, posizione, ecc. È abbastanza evidente che in queste ipotesi è di fatto impossibile derivare con precisione l'output atteso, anche se è normalmente possibile derivarlo a meno di alcuni dettagli grafici.

Esempio 1.4

Con riferimento all'esempio 1.1, se tra i casi di test includiamo il caso in cui gli addendi siano 0 e 3, deriviamo l'uscita attesa 3. Se facciamo eseguire l'algoritmo a un robot in grado di spostare le palline, ci rendiamo conto che il robot, eseguendo il secondo passo, potrebbe effettuare i movimenti richiesti dal passo 2 e spostare una pallina sulla terza riga da destra a sinistra anche se non vi sono palline a sinistra sulla prima riga. Tale possibilità dipende dal fatto che la decisione di passare al quarto passo dell'algoritmo viene presa dopo aver eseguito comunque almeno una volta il secondo passo. In tal caso il risultato alla fine del procedimento sarebbe pari a 4! Il problema, in questo caso, è legato alla formulazione dell'algoritmo, che assume implicitamente che gli operandi siano entrambi maggiori di 0 (o che l'esecutore interpreti in modo "intelligente" i passi che compongono l'algoritmo). In realtà la specifica non dice nulla su cosa fare nel caso di un operando pari a 0, e il caso di test è servito a mettere in evidenza il problema.

Si noti che il problema di ambiguità evidenziato può essere risolto in due modi:

1. modificando la specifica: imponendo di effettuare la somma tra due numeri interi *positivi*;
2. modificando l'algoritmo: anticipando il passo in cui si decide se ripetere il secondo passo o se passare al quarto passo.

Entrambi i metodi sono validi sotto il profilo tecnico. La scelta deve essere fatta sulla base di criteri di opportunità.

Esempio 1.5

Presentiamo ancora un esempio leggermente più complesso per completare l'illustrazione dei concetti di specifica, dati di ingresso e uscita e casi di test.

Specifica. Scrivere un programma che legge un numero naturale N seguito da una lista di N numeri interi e stampa la lista letta invertendo a due a due gl elementi di posto pari e dispari.

Dati di ingresso e di uscita. Dalla specifica si ricava immediatamente che i dati di ingresso sono un numero naturale e una lista di numeri interi (sottinteso relativi). La lunghezza della lista è pari al primo numero (naturale) letto. Si noti che la lunghezza della lista può essere uguale a 0.

I dati di uscita sono una lista di interi relativi ottenuta da quella di ingresso spostando alcuni elementi. Si noti in particolare che la lunghezza della lista di uscita è uguale a quella della lista di ingresso.

Casi di test. È chiaro che i casi di test si distinguono essenzialmente per il valore del numero naturale N. In particolare sembrano siginificativamente diversi i seguenti valori di N: 0, 1, un qualunque numero pari (per es. 4), un qualunque numero dispari (per es. 5).

Provando a derivare l'uscita attesa nei casi $N = 1$ e $N = 5$ ci si accorge immediatamente che la traccia presenta un'ambiguità. Nel caso di valori dispari di N, come bisogna trattare l'ultimo elemento della lista per il quale non esiste un elemento di posizione pari con cui scambiarlo di posto? L'ambiguità può essere risolta in diversi modi: riportando in uscita l'ultimo elemento senza spostarlo, non riportando in uscita l'ultimo elemento, segnalando un errore. La considerazione che la traccia sembra implicitamente affermare che la lunghezza della lista di uscita coincide con quella della lista di ingresso fa propendere per la prima soluzione che appare anche la più ragionevole.

Da queste osservazioni possiamo derivare i seguenti casi di test (il simbolo – indica che non è attesa alcuna uscita):

Primo caso di test

Esempio di dati di ingresso	`0`
Uscita attesa	–

Secondo caso di test

Esempio di dati di ingresso	`1 7`
Uscita attesa	`7`

Terzo caso di test

Esempio di dati di ingresso	`4 7 -9 0 3`
Uscita attesa	`-9 7 3 0`

Quarto caso di test

Esempio di dati di ingresso	`5 7 -9 0 3 1`
Uscita attesa	`-9 7 3 0 1`

1.2.4 Sequenza statica e dinamica di un algoritmo

Si è detto che un algoritmo è una sequenza di passi. È possibile individuare due tipi di sequenze. Il primo tipo, detto *sequenza statica*, è la sequenza con cui i passi sono elencati nella formulazione dell'algoritmo stesso. Esempi di sequenze statiche sono gli elenchi numerati che descrivono gli algoritmi discussi nel paragrafo 1.1.

Se ora assumiamo di voler effettuare la somma $2 + 3$ con questo algoritmo è facile verificare che l'esecutore, seguendo l'algorimo, eseguirà la sequenza di 12 passi:

1, 2, 3, 2, 3, 4, 5, 4, 5, 4, 5, 6

Se invece effettuiamo la somma $3 + 2$, l'esecutore eseguirà la sequenza di 12 passi:

1, 2, 3, 2, 3, 2, 3, 4, 5, 4, 5, 6

che pur avendo ancora 12 passi è diversa dalla precedente.

La *sequenza dinamica* è quindi la sequenza di passi effettivamente eseguita dall'esecutore *in una particolare esecuzione dell'algoritmo*. Essa dipende dalla sequenza statica, ma in generale non coincide con essa. In particolare le due sequenze coincidono solo quando l'algoritmo non contiene passi decisionali.

Osservazione 1.5

È importante osservare che, mentre esiste un'unica sequenza statica, esistono in generale molte sequenze dinamiche diverse, ciascuna associata a particolari valori dei dati di ingresso (gli operandi della somma nell'esempio precedente). Poiché specifici valori dei dati di ingresso determinano un caso di test, possiamo concludere che la sequenza dinamica eseguita dall'algoritmo è determinata dal caso di test considerato.

Esercizi proposti

1. Indicare i dati di ingresso e di uscita dei problemi presentati negli esercizi 2–4 proposti in fondo al paragrafo 1.1
2. Un algoritmo viene di norma formulato sulla base di una specifica già fornita. Discutere se ha senso redigere una specifica sulla base di un algoritmo già formulato. Per rendersi conto di cosa questo significhi provare a redigere la specifica corrispondente all'algoritmo presentato nell'esercizio 5 proposto in fondo al paragrafo 1.1.

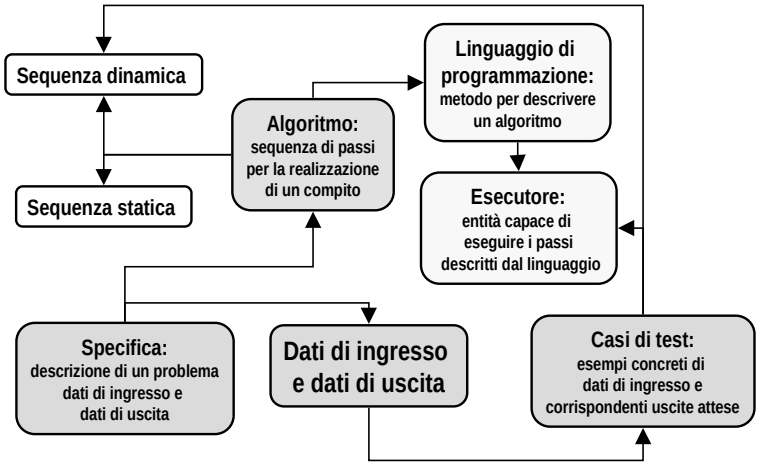


Figura 1.4: Relazioni tra specifica, algoritmo, linguaggio ed esecutore.

3. Dopo aver redatto una specifica (cfr. esercizio 2), indicare quali sono i dati di ingresso e di uscita del problema presentato nell'esercizio 5 proposto in fondo al paragrafo 1.1.

4. Determinare 3 casi di test per i seguenti problemi:

   (a) Leggere un intero k e una lista di interi preceduta dalla sua lunghezza, e stampare la lista ottenuta eliminando i valori maggiori di k dalla lista letta.

   (b) Leggere una lista di interi terminata dal valore -1000, e stampare i valori della lista che seguono il primo valore negativo presente nella lista.

   (c) Leggere due liste di interi, ciascuna preceduta dalla sua lunghezza, e stampare un messaggio che dice se le due liste sono uguali.

   (d) Leggere due liste di interi, ciascuna preceduta dalla sua lunghezza, e stampare un messaggio che dice se le due liste contengono lo stesso numero di zeri.

5. Determinare la sequenza dinamica dell'algoritmo dell'esempio 1.1 nei seguenti casi di test:

   $4 + 6$
   $6 + 4$
   $1 + 2$
   $10 + 4$

6. Osservando le soluzioni dell'esercizio 5 ricavare la legge che lega la lunghezza della sequenza dinamica al risultato della somma.

1.3 Riepilogo

Uno schema riepilogativo dei concetti introdotti è riportato nella figura 1.4.

Al centro c'e' l'oggetto principale del nostro studio: gli algoritmi. Un algoritmo è però sempre legato ad una specifica, che descrive il problema che l'algoritmo risolve, e a un esecutore, che è l'entità che dovrà eseguire l'algoritmo.

Il legame tra algoritmo ed esecutore è costituito dal linguaggio di programmazione che è il mezzo per descrivere l'algoritmo utilizzando le operazioni che possono essere attuate dall'esecutore. Abbiamo osservato che in pratica chi deve ideare e formulare l'algoritmo interagisce con l'esecutore unicamente attraverso il linguaggio di programmazione. Quindi egli non deve necessariamente conoscere la natura e il funzionamento dell'esecutore, ma è sufficiente che conosca il linguaggio con cui istruirlo. In altre parole, dal punto di vista del programmatore, esecutore e linguaggio coincidono.

Nella figura è inoltre evidenziato come la specifica determini i dati di ingresso e di uscita e come i casi di test derivati dalla specifica forniscano le informazioni necessarie per l'esecuzione dell'algoritmo e per verificare che i risultati prodotti siano corretti. Inoltre i casi di test determinano una sequenza dinamica che corrisponde alla sequenza di passi eseguita dall'esecutore durante la risoluzione del problema. Infine, la sequenza dinamica è determinata, oltre che dal caso di test, anche, ovviamente, dall'algoritmo che definisce anche la sequenza statica dei passi.

Capitolo 2

Condizioni, operatori di relazione e operatori logici (bozze, v. 2.0)

Nel capitolo 1 abbiamo osservato come negli algoritmi siano presenti passi decisionali e come tali passi contengano affermazioni la cui verità o falsità determina il successivo passo da eseguire.

Nel seguito, indicheremo tali affermazioni con il termine *condizioni* (per indicarle sono spesso usati nella letteratura anche i termini *proposizione* e *predicato*). Nei casi più semplici tali condizioni sono atomiche (cioè non divisibili in condizioni più semplici), ma in generale le condizioni sono *composte*, cioè costituite da condizioni più semplici collegate da opportuni *operatori logici*.

In questo capitolo introdurremo le principali modalità per esprimere condizioni sia atomiche che composte. In particolare introdurremo gli operatori logici e le loro proprietà. Inoltre esamineremo il problema, di interesse pratico nella programmazione, di confrontare condizioni apparentemente diverse per verificare se esse risultano equivalenti.

2.1 Condizioni atomiche

Cominciamo col definire con precisione il concetto di condizione.

Definizione 2.1 *Una* condizione *è una qualsiasi affermazione che può essere vera o falsa.*

Un esempio di condizione può essere la frase (cfr. l'esempio 1.3):

la lista non è terminata

Si noti che la condizione è atomica perchè non può essere divisa in parti che siano ancora delle condizioni. Ad esempio le parole "la lista" non esprimono alcuna affermazione.

Osservazione 2.1

Dal contesto dell'esempio 1.3 la condizione considerata risulterà vera se vi sono ancora numeri della lista che non sono ancora stati sommati, falsa altrimenti. Durante l'esecuzione dell'algoritmo, la condizione viene presa in considerazione un certo numero di volte, a seconda della lunghezza della lista di numeri da sommare. In momenti diversi della stessa esecuzione dell'algoritmo, la condizione può assumere valori di verità diversi.

Ad esempio, se la lista comprende tre numeri, la condizione viene considerata quattro volte; le prime tre volte la condizione risulta vera e la quarta falsa, determinando così la terminazione dell'algoritmo.

Queste osservazioni, relative alla condizione atomica considerata, sono valide in generale per qualsiasi condizione, atomica o composta, impiegata nella formulazione di un algoritmo.

2.2 Operatori di relazione

Un altro esempio di condizione (cfr. l'esempio 1.2) è la seguente:

vi sono altre cifre a sinistra delle ultime considerate

Anche in questo caso si tratta evidentemente di una condizione (vera se vi sono ancora cifre da considerare, falsa altrimenti) e tale condizione è atomica perché le sue parti non sono condizioni. Tuttavia, in questo caso è possibile evitare di formulare la condizione in linuaggio naturale, esprimendola in modo sintetico nel modo seguente:

$$(\text{numero di cifre a sinistra}) > 0$$

Il simbolo $>$ è un'operatore che, confrontando due operandi numerici, permette di esprimere una condizione. Ad esempio, nel caso precedente, se il numero di cifre a sinistra non ancora considerate è maggiore di zero, allora la condizione risulta vera, altrimenti risulta falsa.

Esistono altri operatori simili a $>$ che permettono il confronto di entità la cui natura può essere molto varia. Essi si dicono per questo motivo *operatori di relazione*. Gli operatori di relazione più noti sono i seguenti:

- uguale (simbolo =)
- diverso (simbolo $\neq$)
- maggiore (simbolo $>$)
- minore (simbolo $<$)
- maggiore o uguale (simbolo $\geq$)
- minore o uguale (simbolo $\leq$)

e vengono solitamente utilizzati per confrontare valori numerici. In realtà gli operatori uguale e diverso si possono applicare a valori di qualunque natura (per esempio per confrontare il modello di due automobili, il colore dei capelli di due persone, ecc.), mentre gli altri operatori si possono applicare a valori su cui sia definita una relazione d'ordine (per esempio le lettere dell'alfabeto, i giorni della settimana, ecc.).

Gli operatori di relazione, pur non essendo operatori logici, consentono di costruire espressioni che possono essere usate come argomenti di operatori logici (vedi paragrafo successivo). Essi permettono pertanto di esprimere condizioni. Le condizioni espresse attraverso gli operartori di relazione sono atomiche in quanto non possono essere scomposte in parti che siano ancora delle condizioni.

Esercizi proposti

1. Calcolare il valore logico delle seguenti condizione atomiche:

$$5 < 7, \quad 7 < 5, \quad 7 = 5, \quad 7 > 0, \quad 5 = 5$$

2. Assumendo che il simbolo x valga 10, calcolare il valore logico delle seguenti condizione atomiche:

$$x < 7, \quad x > 7, \quad x = 7, \quad x = 0, \quad x = x$$

3. Ricalcolare il valore logico delle condizione atomiche dell'esercizio 2, assumendo che il simbolo x valga -10.
4. Per ciascuna condizione atomica dell'esercizio 2, calcolare i valori di x per i quali la condizione stessa il valore logico **vero**.
5. Per ciascuna condizione atomica dell'esercizio 2, calcolare i valori di x per i quali la condizione stessa il valore logico **falso**.

NOT

Operando	Risultato
falso	**vero**
vero	**falso**

AND

Operando 1	Operando 2	Risultato
falso	**falso**	**falso**
falso	**vero**	**falso**
vero	**falso**	**falso**
vero	**vero**	**vero**

OR

Operando 1	Operando 2	Risultato
falso	**falso**	**falso**
falso	**vero**	**vero**
vero	**falso**	**vero**
vero	**vero**	**vero**

Figura 2.1: Tabelle di verità degli operatori logici

2.3 Gli operatori logici e le loro proprietà

Fino ad ora in tutti gli esempi abbiamo considerato condizioni atomiche. In generale però le condizioni sono composte da condizioni più semplici, unite da *operatori* (o *connettivi*) *logici*.

Ad esempio, l'algoritmo seguito dall'impiegato di un ufficio postale nell'accettare conti corrente da parte degli utenti in coda potrebbe contenere un passo del tipo:

> accetta un altro conto corrente se *l'utente ha già consegnato meno di 5 conti corrente o non ci sono altri utenti in coda* altrimenti passa al successivo utente in coda

Questa passo contiene una condizione (testo in corsivo). Tale condizione è composta perché le sue parti:

> *l'utente ha già consegnato meno di 5 conti corrente*

e:

> *non ci sono altri utenti in coda*

sono ancora delle condizioni. Si noti poi che la seconda frase è a sua volta composta perché anche la sottofrase: *ci sono altri utenti in coda* è una condizione (questa volta atomica).

Analizzando la condizione composta si osserva facilmente che essa è stata costruita usando le parole "**e**" e "**non**", che corrispondono ai due operatori logici AND e NOT secondo l'espressione:

> (*l'utente ha già consegnato meno di 5 conti corrente*) AND (NOT (*ci sono altri utenti in coda*))

Gli operatori AND e NOT hanno come operandi delle condizioni, cioè espressioni che possono assumere i valori di verità **vero** o **falso**, e hanno come risultato ancora il valore **vero** o **falso**. La corrispondenza tra i valori degli operandi e il risultato è descritta dalle tabelle riportate in figura 2.1, dove compare anche un terzo operatore, l'operatore OR, che corrisponde alla congiunzione "**o**" nella costruzione di condizioni composte mediante il linguaggio naturale. Si noti che l'operatore NOT è unario, cioè ha un solo operando, mentre gli operatori logici AND e OR sono binari, cioè hanno due operandi.

Tabelle come quelle usate nella figura 2.1 per definire gli operatori logici vengono dette *tabelle di verità*. Esse possono essere usate per definire qualunque *funzione logica*, intendendo con questo termine una funzione che abbia valori logici sia come argomenti che come risultato. Le tabelle di verità prevedono un numero di colonne pari al numero di operandi dell'operatore più una (corrispondente al risultato), e un numero di righe pari a tutte le possibili

combinazioni di valori di verità per gli n argomenti della funzione. Se gli argomenti sono n, dal calcolo combinatorio sappiamo che il numero di righe della tabella (cioè di combinazioni) è 2^n.

Si può facilmente verificare che una condizione equivale a una funzione logica i cui argomenti sono le condizioni atomiche in essa contenute. Si può anche dimostrare che gli operatri logici NOT, AND e OR consentono di costruire qualsiasi condizione a partire da opportune condizioni atomiche. A partire dagli operatori logici impiegati per costruire la condizione e dalle loro tabelle di verità è dunque possibile costruire la tabella di verità della condizione. Tale tabella indica il suo valore logico della condizione in funzione del valore logico delle sue condizioni componenti.

Esempio 2.1

Si consideri la condizione composta:

devo lavorare, e ho riposato o ho bevuto del caffè.

Indicando con a la condizione atomica *devo lavorare*, con b la condizione atomica *ho riposato*, e con c la condizione atomica *ho bevuto del caffè*, la condizione composta corrisponde alla funzione logica di tre argomenti:

$$a \text{ AND } (b \text{ OR } c)$$

La tabella di verità della condizione ha pertanto $2^3 = 8$ righe e quattro colonne. Nelle prime tre colonne vengono riportate tutte le combinazioni di valori logici delle tre condizioni atomiche componenti. Nella quarta colonna è riportato, per ogni combinazione, il risultato calcolato sulla base delle tabelle di figura 2.1. Ad esempio, se $a = \mathbf{vero}$, $b = \mathbf{vero}$ e $c = \mathbf{falso}$, dalla tabella dell'OR si ha $b \text{ OR } c = \mathbf{vero}$ e da quella dell'AND abbiamo $a \text{ AND } (b \text{ OR } c) = \mathbf{vero}$. Ripetendo il procedimento per tutte le 8 combinazioni si ottiene la tabella di verità:

devo lavorare (a)	*ho riposato* (b)	*ho bevuto del caffè* (c)	condizione intermedia (b OR c)	condizione composta (a AND (b OR c))
falso	**falso**	**falso**	**falso**	**falso**
falso	**falso**	**vero**	**vero**	**falso**
falso	**vero**	**falso**	**vero**	**falso**
falso	**vero**	**vero**	**vero**	**falso**
vero	**falso**	**falso**	**falso**	**falso**
vero	**falso**	**vero**	**vero**	**vero**
vero	**vero**	**falso**	**vero**	**vero**
vero	**vero**	**vero**	**vero**	**vero**

Osservazione 2.2

Vale la pena osservare la disposizione delle 8 combinazioni di valori di verità corrispondenti ai tre argomenti. Nella terza colonna i valori **falso** e **vero** sono alternati, nella colonna immediatamente a sinistra sono alternati a coppie e nella prima sono alternati a gruppi di quattro. Seguendo questa regola si è sicuri di elencare tutte le combinazioni diverse *sempre nello stesso ordine*, cosa che risulta utile quando bisogna confrontare tra loro più tabelle di verità. Un altro vantaggio di questo modo di procedere è che se si sostituiscono i valori di verità **falso** e **vero** con le cifre binarie 0 e 1, le combinazioni risultano elencate in ordine crescente rispetto al valore numerico associato dalla rappresentazione posizionale binaria a ciascuna combinazione (vedi capitolo 3). È infine ovvio come quanto detto si possa immediatamente estendere ad un numero di operandi qualsiasi, cominciando però sempre dalla colonna più a destra.

Osservazione 2.3

Nella pratica è frequente il caso che si debba formulare una condizione c, composta da due condizioni più semplici a e b tale che verifichi la seguente tabella di verità:

a	b	c
falso	**falso**	**falso**
falso	**vero**	**vero**
vero	**falso**	**vero**
vero	**vero**	**falso**

La condizione composta deve cioè essere vera solo quando una delle due condizioni componenti è vera. Tale situazione corrisponde a una frase del tipo:

o la giacca è grigia o la camicia è bianca

che è evidentemente vera se la giacca è grigia, oppure se la camicia è bianca, ma non se la giacca è grigia *e* la camicia è bianca allo stesso tempo!

La funzione logica di due argomenti che corrisponde alla tabella di verità riportata sopra prende il nome di *OR esclusivo*, per distinguerlo dall'operatore OR introdotto in precedenza che viene detto *inclusivo* perché dà un risultato vero anche quando sono veri entrambi i suoi operandi. L'OR esclusivo viene anche indicato comunemente con il simbolo XOR (acronimo dell'inglese eXclusive OR).

Volendo esprimere la condizione:

(*la giacca è grigia*) XOR (*la camicia è bianca*)

usando gli operatori NOT, AND e OR, si può usare la seguente identità:

$$a \text{ XOR } b \equiv (a \text{ AND } (\text{NOT } b)) \text{ OR } (\text{NOT } a) \text{ AND } b)$$

e riscrivere pertanto la condizione nel modo seguente:

(*la giacca è grigia*) AND (NOT (*la camicia è bianca*)) OR
(NOT (*la giacca è grigia*)) AND (*la camicia è bianca*)

Esercizi proposti

1. Scrivere la tabella di verità corrispondente alla seguente funzione logica di tre variabili:

$$((\text{NOT } a) \text{ AND } b) \text{ OR } ((a \text{ AND } b) \text{ AND } c)$$

2. Scrivere la tabella di verità corrispondente alla seguente funzione logica di quattro variabili:

$$((\text{NOT } (a \text{ OR } b)) \text{ OR } ((\text{NOT } c) \text{ AND } d)) \text{ OR } a$$

3. Scrivere la tabella di verità corrispondente a una funzione logica a tre variabili che dia come risultato **vero** se e solo se due qualsiasi dei suoi argomenti sono uguali a **vero**.

2.4 Proprietà degli operatori logici

Gli operatori logici NOT, AND e OR godono di una serie di proprietà che è utile mettere in evidenza perchè consentono di trasformare se necessario le condizioni in forme alternative, equivalenti dal punto di vista logico, ma più convenienti al fine di una chiara ed efficace formulazione degli algoritmi.

Le proprietà di maggiore interesse pratico degli operatori AND e OR sono (a, b e c rappresentano tre valori logici qualsiasi):

- **associativa**:

$$a \text{ OR } (b \text{ OR } c) = (a \text{ OR } b) \text{ OR } c$$

$$a \text{ AND } (b \text{ AND } c) = (a \text{ AND } b) \text{ AND } c$$

- **commutativa**:

$$a \text{ OR } b = b \text{ OR } a$$

$$a \text{ AND } b = b \text{ AND } a$$

- **idempotenza**:

$$a \text{ OR } a = a$$

$$a \text{ AND } a = a$$

I due operatori godono poi della proprietà **distributiva** dell'uno rispetto all'altro. Valgono cioè le seguenti uguaglianze:

$$a \text{ OR } (b \text{ AND } c) = (a \text{ OR } c) \text{ AND } (a \text{ OR } c)$$

$$a \text{ AND } (c \text{ OR } c) = (a \text{ AND } c) \text{ OR } (a \text{ AND } c)$$

Sono infine molto utili le seguenti proprietà che coinvolgono anche l'operatore NOT:

- **del minimo e del massimo**

$$a \text{ OR } \mathbf{falso} = a$$

$$a \text{ OR } \mathbf{vero} = \mathbf{vero}$$

$$a \text{ AND } \mathbf{vero} = a$$

$$a \text{ AND } \mathbf{falso} = \mathbf{falso}$$

- **del complemento**

$$a \text{ OR } (\text{NOT } a) = \mathbf{vero}$$

$$a \text{ AND } (\text{NOT } a) = \mathbf{falso}$$

$$\text{NOT } (\text{NOT } a) = a$$

Si osservi che in tutti i casi elencati, a ogni uguaglianza ne corrisponde sempre una "duale" ottenuta sostituendo l'operatore AND con OR, l'operatore OR con AND, il valore di verità **falso** con **vero**, e il valore di verità **vero** con **falso**. Questo corrisponde ad una legge generale, detta *legge di dualità*, che vale per tutte le uguaglianze tra espressioni contenenti valori di verità e operatori logici. Ad esempio, è possibile dimostrare (lo si può fare tra l'altro usando le tabelle di verità come mostrato nel paragrafo 2.5) che per ogni possibile combinazione dei valori di verità di a e b vale l'identità:

$$a \text{ OR } b \equiv \text{NOT } ((\text{NOT } a) \text{ AND } (\text{NOT } b))$$

Per la legge di dualità vale pertanto senza bisogno di dimostrarlo:

$$a \text{ AND } b \equiv \text{NOT } ((\text{NOT } a) \text{ OR } (\text{NOT } b))$$

Le due identità, che permettono di sostituire un operatore con il suo duale, sono note come *Teoremi di De Morgan*. Si noti che nei passaggi da una proprietà alla sua duale l'operatore NOT non subisce modifiche.

In tutte le espressioni contenenti operatori logici abbiamo fino ad ora usato le parentesi per indicare l'ordine con cui gli operatori vanno applicati per calcolare il risultato. Tale pratica è del tutto analoga a quella usata abitualmente nelle espressioni contenenti operatori matematici. Per evitare però un uso eccessivo di parentesi, anche tra gli operatori

logici sono stabilte delle regole convenzionali di precedenza che permettono di omettere le parentesi quando l'ordine di applicazione degli operatori segue tali regole.

Le regole di precedenza tra gli operatori logici stabiliscono la precedenza massima per l'operatore NOT, seguito dall'operatore AND, a sua volta seguito dall'operatore OR, che ha la precedenza minima.

Pertanto le espressioni:

$$a \text{ OR NOT } b \text{ AND NOT } c$$

$$a \text{ OR NOT } b \text{ OR NOT } c$$

$$\text{NOT } a \text{ AND } b \text{ OR } c \text{ OR } a$$

sono rispettivamente equivalenti alle espressioni:

$$a \text{ OR } ((\text{NOT } b) \text{ AND } (\text{NOT } c))$$

$$(a \text{ OR } (\text{NOT } b)) \text{ OR } (\text{NOT } c)$$

$$(((\text{NOT } a) \text{ AND } b) \text{ OR } c) \text{ OR } a$$

dove sono state esplicitamente indicate tutte le parentesi.

Si noti che nel secondo esempio, dopo l'applicazione dei due operatori NOT, per la proprietà associativa dell'OR è del tutto inutile indicare un ordine di applicazione tra i due operatori OR. Analoga considerazione si potrebbe fare nel caso fossero presenti più operatori AND allo stesso livello di precedenza.

Parlando delle regole di precedenza è opportuno osservare che gli operatori di relazione hanno sempre precedenza su tutti gli operatori logici. Pertanto, l'espressione:

$$-1 \leq x \text{ AND } x \leq 1$$

è equivalente all'espressione:

$$(-1 \leq x) \text{ AND } (x \leq 1)$$

come peraltro è naturale, non avendo senso pensare di applicare prima l'operatore AND ad x, se tale simbolo rappresenta una quantità numerica.

Infine, a conclusione del paragrafo, è opportuno notare che, dal punto di vista matematico, l'insieme dei valori di verità $\{\mathbf{falso}, \mathbf{vero}\}$ unitamente agli operatori NOT, AND e OR verificano tutte le proprietà che carattertizzano una struttura algebrica detta *Algebra di Boole*. Per questo motivo i valori di verità e gli operatori logici vengono spesso indicati con il termine *booleano*.

Esercizi proposti

1. Si riscrivano le seguenti espressioni indicando tutte le precedenze:

$$\text{NOT } a \text{ OR NOT } b \text{ AND NOT } c \text{ AND } b \text{ AND NOT } a \text{ OR } b$$

$$a \text{ AND } b \text{ OR } c \text{ OR NOT } a \text{ AND } c$$

2. Scrivere la proprietà duale della seguente:

$$a \text{ OR NOT } a \text{ AND } b \equiv a \text{ OR } b$$

3. Usando operatori di relazione e operatori logici si scriva la condizione che esprime che il valore della variabile x appartiene all'insieme:

$$[-5, 3] \cup [8, 15]$$

4. Esprimere la condizione opposta a quella dell'esercizio 3.

5. Nel caso si sia usato l'operatore NOT nel risolvere l'esercizio 4, formulare una soluzione alternativa senza usare l'operatore NOT.

2.5 Equivalenza di condizioni composte

In generale, nella formulazione di un algoritmo occorre esprimere correttamente le condizioni che controllano i passi di tipo decisionale. Spesso però non esiste un solo modo per esprimere tali condizioni e si pone pertanto il problema di stabilire se condizioni apparentemente diverse (cioè formulate in modo diverso) siano o meno equivalenti, e cioè se assumono lo stesso valore di verità in tutte le situazioni.

Per spiegare la cosa con un esempio, torniamo all'algoritmo seguito dall'impiegato di un ufficio postale nell'accettare conti corrente da parte degli utenti in coda. Abbiamo già visto che tale algoritmo potrebbe contenere la condizione:

l'utente ha già consegnato meno di 5 conti corrente o non ci sono altri utenti in coda

Tuttavia, nella formulazione dell'algoritmo, potrebbe anche venire in mente di esprimere la stessa condizione nel modo seguente:

non accade che l'utente ha già consegnato almeno 5 conti corrente e ci sono altri utenti in coda

Confrontando le due condizioni che determinano il comportamento prescritto all'impiegato, non è immediato stabilire se esse siano equivalenti. Si noti che le condizioni atomiche che le compongono sono le stesse o sono in stretta relazione (entrambe contengono condizioni atomiche identiche, come ad esempio *ci sono altri utenti in coda*, o condizioni opposte come ad esempio *l'utente ha già consegnato meno di 5 conti corrente* e *l'utente ha già consegnato almeno 5 conti corrente*), ma tali condizioni sono composte con operatori diversi.

Per risolvere il problema di stabilire l'equivalenza tra le due condizioni, occorre procedere nel seguente modo:

1. Si esprimono le due condizioni attraverso le stesse condizioni atomiche, opportunamente composte con gli operatori logici. Ad esempio, la prima condizione contiene i predicati elementari:

   A: l'utente ha già consegnato meno di 5 conti corrente
   B: ci sono altri utenti in coda

   e da un'analisi della condizione si ricava che in essa le condizioni A e B sono unite dagli operatori logici OR e NOT in modo da formare l'espressione:

   $$A \text{ OR } (\text{NOT } B) \tag{2.1}$$

   dove le parentesi non sono necessarie. Analogamente, la seconda condizione contiene le condizioni atomiche:

   A': l'utente ha già consegnato almeno 5 conti corrente
   B: ci sono altri utenti in coda

   e tali condizioni sono unite dall'operatore logico NOT e AND in modo da formare l'espressione:

   $$\text{NOT } (A' \text{ AND } B)$$

   dove, questa volta, le parentesi sono necessarie. Infine, ricordando l'osservazione già fatta che il predicato elementare A' è il contrario del predicato elementare A, la seconda condizione si può scrivere:

   $$\text{NOT } ((\text{NOT } A) \text{ AND } B) \tag{2.2}$$

2. Si costruiscono le tabelle di verità delle due formulazioni (2.1) e (2.2). Per far questo, analogamente a quanto fatto nell'esempio 2.1, si costruiscono le tabelle di verità delle sotto espressioni seguendo l'ordine di applicazione degli operatori. Ad esempio, nel caso della (2.2) si costruisce la tabella:

A	B	NOT A	(NOT A) AND B	NOT ((NOT A) AND B)
falso	**falso**	**vero**	**falso**	**vero**
falso	**vero**	**vero**	**vero**	**falso**
vero	**falso**	**falso**	**falso**	**vero**
vero	**vero**	**falso**	**falso**	**vero**

dove la terza colonna si è ottenuta applicando la tabella dell'operatore NOT ad A (prima colonna), la quarta colonna si è ottenuta applicando la tabella dell'operatore AND alla terza colonna e a B (seconda colonna), e l'ultima colonna si è ottenuta applicando nuovamente la tabella dell'operatore NOT alla quarta colonna. Ripetendo il procedimeto per la (2.1) si ricava la tabella:

A	B	NOT B	A OR (NOT B)
falso	**falso**	**vero**	**vero**
falso	**vero**	**falso**	**falso**
vero	**falso**	**vero**	**vero**
vero	**vero**	**falso**	**vero**

3. Si confrontano le due tabelle di verità: se esse coincidono le due formulazioni risultano equivalenti, altrimenti esse esprimono condizioni diverse. Nel nostro caso risulta che:

$$A \text{ OR } (\text{NOT } B) = \text{NOT } ((\text{NOT } A) \text{ AND } B)$$

Esercizi proposti

1. Verificare se le due condizioni:
   (a) *sono stati superati almeno 10 esami e la media è almeno del 27 oppure sono stati superati almeno 10 esami e il reddito non supera i 20.000 euro oppure la media è almeno del 27 e il reddito non supera i 20.000 euro*
   (b) *sono stati superati almeno 10 esami, la media è almeno del 27 e il reddito non supera i 20.000 euro*

   sono equivalenti e, in caso negativo, indicare per quali valori delle condizioni atomiche componenti le due condizioni risultano diverse.

2. Verificare se le due condizioni:
   (a) NOT $((a-b) > 2$ OR $(b-a) > 2)$
   (b) NOT $((a-b) \leq 2$ AND $(b-a) \leq 2)$

   sono equivalenti.

3. Dato il passo decisionale:

   domani andiamo in gita se abbiamo terminato il lavoro o è una bella giornata

   esprimere un passo equivalente senza usare la parola "o" nella condizione (*suggerimento: usare i teoremi di De Morgan*).

4. Riscrivendo in linguaggio naturale le condizioni atomiche contenute nella risposta all'esercizio 3, esprimere la condizione senza usare la parola "non".

5. Dimostrare l'equivalenza tra le due formulazioni (2.1) e (2.2) utilizzando i teoremi di De Morgan e le proprietà notevoli del paragrafo 2.4.

2.6 Impiego di condizioni nei passi decisionali

Gli operatori di relazione e gli operatori logici ci consentono di esprimere un'ampia gamma di condizioni. Per approfondire l'impiego delle condizioni nei passi decisionali di un algoritmo consideriamo il seguente esempio.

Esempio 2.2

Assumendo che a, b e c siano le misure dei tre lati di un triangolo, vogliamo scrivere le condizioni che identificano se il triangolo è equilatero, isoscele o scaleno.

Il triangolo è equilatero se[1]:

$$a = b \text{ AND } b = c \text{ AND } a = c$$

Il triangolo è isoscele se[2]:

$$(a = b \text{ AND } a \neq c) \text{ OR } (b = c \text{ AND } b \neq a) \text{ OR } (a = c \text{ AND } a \neq b)$$

Infine, il triangolo è scaleno se:

$$a \neq b \text{ AND } a \neq c \text{ AND } b \neq c$$

Un algoritmo per un esecutore che scriva il risultato della verifica potrebe dunque essere il seguente:

1. se $a = b$ AND $b = c$ AND $a = c$, scrivi equilatero
2. se $(a = b$ AND $a \neq c)$ OR $(b = c$ AND $a \neq c)$ OR $(a = b$ AND $b \neq c)$, scrivi isoscele
3. se $a \neq b$ AND $a \neq c$ AND $b \neq c$, scrivi scaleno

dove l'esecutore esegue sempre tutti e tre i passi (cioè la sequenza dinamica coincide sempre con la sequenza statica), ma stampa sempre una sola tra le tre possibili scritte, a causa della natura mutuamente esclusiva delle condizioni impiegate.

Una formulazione alternativa dell'algoritmo potrebbe essere la seguente:

1. se $a = b$ AND $b = c$ AND $a = c$, scrivi equilatero **altrimenti**
2. se $a = b$ OR $b = c$ OR $a = c$, scrivi isoscele **altrimenti**
3. scrivi scaleno

dove la parola **altrimenti** significa che, se la condizione è vera, l'esecutore, dopo aver scritto il risultato, deve terminare l'esecuzione dell'algoritmo senza considerare i passi successivi.

Questa nuova formulazione risulta più sintetica e, una volta che ne sia stato compreso il significato, più semplice. Essa, infatti, sfrutta il fatto di avere escluso al passo 1 che il triangolo sia equilatero per semplificare, al passo 2, la condizione che determina se il triangolo e isoscele; e sfrutta il fatto di aver escluso al passo 2 che il triangolo sia isoscele per concludere, senza ulteriori controlli al passo 3, che è scaleno.

Al contrario, la prima formulazione dell'algoritmo, imponendo sempre l'esecuzione di tutti e tre i passi, deve usare condizioni mutuamente esclusive.

Osservazione 2.4

Se invece di determinare prima se il triangolo è equilatero poi se è isoscele e infine se è scaleno, si scegliesse di procedere al contrario, anche usando la clausola **altrimenti**, l'algoritmo sarebbe:

[1] In realtà, per la proprietà transitiva dell'uguaglianza e le proprietà dell'algebra di Boole, l'ultima clausola è inutile.

[2] In realtà per le regole di precedenza le parentesi non sono necessarie, ma è conveniente metterle per facilitare la lettura dell'espressione.

1. se $a \neq b$ AND $a \neq c$ AND $b \neq c$, scrivi scaleno **altrimenti**
2. se $(a = b$ AND $a \neq c)$ OR $(b = c$ AND $a \neq c)$ OR $(a = b$ AND $b \neq c)$, scrivi isoscele **altrimenti**
3. scrivi equilatero

Si noti che in questo caso la condizione al passo 2 non si è semplificata, in quanto l'aver escluso che il triangolo sia scaleno non dà alcuna informazione utile a semplificare le condizioni successive. Si noti che questa osservazione mette in risalto che l'uso della clausola **altrimenti**, se da un lato rende più sintetica e naturale la formulazione, dall'altro richiede una maggiore attenzione sulla scelta dell'ordine con cui vengono prese le decisioni.

Esercizi proposti

1. Se x e y corrispondono rispettivamente al prezzo di un bene in vendita e all'importo offerto da un acquirente, formulare un algoritmo per un esecutore che scriva se il bene può effettivamente essere comprato e se occorre dare un resto all'acquirente, oppure se il bene non può essere comprato. Non usare la parola altrimenti nel formulare l'algoritmo.
2. Risolvere l'esercizio 1 usando la parola altrimenti nel formulare l'algoritmo.

Capitolo 3

Informazioni (bozze, v. 2.0)

Nel capitolo 1, parlando dei dati di ingresso e dei dati di uscita di un problema, abbiamo usato senza definirlo il termine *informazione*, affermando che un algoritmo parte da alcune informazioni iniziali per produrre le informazioni finali e che pertanto un algoritmo *trasforma* o *elabora* informazioni.

Questo capitolo è pertant dedicato a chiarire la natura delle informazioni e come tali informazioni possono essere trasformate da un algoritmo. In particolare ci occuperemo del problema fondamentale in Informatica di rappresentare le informazioni in modo da renderne possibile la loro manipolazione da parte di un esecutore automatico.

3.1 Nozione di informazione

Trattandosi di un concetto primitivo, la nozione di informazione non può essere definita. Tuttavia è possibile fornire degli esempi di informazioni, ed attraverso di essi cercare di illustrarne le proprietà di interesse per lo studio dei Fondamenti dell'Informatica.

Nella figura 3.1 l'omino a sinistra chiede all'altro: "mi dici il numero di Andrea?" e ottiene come risposta la sequenza di cifre : "0817651831". Tale sequenza è quello che intuitivamente chiamamiamo *informazione*.

Tuttavia, ad un esame più attento, si può osservare che la sequenza 0817651831 da sola non è un'informazione completa. Ciò che rende tale sequenza di interesse è sapere che essa è il numero (sottinteso di telefono) di Andrea. Possiamo pertanto affermare che è un'informazione l'intera frase:

Il numero di telefono di Andrea è 0817651831

Questa osservazione permette di mettere in evidenza che nel concetto di informazione sono presenti due elementi:

- il *valore* dell'informazione, che nel caso specifico è la sequenza di cifre 0817651831;
- l'*attributo* dell'informazione, che nel caso specifico è la frase *il numero di telefono di Andrea*.

Entrambi gli elementi sono essenziali perchè si abbia un'informazione. È infatti immediato comprendere che la sola sequenza di cifre non è un'informazione, perchè essa potrebbe essere qualunque cosa: la matricola di un'arma o di un automobile, il numero ottocentodiciassettemilioniseicentocinquantunomilaottocentotrentuno, o anche soltano il numero di telefono di qualcuno che non è Andrea! D'altra parte è del tutto evidente che, senza il valore, l'attributo non fornisce alcuna informazione.

Per chiarire ulteriormente la nozione di informazione e dei suoi elementi costitutivi, valore e attributo, si consideri il seguente valore isolato:

Roma

Figura 3.1: Un esempio di informazione.

Cosa significa tale valore? Che informazione rappresenta? Perchè il valore “Roma” si trasformi in un’informazione è necessario associarlo ad un attributo. Ad esempio “Roma è la capitale d’Italia”, oppure “Roma è la città dove è nato Piero”, oppure semplicemente “Roma è una città”.

Osservazione 3.1

Ancora qualche commento sugli elementi costituenti un’informazione. L’attributo è il *siginificato* dell’informazione, specifica cioè di quale informazione si tratta. Tuttavia, sebbene l’attributo sia necessario per avere un’informazione, esso è spesso implicito, a motivo del fatto che normalmente la conoscenza del valore, unitamente al contesto in cui si sta operando, consente di ricavare l’attributo. Ad esempio, nella figura 3.1, l’omino che risponde fornisce solo la sequenza di cifre, dando per scontato che si tratta del numero di telefono di Andrea.

Non è pertanto raro che in informatica si manipolino i valori delle informazioni prescindendo apparentemente dai loro attributi. Occorre però ricordare che a ciascun valore dovrebbe sempre essere associato un attributo e che qualunque manipolazione di informazioni ha la sua ragion d’essere proprio negli attributi dei valori manipolati.

Ad esempio, se dopo aver fatto la spesa in più negozi si vuole calcolare quanto si è speso, si effettuerà la somma degli importi riportati sugli scontrini accumulati. Naturalemente, nel fare la somma (cioè nell’eseguire un algoritmo che somma una sequenza di numeri) si considereranno gli importi (cioè i valori), prescindendo da ciò che si è effettivamente comprato (gli attributi associati a quei valori), ma ciò non toglie che il risultato finale ha senso (è la cifra complessivamente spesa), proprio perchè i singoli addendi erano il costo di ciò che si è comprato.

Non ha infatti senso manipolare un valore senza attributo perchè in tal caso anche il risultato della manipolazione sarebbe privo di attributo e pertanto senza siginificato.

3.2 Rappresentazione di un’informazione

Se vogliamo elaborare informazioni attraverso un esecutore automatico è necessario *rappresentare* le informazioni in una forma che possa essere manipolata dall’esecutore. Come abbiamo visto un’informazione è formata da un valore e un attributo. Per quanto osservato alla fine del paragrafo precedente, quasi sempre l’attributo non viene esplicitamente rappresentato nell’esecutore, ma l’associazione valore–attributo rimane implicita nelle intenzioni di chi formula l’algoritmo. Il valore invece deve essere rappresentato esplicitamente all’interno dell’esecutore e l’esecutore elabora le informazioni *trasformando la rappresentazione dei corrispondenti valori*.

Quanto detto risulta evidente nell esempio 1.1, dove i numeri sono rappresentati all'interno dell'esecutore da gruppi di palline e l'elaborazione consiste nello spostare palline da un gruppo all'altro secondo un opportuna strategia (cioè l'algoritmo). Analogamente, nell esempio 1.2 i numeri sono rappresentati all'interno dell'esecutore da sequenze di cifre scritte su una lavagna secondo la rappresentazione posizionale e l'elaborazione consiste in somme sulle singole cifre secondo un'opportuno algoritmo.

Questo significa che un algoritmo formulato per un esecutore automatico consiste in pratica in *operazioni sulla rappresentazione dei valori* delle informazioni da trasformare; le operazioni sono inoltre organizzate sulla base di un'*associazione implicita tra le suddette rappresentazioni e gli attributi delle informazioni da trasformare*.

Ma quali sono i metodi più adatti per rappresentare valori in modo da renderne agevole la manipolazione da parte di un esecutore automatico? Prima di rispondere a questa domanda, è opportuno introdurre la nozione di *stringa* di simboli appartenenti ad un insieme finito.

Definizione 3.1 *Dato un insieme finito R, una qualunque sequenza finita di elementi di R viene detta* stringa di simboli in R.

Una stringa è caratterizzata dalla sua lunghezza, cioè dal numero di simboli che contiene. L'insieme delle stringhe di simboli in R di lunghezza m assegnata coincide con il prodotto cartesiano:

$$R^m = \underbrace{R \times R \times \ldots \times R}_{m}$$

È importante ricordare alcune proprietà dell'insieme R^m. Se $|R| = h$, cioè se h è la cardinalià di R, allora la cardinalità di R^m è:

$$|R^m| = h^m$$

Viceversa, per poter disporre di un insieme di almeno N stringhe di simboli in R, tutte di ugual lunghezza, occorre considerare stringhe di lunghezza:

$$m = \lceil log_h N \rceil$$

dove il simbolo $\lceil\ \rceil$ indica la funzione di argomento reale *ceiling*, che restituisce il primo intero maggiore o uguale all'argomento. Ad esempio si ha:

$$\lceil 4.08 \rceil = 5, \ \lceil 4 \rceil = 4, \ \lceil -4.08 \rceil = -4, \ \lceil -4 \rceil = -4$$

Definizione 3.2 *Si dice* rappresentazione *o* codifica *dei valori appartenenti a un insieme finito D, una funzione iniettiva dall'insieme D, detto* dominio, *a un insieme di stringhe R^m, detto* codominio, *o* codice, *o* insieme dei rappresentanti[1].

Con notazione matematica, una rappresentazione c dei valori appartenenti a D è dunque una funzione:

$$c : D \rightarrow R^m \tag{3.1}$$

dove, per soddisfare la proprietà di iniettività, occorre che sia $|R^m| \geq |D|$ che, per quanto detto in precedenza, implica $m \geq \lceil log_{|R|} |D| \rceil$.

Esempio 3.1

[1]Questa definizione si riferisce esclusivamente a codifiche a lunghezza fissa, in cui cioe' l'insieme dei rappresentanti è formato da stringhe tutte della stessa lunghezza. È possibile anche considerare codifiche a lunghezza variabile in cui i valori del dominio sono associati a striinghe di lunghezza diversa. Sebbene le codifiche a lunghezza variabile siano di interesse in molti casi pratici, per semplificare la trattazione ci limiteremo a cnsiderare in questo capitolo esclusivamente codifiche a lunghezza fissa.

Sulle targhe automobilistiche italiane viene riportata una rappresentazione della provincia di immatricolazione utilizzando un codice di 2 lettere dell'alfabeto italiano, inclusa la lettera K (la sigla di Crotone è KR). In questo caso l'insieme D dei valori da rappresentare è l'insieme delle provicie italiane, mentre l'insieme dei rappresentanti sono le stringhe di lunghezza 2 di simboli apparenenti all'insieme delle lettere maiuscole.

Attualmente vi sono in Italia 103 provincie, pertanto si ha $|D| = 103$. Poichè il numero delle lettere considerate è 22, abbiamo:

$$22^2 = 484 \geq 103$$

per cui abbiamo un numero sufficiente di rappresentanti. Si noti anche che 2 è uguale a $\lceil log_{22} 103 \rceil$, cioè è il numero minimo di lettere che permette di avere un rappresentante distinto per ogni provincia.

Osservazione 3.2

Il numero di coppie di lettere disponibili è sovrabbondante, ma questo non crea particolari problemi: implica semplicemente che solo 103 coppie sulle 484 disponibili sono effettivamente usate come rappresentanti. In linea di principio, sia la scelta delle 103 coppie da usare, sia l'associazione tra le 103 provincie e le 103 coppie scelte sono arbitrarie, nel senso che tutte costituiscono rappresentazioni valide.

Tuttavia considerazioni pratiche possono far preferire associazioni che godano di particolare proprietà tra tutte quelle possibili. Nel caso specifico, la convenienza di avere un'associazione facilmente memorizzabile tra provincie e stringhe rappresentanti (dette normalmente *sigle*) suggerisce di usare una regola di associazione basata sulle iniziali di ciascuna provincia. Poichè tale regola da sola non è sufficiente (molte provincie hanno le stesse iniziali) intervengono scelte più o meno arbitrarie fino a giungere alla seguente associazione (mostrata solo parzialmente):

Provincia	Sigla
Agrigento	AG
Alessandria	AL
Ancona	AN
Aosta	AO
...	...
Viterbo	VT

Osservazione 3.3

Conviene osservare che la funzione che associa ciascun valore (provincia) al suo rappresentante (sigla) è completamente definita dalla tabella appena riportata e che anzi *tale tabella è l'unica descrizione di tale funzione*. In altre parole, la funzione di rappresentazione non è definita da un'espressione, ma da una tabella.

In effetti, una funzione a dominio finito (come è sempre il caso di una codifica) può sempre essere rappresentata in forma tabellare. Tale situazione è molto comune nel caso delle codifiche, quando l'insieme dei valori da rappresentare non ha particolari strutture algebriche (a volte non è neppure ordinato).

3.3 Rappresentazione di informazioni mediante stringhe di bit

Dal punto di vista pratico, risulta di enorme interesse il caso in cui l'insieme dei simboli R è costituito da due soli elementi. Il motivo di tale particolare interesse è che è molto facile realizzare dispositivi fisici di varia natura che possano trovarsi in due diversi stati facilmente riconoscibili[2]. Associando convenzionalmente ai due stati del dispositivo i due elementi di R, è possibile usare gruppi di questi dispositivi (che equivalgono a stringhe di elementi di R) come mezzo concreto per rappresentare i valori che interessa manipolare all'interno di un esecutore automatico.

[2] Si pensi per esempio a una lamapadina, che può essere spenta o accesa, a una superficie, che può essere opaca o riflettente, ad un contatto elettrico che può trovarsi alla tensione di alimentazione o a massa, ecc.

Anche se quanto detto non dipende dalla natura dell'insieme R, è consuetudine indicare i suoi due elementi con i simboli 0 e 1. Formalmente, abbiamo:

$$R = \{0, 1\}.$$

I simboli 0 e 1 vengono detti *bit*, che è un acronimo derivato dal termine inglese *bi*nary digi*t* (cifra binaria).

Un qualunque insieme di valori D può pertanto essere rappresentato da stringhe di bit di lunghezza almeno pari a:

$$n = \lceil log_2 |D| \rceil. \tag{3.2}$$

Esempio 3.2

Volendo rappresentare i valori dell'insieme dei semi delle carte da gioco francesi:

$$D = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$$

sono sufficienti $\lceil log_2 |D| \rceil = \lceil log_2 4 \rceil = 2$ bit per rappresentare i 4 elementi dell'insieme. La rappresentazione può essere definita tramite la seguente tabella dove l'associazione tra gli elementi di D e le stringhe di 2 bit è arbitraria.

seme	rappresentante
♣	00
♢	01
♡	10
♠	11

Esempio 3.3

Un secondo esempio è la codifica dei giorni della settimana. In questo caso l'insieme da codificare è:

$$D = \{ \text{ lunedì, martedì, mercoledì, giovedì, venerdì, sabato, domenica } \}$$

e il numero di bit dai rappresentanti è $\lceil log_2 7 \rceil = 3$.

Anche in questo caso la rappresentazione può essere definita mediante una tabella:

giorno	rappresentante
lunedì	111
martedì	001
mercoledì	110
giovedì	000
venerdì	101
sabato	100
domanica	010

dove è stata arbitrariamente esclusa la stringa 011 (le stringe di 3 bit sono in totale $2^3 = 8$) e dove l'associazione tra gli elementi di D e le riamananti 7 stringhe di 3 bit è arbitraria.

Osservazione 3.4

Un'ultima osservazione da fare sul concetto di rappresentazione è che un rappresentante *non ha un siginificato di per sè*. Una stringa di bit può infatti rappresentare qualunque cosa come appare evidente dalla seguente tabella che definisce una rappresentazione per i primi 8 numeri naturali:

numero	rappresentante
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

la stringa di bit 110 rappresenta il mercoledì se si sta usando la rappresentazione per i giorni della settimana definita in precedenza, mentre rappresenta il numero naturale 6 se si sta usando la rappresentazione appena definita per i primi 8 numeri naturali.

Un rappresentante pertanto ha siginificato, cioè rappresenta un valore, se ad esso è associata una rappresentazione, cioè una tabella che stabilisce una corrispondenza tra un particolare insieme di valori e stringhe di bit.

Spesso, nella pratica, tale corrispondenza è implicita, nel senso che risulta evidente dal contesto in cui la stringa di bit è usata. È importante osservare però che essa deve essere sempre in qualche modo presente, e che qualunque manipolazione su rappresentanti di informazioni viene sempre effettuata assumendo una precisa rappresentazione. In caso contrario, le manipolazioni perdono di siginificato perchè non hanno siginificato le stringhe di bit manipolate.

Sotto questo aspetto, il rapporto tra rappresentante e codifica è analogo a quello tra valore e attributo

Esercizi proposti

1. Definire le 24 rappresentazioni diverse possibili dei semi delle carte usando stringhe di 2 bit.
2. Definire 3 rappresentazioni diverse dei giorni della settimana usando stringhe di 3 bit.
3. Le rappresentazioni diverse di 4 valori mediante stringhe di 2 bit sono 24 (cfr. esercizio 1). Spiegarne la ragione e calcolare il numero di rappresentazioni diverse dei giorni della settimana mediane stringhe di 3 bit.

3.4 Tipo di un'informazione

Per quanto detto fino ad ora, le informazioni manipolate da un esecutore sono caratterizzate da un valore, un attributo e una rappresentazione. Anche se l'informazione vera e propria consiste nei primi due elementi, la rappresentazione è ciò che è effettivamente presente nell'esecutore e che viene manipolato. In altre parole, se si vuole elaborare un'informazione occorre associare ad essa una rappresentazione.

Ora, una rappresetazione dipende dall'insieme di valori che si devono rappresentare, e dunque per associare una rappresentazione a un'informazione occorre stabilire quale è l'insieme dei valori che l'informazione può assumere. Ad esempio, volendo rappresentare l'informazione:

Piero ha 17 anni

occorre stabilire un'intervallo di valori possibili per l'età, perchè per rappresentare il valore 17 è necessario avere una corrispondenza tra numeri e stringhe di bit e tale corrispondenza dipende da quali numeri si vogliono considerare.

Nel caso specifico potrebbe essere ragionevole assumere che l'età di una persona possa assumere un qualunque valore tra 0 e un massimo di 120 e scegliere quindi come rappresentanti stringhe di 7 bit ($\lceil log_2 121 \rceil = 7$). A questo punto una qualsiasi tabella, che rappresenti una funzione iniettiva tra l'intervallo dei numeri interi $[0, 120]$ e le stringhe di 7 bit, può essere usata come rappresentazione. Naturalmente, una volta operata tale scelta, se dovessimo trattare l'informazione relativa all'età di Piero nel caso Piero abbia più di 120 anni, non potremmo farlo senza *cambiare la rappresentazione associata all'informazione "età di Piero"*.

Proseguendo l'esempio, se decidessimo di rappresentare le età con una rappresentazione posizionale (vedi pargrafi successivi) su 7 bit, l'informazione che Piero ha 17 anni sarebbe rappresentata dalla stringa:

0010001

Si noti che, come già osservato, la stringa 0010001 *non rappresenta di per sè che Piero ha 17 anni*. È possibile attribuire alla suddetta stringa tale siginificato solo perchè:

1. abbiamo stabilito di usare come rappresentazione dell'informazione la rappresentazione binaria naturale su 7 bit che associa alla stringa il numero 17;
2. il contesto dell'esempio ci permette di associare al valore 17 così ottenuto l'attributo "età di Piero".

Non è inutile sottolineare ancora una volta, che all'interno di un esecutore che dovesse elaborare tale informazione sarebbe presente fisicamente solo il rappresentante del valore dell'informazione (nel caso considerato la stringa 0010001). L'associazione con la rappresentazione e con l'attributo è normalmente presente solo *nelle intenzioni di chi programma l'algoritmo*.

In definitiva, nel contesto della formulazione di un algoritmo per un esecutore automatico, ogni informazione di interesse è associata, oltre al valore e all'attributo, a un *insieme di valori possibili* e ad una *rappresentazione*. Siamo quindi in grado di dare la seguente definizione.

Definizione 3.3 *Il* tipo *di un'informazione manipolata da un esecutore che esegue un algoritmo è una coppia di elementi:*

1. *l'insieme di valori che l'informazione può assumere;*
2. *la rappresentazione utilizzata per rappresentare tali valori durante l'esecuzione dell'algoritmo da parte dell'esecutore.*

Osservazione 3.5

Prima di fare qualche ulteriore commento, vale la pena osservare che in un contesto più avanzato, la definizione di tipo include generalmente anche un terzo elemento: l'insieme di possibili manipolazioni (dette tecnicamente *operazioni*) che possono essere applicate ai valori che un informazione può assumere. Tuttavia nel seguito trascureremo tale estensione, poiché essa da un lato non modifica in modo radicale il concetto di tipo, e dall'altro non trova applicazione in una trattazione di natura introdutiva.

Esempio 3.4

Per esemplificare il concetto di tipo di un'informazione si consideri le informazioni riportate nella seguente tabella:

attributo	valore	tipo
giorno del mese	13	numero naturale
temperatura (gradi Celsius)	20.74	numero reale
luogo di nascita	Napoli	stringa di 6 caratteri

Nella tabella il tipo di ciascuna informazione è indicato con una breve frase e, secondo la definizione data, tale frase corrisponde a un insieme di valori e a una funzione tra tale insieme e stringhe di bit. Dobbiamo pertanto chiarire cosa intendiamo in questo contesto per "numero naturale", "numero reale" e "stringa di 6 caratteri". In altre parole dobbiamo definire i *tipi* associati alle tre informazioni.

Per *numero naturale* intendiamo un numero naturale nell'intervallo $[0, 65535]$ rappresentato in binario naturale su 16 bit. Si noti che ovviamente l'insieme numerico è molto più ampio di quello necessario a rappresentare il giorno del

mese, ma questo siginifica solo che di fatto l'informazione assumerà solo alcuni dei valori possibili (quelli compresi tra 1 e 31).

Per *numero reale* intendiamo un numero appartenente a un insieme finito di numeri reali opportunamente scelti, rappresentati secondo una rappresentazione standard denominata IEEE 754, che verrà introdotta nel paragrafo 3.12 e che richiede stringhe di 32 bit.

Per *stringa di 6 caratteri* intendiamo l'insieme delle sequenza di 6 simboli scelti tra lettere, cifre e alcuni caratteri speciali, ciascun simbolo rappresentato mediante il codice ASCII che verrà introdotto nel paragrafo 3.9.

Anche se non abbiamo ancora gli strumenti per "calcolare" le stringhe di bit associate ai tre valori riportati nella tabella precedente, per evidenziare che le tre rappresentazioni non sono altro che un modo per associare univocamente a ciascun valore una stringa di bit, mostriamo nella seguente tabella i tre rappresentanti utilizzati da un ipotetico esecutore automatico.

informazione	stringa di bit
il è giorno del mese 13	0000000000001101
la temperatura in gradi Celsiusè 20.74	01000001101001011110101110000101
il luogo di nascita è Napoli	01001110011000010111000001101111011010001101001

3.5 Rappresentazione di valori logici

In questo paragrafo e nei successivi illustreremo i metodi di rappresentazione mediante stringhe di bit di particolari insiemi di valori che svolgono un ruolo privilegiato nella programmazione di algoritmi.

Cominciamo dall'insieme più semplice, ma non per questo meno importante, come abbiamo visto nel capitolo 2: l'insieme dei valori logici, detti anche valori di verità.

Tale insieme è costituito da solo due elementi che rappresentano, rispettivamente, il valore *falso* e il valore *vero*. E' consuetudine rappresentare questi due valori con la corrispondenti parole inglesi, **false** per il valore falso, e **true** per il valore vero, dove l'impiego del neretto è motivato dalla volontà di mettere in evidenza che si tratta degli elementi dell'insieme dei valori di verità.

È evidente che l'insieme:

$$\{\mathbf{false}, \mathbf{true}\}$$

si può rappresentare semplicemente con stringhe di un solo bit. Tuttavia, poichè gli esecutori automatici manipolano generalmente stringhe di bit di lunghezza multipla di una quantità base fissata (solitamente 8), la rappresentazione dei valori di verità impiega stringhe di lunghezza solitamente pari a quella usata per rappresentare i numeri interi (cfr. paragrafi 3.6 e 3.10, con la convenzione che la stringa formata da tutti zero rappresenta il valore **false** e una stringa con uno o più bit diversi da zero rappresenta il valore **true**. La scelta più diffusa è pertanto la seguente (si è fatta l'ipotesi, peraltro realistica che gli interi siano rappresentati con 32 bit):

valore logico	rappresentante
false	00000000000000000000000000000000
true	00000000000000000000000000000001

che affida al bit più a destra il compito di discriminare tra i due valori.

Osservazione 3.6

In alcuni linguaggi (e dunque nei corrispondenti esecutori) qualunque stringa di bit che non sia quella composta da tutti zero rappresenta validamente il valore **true**. Un tale modo di procedere è di fatto una violazione del principio che una rappresentazione è una funzione dall'insieme dei valori da rappresentare all'insieme delle stringhe rappresentanti, perchè ad un valore non corrisponderebbe più un unico rappresentante. Tuttavia la cosa offre dei vantaggi dal punto di vista realizzativo senza particolari complicazioni aggiuntive perchè, da un lato dovendo rappresentare il valore **true**

l'esecutore può usare una qualunque stringa che contenga almeno un 1 (tipicamente quella usata nella tabella riportata in precedenza), dall'altro volendo passare da una stringa al valore logico rappresentato è sufficiente verificare se i bit sono tutti zero (valore logico **false**) o no (valore logico **true**).

3.6 Rappresentazione dei numeri naturali

Sebbene l'introduzione della nozione di rappresentazione dovrebbe aver chiarito la distinzione tra valore di un'informazione e suo rappresentante, iniziando a discutere il problema della rappresentazione degli insiemi numerici mediante stringhe di bit può essere opportuno sottolineare ancora una volta la differenza tra *numero* e *rappresentazione di un numero*.

Ad esempio, il numero quindici può essere rappresentato in molti modi tra cui i seguenti:

15 XV 1111 F

che corrispondono, rispettivamente, alla rappresentazione decimale, a quella romana, a quella binaria e a quella esadecimale. In tutti i casi il valore rappresentato è sempre *il numero quindici*. Di passaggio si noti che il terzo rappresentante (1111) rappresenta il numero quindici perchè abbiamo dicharato esplicitamente che si tratta di una rappresentazione binaria, perchè altrimenti, secondo la rappresentazione decimale che per abitudine tendiamo implicitamente ad assumere quando abbiamo a che fare con cifre, la sequenza di cifre 1111 sarebbe il rappresentante del numero millecentoundici.

Tornando al problema di determinare una rappresentazione dei numeri mediante stringhe di bit, cominciamo a occuparci dell'insieme numerico più semplice: i numeri naturali.

La rappresentazione più usata per i numeri naturali è basata sulla possibilità di esprimere un qualunque numero naturale n sotto forma di *sommatoria di potenze*. Dato infatti un numero naturale n e un numero intero positivo b, detta base di rappresentazione, è sempre possibile scrivere la seguente uguaglianza:

$$n = \sum_{i=0}^{\infty} c_i \times b^i \tag{3.3}$$

dove i *coefficienti* c_i sono numeri naturali tali che:

$$\begin{array}{ll} 0 \leq c_i \leq b-1 & \text{per } 0 \leq i < \lfloor log_b n \rfloor \\ 1 \leq c_i \leq b-1 & \text{per } i = \lfloor log_b n \rfloor \\ c_i = 0 & \text{per } i > \lfloor log_b n \rfloor \end{array}$$

dove il simbolo $\lfloor\ \rfloor$ indica la funzione *parte intera* di argomento reale che restituisce il più grande intero minore o uguale dell'argomento (la fuzione parte intera viene anche detta *di troncamento*, o, con temine più informatico, *floor*). In queste ipotesi è anche possibile dimostrare che i coefficienti c_i sono unici.

L'equazione (3.3) è alla base della rappresentazione decimale dei numeri naturali che siamo abituati a usare. Scegliendo $b = 10$, la (3.3) associa in modo univoco $\lfloor log_{10} n \rfloor + 1$ numeri naturali compresi tra 0 e 9 a ogni numero naturale n. La rappresentazione decimale si ottiene quindi usando le *cifre decimali* per rappresentare tali numeri (tale rappresentazione, anche se ovvia, viene riportata nella tabella 3.1 per completezza), adottando la convenzione di elencare le cifre da sinistra verso destra, partendo da quella associata alla potenza $10^{\lfloor log_{10} n \rfloor}$, e non rappresentando esplicitamente le cifre (nulle) associate alle potenze maggiori.

Esempio 3.5

In notazione decimale il numero centoventicinque viene rappresetato da $\lfloor \log_{10} 125 \rfloor + 1 = 3$ cifre associate rispettivamente ai numeri uno, due e cinque. Tali coefficienti della somma di potenze moltiplicano rispettivamente le potenze 10^2 ($\lfloor \log_{10} 125 \rfloor = 2$), 10^1 e 10^0.

numero	rappresentante (cifra)
zero	0
uno	1
due	2
tre	3
quattro	4
cinque	5
sei	6
sette	7
otto	8
nove	9

Tabella 3.1: Rappresentazione dei primi dieci numeri naturali mediante le 10 cifre decimali.

Analogamente al caso $b = 10$, l'equazione (3.3) può essere usata per definire un'associazione tra numeri naturali e stringhe di bit. L'associazione si ottiene scegliendo $b = 2$, associando i coefficienti della sommatoria (che possono assumere solo i valori 0 e 1) alle cifre binarie secondo la ovvia tabella:

numero	rappresentante (cifra)
zero	0
uno	1

e adottando le stesse convenzioni di scrittura usate per la rappresentazione decimale.

Esempio 3.6

Usando 2 come base, il numero centoventicinque viene rappresetato da $\lfloor \log_2 125 \rfloor + 1 = 7$ cifre associate rispettivamente ai numeri uno, uno, uno, uno, uno, zero, uno. Tali coefficienti della somma di potenze moltiplicano rispettivamente le potenze 2^6 ($\lfloor \log_2 125 \rfloor = 6$), 2^5, 2^4, 2^3, 2^2, 2^1 e 2^0.

Quanto detto trova immediata applicazione nella rappresentazione, all'interno di un esecutore, di informazioni che possono assumere come valore un numero naturale. A tal fine dobbiamo infatti definire un *tipo "numero naturale"* che faccia riferimento a un insieme finito di valori possibili e a un'associazione tra questi valori e stringhe di bit di lunghezza assegnata.

Come insieme di valori si sceglie un intervallo di N numeri naturali del tipo $[0, N-1]$, e come rappresentazione si sceglie quella che associa ad ogni numero di tale intervallo la stringa di bit coincidente con i primi $\lceil \log_2 N \rceil$ coefficienti definiti dalla (3.3) quando $b = 2$. Si noti che questo significa che, a differenza della convenzione abituale, dato un numero n appartenente all'intervallo, anche eventuali cifre nulle iniziali fino a quella associata alla potenza $2^{\lceil \log_2 N \rceil - 1}$ vengono comunque rappresentate esplicitamente. La condizione perché siano presenti tali cifre nulle iniziali è che n verifichi la disuglaglianza: $\lfloor \log_2 n \rfloor < \lceil \log_2 N \rceil - 1$. La rappresentazione descritta prende il nome di rappresentazione *posizionale* o *binaria naturale*.

Esempio 3.7

Ad esempio, se $N = 10$, la rappresentazione dei primi 10 numeri naturali che si ottiene usando il tipo numero naturale è quello riportato nella seguente tabella:

numero	rappresentante
zero	0000
uno	0001
due	0010
tre	0011
quattro	0100
cinque	0101
sei	0110
sette	0111
otto	1000
nove	1001

dove i numeri fino a sette incluso hanno almeno una cifra nulla iniziale ($\lfloor \log_2 7 \rfloor = 2 < 3 = \lceil \log_2 10 \rceil - 1$).

Osservazione 3.7

È molto importante notare che nella rappresentazione posizionale, oltre che in forma tabellare, l'associazione può essere definita anche attraverso la (3.3), unitamente alle convenzioni di rappresentazione sopra descritte. Anzi, in questo caso la forma tabellare è stata ricavata proprio dalla rappresentazione dei numeri naturali mediante somma di potenze.

Osservazione 3.8

Poichè con $\lceil \log_2 N \rceil$ bit si possono rappresentare in ogni caso $2^{\lceil \log_2 N \rceil}$ numeri diversi, nel definire il tipo numero naturale si sceglie normalmente un valore di N che sia potenza di 2 in modo che, fissato il numero di bit, l'intervallo di numeri rappresentati sia il più ampio possibile. A esempio, invece di definire una rappresentazione per l'intervallo di 10 numeri $[0, 9]$ è più conveniente farlo per l'intervallo $[0, 15]$ che contiene $16 = 2^4$ numeri. La rappresentazione corrispondente è riportata nella tabella 3.2.

Da quanto detto, per definire il tipo numero naturale, è sufficiente scegliere il numero di bit h che si intende usare. Il valore di h determina sia l'insieme di valori che viene rappresentato (l'intervallo di numeri naturali $[0, 2^h - 1]$), sia l'associazione tra i numeri e le stringhe di bit basata sulla (3.3). Le proprietà generali di tale equazione assicurano che l'associazione tra i numeri nell'intervallo e le stringhe di h bit sia univoca e che pertanto tale associazione definisca una rappresentazione.

È tuttavia conveniente illustrare in che modo si possa praticamente determinare il rappresentnte di un qualunque numero $n \in [0, 2^h - 1]$ e, viceversa, dato un rappresentante come si possa determinare il numero rappresentato.

Per determinare il rappresentante di un numero si procede nel modo seguente. Si calcola ripetutamente la divisione intera di n per 2 fino a che non si ottiene un quoto nullo. I resti delle divisioni ripetute formano una sequenza di valori che possono assumere solo i valori zero e uno. La rappresentazione di n si ottiene scrivendo da destra verso sinistra le cifre associate a tali valori partendo dal primo resto calcolato e completando eventualmente la sequenza di bit così ottenuta con cifre nulle a sinistra fino a raggiungere h bit complessivi.

Esempio 3.8

Se vogliamo usare 10 bit per rappresentare i numeri naturali, saremo ingrado di rappresentare tutti i numeri naturali da 0 a $2^{10} = 1023$. Volendo determinare il rappresentante del numero 212, dovremo dividere tale numero per 2 ottenendo come quoto 106 e come resto 0. Dovremo quindi dividere 106 per 2 ottenendo come quoto 53 e come resto ancora 0. Ripetendo l'operazione 8 volte di giunge ad un quoto nullo con resto 1. Il procedimento è riportato nella figura 3.2 in una forma di immediato siginificato. La sequenza di 8 resti, partendo dal primo è:

0 0 1 0 1 0 1 1

numero	rappresentante
zero	0000
uno	0001
due	0010
tre	0011
quattro	0100
cinque	0101
sei	0110
sette	0111
otto	1000
nove	1001
dieci	1010
undici	1011
dodici	1100
tredici	1101
quattordici	1110
quindici	1111

Tabella 3.2: Rappresentazione posizionale dei primi 16 numeri naturali.

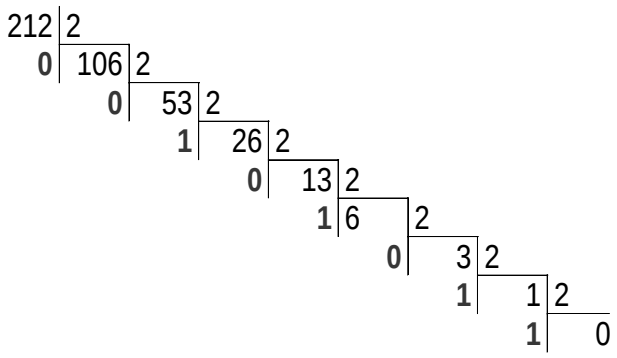


Figura 3.2: Detrerminazione del rappresentante del numero 212.

che scritta da destra a sinistra e completata con due zeri determina come rappresentante di 212 la stringa di bit:

0011010100

Si noti che, naturalmente, un volta deciso di usare h bit per rappresntare i numeri naturali, non è possibile rappresentare numeri maggiori di $2^h - 1$. Ad esempio, applicando la regola appena descritta, il numero 2535 dovrebbe essere rappresentato dalla stringa 100111100111 che ha una lunghezza maggiore di 10 (vedi anche il paragrafo 3.7).

Per sapere invece quale è il valore rappresentato da una particolare stringa di bit basta semplicemente applicare l'equazione 3.3, ricordando che la prima cifra di una stringa di lunghezza h rappresenta il coefficiente della potenza 2^{h-1}, la seconda quello della potenza 2^{h-2}, fino all'ultima cifra che rappresenta il coefficinete della potenza 2^0.
Ad esempio, la stringa:

0011010100

rappresenta ovviamente il numero:

$$\begin{aligned} \mathbf{0} \times 2^9 + \mathbf{0} \times 2^8 + \mathbf{1} \times 2^7 + \mathbf{1} \times 2^6 + \mathbf{0} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{0} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{0} \times 2^0 &= \\ 128 + 64 + 16 + 4 &= 212 \end{aligned}$$

La rappresentazione posizionale ha numerosi vantaggi. Oltre a fornire una regola semplice per associare numeri e stringhe di bit, essa gode di importanti proprietà che semplificano la manipolazione della rappresentazione da parte di un esecutore automatico. In particolare, è possibile definire semplici algoritmi per effettuare tutte le più importanti operazioni tra numeri. Riportiamo nel seguito tali algoritmi per alcune operazioni particolarmente significative.

Somma. La somma si effettua in modo analogo al caso di rappresentazione decimale, facendo uso delle seguenti tabelle che definiscono la somma e il riporto della somma di due cifre e di un riporto.

somma

riporto = 0

	0	1
0	0	1
1	1	0

riporto = 1

	0	1
0	1	0
1	0	1

riporto successivo

riporto = 0

	0	1
0	0	0
1	0	1

riporto = 1

	0	1
0	0	1
1	1	1

Ad esempio, volendo calcolare la somma di due numeri naturali rappresentati su 10 bit dalle stringhe 0000001011 e 0000000111, sommando cifra a cifra e usando le tabelle riportate sopra si ottiene la stringa:

0000010010

Infatti, partendo da destra con riporto iniziale nullo si ottiene:

- riporto 0 e cifre 1 e 1 danno somma 0 e riporto successivo 1
- riporto 1 e cifre 1 e 1 danno somma 1 e riporto successivo 1
- riporto 1 e cifre 0 e 1 danno somma 0 e riporto successivo 1
- riporto 1 e cifre 1 e 0 danno somma 0 e riporto successivo 1

- riporto 1 e cifre 0 e 0 danno somma 1 e riporto successivo 0
- riporto 0 e cifre 0 e 0 danno somma 0 e riporto successivo 0
- i passi successivi sono identici al precedente e danno tutti come somma 0 e come riporto successivo 0.

Si noti che il procedimento fornisce il risultato corretto dal punto di vista aritmetico. Infatti, le stringhe iniziali rappresentavano i numeri 11 e 7 rispettivamente, mentre la stringa ottenuta applicando l'algoritmo rappresenta il numero 18 che è proprio la somma di 11 e 7.

3.7 Overflow

Un'importante osservazione è la seguente. Quando si effettua un'operazione aritmetica sui rappresentanti dei numeri naturali, la stringa risultato può essere più lunga di h bit. Ad esempio, se sommiamo i numeri rappresentati dalle stringhe 1100101101 e 0111100010, applicando l'algoritmo ai rappresentanti, otteniamo la stringa 10100001111 più lunga di 10 bit. In effetti, le stringhe di partenza rappresentano rispettivamente i numeri 813 e 482 e dunque il risultato della loro somma è 1295. Tale numero non è compreso nell'intervallo $[0, 1023]$ e dunque non meraviglia il fatto che esso non possa essere rappresentato con stringhe di 10 bit.

Nel corso della manipolazione di stringhe che rappresentano numeri si può ottenere un risultato che eccede l'insieme di valori rappresentabili secondo le convenzioni adottate. Tale evento viene denominato *overflow*. Come è evidente, il fenomeno dell'overflow è legato alla necessità di rappresentare comunque insiemi numerici *limitati*; esso può verficarsi pertanto per qualsiasi rappresentazione di insiemi numerici e non riguarda solo il caso di rappresentazione posizionale dei numeri naturali.

Quando si verifica un overflow, il risultato richiesto non è rappresentabile e sarebbe logico aspettarsi che il calcolo venga interrotto e l'utente sia informato. Di fatto un esecutore automatico che manipola rappresentazioni di numeri non segue necessariamente questo comportamento "logico", ma, per varie ragioni, può comportarsi in modo diverso. È pertanto importante nello studio dell'esecutore evidenziare quale sia il comportamento dell'esecutore quando si verifica un overflow (tale comportamento può anche dipendere dal tipo numerico considerato). Infatti, nel programmare un algoritmo di calcolo, o si riesce ad evitare il verificarsi di overflow, o, nel caso questo non sia possibile in assoluto, si deve riconoscere il verificarsi di tale evento per prendere le contromisure opportune. Sarebbe infatti errato e pericoloso programmare un algoritmo di calcolo senza considerare questo aspetto perchè, qualora si verificasse un overflow, il risultato dell'algoritmo sarebbe certamente errato *anche se l'algoritmo, sotto il profilo logico, fosse corretto*.

3.8 Cifre esadecimali

L'impiego di stringhe di bit è giustificato dalla semplicità con cui possono essere realizzati dispositivi fisici con due stati stabili. Tuttavia, per un operatore umano, l'uso di stringhe di bit è scomodo perché le sequenze costituite da due simboli non sono facili da leggere per l'occhio umano e perché nella rapprersentazione di informazioni reali è spesso necessario ricorrere a stringhe di una certa lunghezza.

Per rendersi conto di quanto detto si consideri il problema di stabilire se le seguenti stringhe, rappresentanti di numeri reali secondo la rappresentazione descritta nel successivo paragrafo 3.12, sono uguali o no:

$$01000001101001011110101110000101 \qquad 01000001101001011100101110000101$$

Per questo motivo, è molto diffuso nella documentazione tecnica l'impiego di rappresentazioni che usano un numero maggiore di cifre, ma comunque pari a una potenza di 2. Infatti, scegliendo come numero di cifre pari a 2^p, con $p > 1$, si ottengono due vantaggi:

1. i rappresentanti sono più leggibili perchè contengono un numero di simboli diversi maggiore e sono più brevi;

cifra esadecimale	valore numerico	stringa di 4 bit
0	zero	0000
1	uno	0001
2	due	0010
3	tre	0011
4	quattro	0100
5	cinque	0101
6	sei	0110
7	sette	0111
8	otto	1000
9	nove	1001
A	dieci	1010
B	undici	1011
C	dodici	1100
D	tredici	1101
E	quattordici	1110
F	quindici	1111

Tabella 3.3: Cifre esadecimali.

2. è immediato trasfomare una stringa di due simboli in una stringa di 2^p simboli e viceversa.

Per verificare quanto detto, consideriamo il caso $p = 4$ che è il più usato in pratica e corrisponde a usare codici con 16 cifre. In tale ipotesi, come simboli si usano le 10 cifre decimali e le prime 6 lettere dell'alfabeto inglese, associando ciascuna cifra a un numero da zero a quindici, secondo la tabella 3.3 (nella tabella sono impiegate le lettere maiuscole, ma al loro posto possono indifferentemente essere usate le corrispondenti lettere minuscole).

Il codice risultante viene detto *esadecimale* e, la corrispondenza con i primi 16 numeri naturali induce una associazione tra le 16 cifre e le stringhe di 4 bit che rappresentano tali numeri nella rappresentazione posizionale.

È appena il caso di notare che la trasformazione da stringa di cifre esadecimali a stringa di bit è banale: basta sostituire a ciascuna cifra esadecimale la rappresentazione posizionale del numero associato alla cifra (cfr. tabella 3.3).

L'associazione riportata nella tabella 3.3 consente peraltro anche di trasformare facilmente una stringa di bit in una stringa di cifre esadecimali. Infatti, raggruppando i bit della prima delle due stringhe di 32 bit riportate in precedenza in gruppi di 4 si ha:

$$0100\ \ 0001\ \ 1010\ \ 0101\ \ 1110\ \ 1011\ \ 1000\ \ 0101$$

e consultando la tabella si ottiene la stringa di cifre esadecimali:

$$41\mathrm{A}5\mathrm{EB}85$$

Procedendo allo steso modo, per la seconda stringa si ottiene la stringa di cifre esadecimali:

$$41\mathrm{A}5\mathrm{CB}85$$

È evidente come la rappresentazione esadecimale dei due numeri reali sia molto più leggible per un operatore umano e in particolare come sia adesso evidente che si tratta di due stringhe diverse (che quindi rappresentano due numeri reali diversi).

Il procedimento appena seguito può essere usato per qualunque stringa di bit, avendo l'accortezza di aggiungere degli 0 all'inizio della stringa prima di raggrupare i bit a quattro a quattro se la lunghezza della stringa non fosse un multiplo di 4. Ad esempio, la stinga di 10 bit:

$$1001011001$$

viene completata aggiungendo 2 zeri a sinistra e quindi trasformata nella stringa di cifre esadecimali:

$$2\ 5\ 9$$

Osservazione 3.9

Il motivo per cui le cifre si aggiungono a sinistra è che in questo modo, interpretando sia la stringa di bit che quella di cifre esadecimali come una rappresentazione posizionale, il numero rappresentato è lo stesso. Infatti, con riferimento all'ultimo esempio, la stringa di bit rappresenta il numero:

$$\begin{aligned} \mathbf{1} \times 2^9 + \mathbf{0} \times 2^8 + \mathbf{0} \times 2^7 + \mathbf{1} \times 2^6 + \mathbf{0} \times 2^5 + \mathbf{1} \times 2^4 + \mathbf{1} \times 2^3 + \mathbf{0} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 &= \\ 512 + 64 + 16 + 8 + 1 &= 601 \end{aligned}$$

e la, analogamente la stringa di cifre esadecimale rappresenta il numero:

$$\mathbf{2} \times 16^2 + \mathbf{5} \times 16^1 + \mathbf{9} \times 16^0 = 512 + 80 + 9 = 601$$

Osservazione 3.10

Le regole di trasformazione appena date sono rese possibili dal fatto che 16 è una potenza di 2 e regole analoghe varrebbero se invece di stringhe di 16 simboli si volessero usare stringhe di 4, 8, 32 simboli. La scelta di usare cifre esadecimali è però dettata dalle seguenti considerazioni:

- nella struttura fisica dei calcolatori i bit sono sempre raggruppati in gruppi di 8 e quindi, per non avere la complicazione di aggiungere cifre nulle a sinistra prima della trasformazione, si deve scegliere di usare 4, 16 o 256 simboli;
- l'impiego di 16 simboli è il più conveniente perchè è il numero più prossimo a 10 (4 simboli sarebbero troppo pochi e 256 troppi).

Grazie alle proprietà della rappresentazioe esadecimale essa risulta perfettamente intercambiabile con quella binaria, ed è pertanto molto usata per manipolare manualmente la rappresentazione di informazioni all'interno di un esecutore automatico. Occorre però sottolineare che l'impiego della rappresentazione esadecimale è limitato *esclusivamente* alla letteratura e alla documentazione tecnica (cioè nelle manipolazioni in cui è coinvolto un operatore umano) e che all'interno degli esecutori viene sempre usata la corrispondente rappresentazione basata su cifre binarie.

3.9 Rappresentazione di caratteri

Prima di proseguire la presentazione dei metodi utilizzati per rappresentare gli insiemi numerici, introduciamo un metodo di rappresentazione per un altro insieme di valori di particolare importanza: i *caratteri*.

La necessità di elaborare (e quindi di rappresentare) caratteri si presenta in moltissimi contesti applicativi come l'elaborazione di testi, le basi di dati, le interfacce uomo-macchina, ecc. A ben vedere non esiste applicazione, incluse quelle decisamente orientate al calcolo scientifico, che non richieda in una qualche misura l'impiego di caratteri per comunicare con l'utente nelle fasi di acquisizione dei dati di ingresso e nella presentazione dei risultati.

Quando parliamo di caratteri non ci riferiamo solo alle lettere dell'alfabeto, ma intendiamo un insieme di simboli più vasto che comprende almeno:

- le lettere maiuscole e minuscole dell'alfabeto inglese;
- le cifre decimali;

- i caratteri di interpunzione (la virgola, il punto, il punto e virgola, i due punti, il punto esclamativo, il punto interrogativo, ecc.);
- altri caratteri di varia origine (il più, il meno, il trattino allineato in basso, detto anche *underscore*, le parentesi di vario tipo, l'asterisco, ecc.).

Un rapido conto basato su questa grossolana enumerazione dei caratteri di interesse indica che l'insieme dei caratteri ha un numero di elementi di poco inferiore a 100. Dalla 3.2 si ottiene che per rappresentare tale insieme sono necessari 7 bit. Una rappresetazione per i caratteri si traduce pertanto in una tabella che stabilisca una qualunque corrispondenza iniettiva tra l'insieme dei caratteri da rappresentare e le stinghe di 7 bit.

In effetti, poiché esitono numerosissime possibilità di costruire tale tabella tutte in principio equivalenti, per motivi pratici è opportuno individuarne una e accordarsi per usare sempre quella. La scelta è caduta su una particolare rappresentazione stabilita da un ente di standardizzazione americano e denominata American Standard Code Information Interchange (brevemente nel seguito codice ASCII), che è ormai universalmente accettata.

Nella sua versione originale tale rappresentazione prevedeva di rappresentare 128 caratteri attraverso le 128 strighe di 7 bit, aggiungendo al centinaio di caratteri di uso comune alcuni simboli speciali che, pur non essendo dei veri e propri caratteri utilizzabili nella redazione di testi, vengono indicati con il termine *caratteri di controllo* per analogia con i caratteri veri e propri. Ai caratteri di controllo non è associato alcun simbolo grafico, ma vengono convenzionalmente indicati con una sigla di 2 o 3 lettere. I caratteri di controllo sono 32 e servono per scopi particolari come per esempio:

- *CR* indica la fine di una linea di testo;
- *BEL* indica che la macchina deve emettere un suono;
- *TAB* indica il carattere tabulatore (che allinea verticalmente le righe di testo).

I 128 caratteri previsti dal codice ASCII sono associati alle stringhe di 7 bit tenendo conto del valore numerico che corrisponde a tali stringhe se le si interpreta secondo la rappresentazione posizionele (cioè come i coefficienti di una somma di potenze di due). In altre parole i caratteri sono elencati in un particolare ordine e associati a stringe corrispondenti a numeri naturali crescenti, partendo da 0 fino al numero 127. Rinviando per i dettagli alle tabelle 3.4 e 3.5, interessa qui notare che i caratteri sono raggruppati per tipologia secondo il seguente schema (i numeri indicati rappresentano il numero naturale rappresentato dalla stringa associata al carattere):

- da 0 a 31: caratteri di controllo (nela tabella sono indicate le corrispondenti sigle in corsivo);
- 32: lo spazio bianco;
- da 33 a 47: caratteri di interpunzione e caratteri speciali;
- da 48 a 57: cifre decimali da 0 a 9;
- da 58 a 64: caratteri di interpunzione e caratteri speciali;
- da 65 a 90: lettere maiuscole dell'alfabeto inglese in ordine crescente;
- da 91 a 96: caratteri speciali;
- da 97 a 122: lettere minuscole dell'alfabeto inglese in ordine crescente;
- da 123 a 127: caratteri speciali.

Osservazione 3.11

carattere	codice ASCII	equivalente esadecimale	equivalente numerico	carattere	codice ASCII	equivalente esadecimale	equivalente numerico
NUL	00000000	00	0	*spazio*	00100000	20	32
SOH	00000001	01	1	!	00100001	21	33
STX	00000010	02	2	”	00100010	22	34
ETX	00000011	03	3	#	00100011	23	35
EOT	00000100	04	4	$	00100100	24	36
ENQ	00000101	05	5	%	00100101	25	37
ACK	00000110	06	6	&	00100110	26	38
BEL	00000111	07	7	’	00100111	27	39
BS	00001000	08	8	(	00101000	28	40
TAB	00001001	09	9	)	00101001	29	41
LF	00001010	0A	10	*	00101010	2A	42
VT	00001011	0B	11	+	00101011	2B	43
FF	00001100	0C	12	,	00101100	2C	44
CR	00001101	0D	13	-	00101101	2D	45
SO	00001110	0E	14	.	00101110	2E	46
SI	00001111	0F	15	/	00101111	2F	47
DLE	00010000	10	16	0	00110000	30	48
DC1	00010001	11	17	1	00110001	31	49
DC2	00010010	12	18	2	00110010	32	50
DC3	00010011	13	19	3	00110011	33	51
DC4	00010100	14	20	4	00110100	34	52
NAK	00010101	15	21	5	00110101	35	53
SYN	00010110	16	22	6	00110110	36	54
ETB	00010111	17	23	7	00110111	37	55
CAN	00011000	18	24	8	00111000	38	56
EM	00011001	19	25	9	00111001	39	57
SUB	00011010	1A	26	:	00111010	3A	58
ESC	00011011	1B	27	;	00111011	3B	59
FS	00011100	1C	28	$<$	00111100	3C	60
GS	00011101	1D	29	$=$	00111101	3D	61
RS	00011110	1E	30	$>$	00111110	3E	62
US	00011111	1F	31	?	00111111	3F	63

Tabella 3.4: Tabella del codice ASCII (dal carattere 0 al 63).

carattere	codice ASCII	equivalente esadecimale	equivalente numerico	carattere	codice ASCII	equivalente esadecimale	equivalente numerico
@	01000000	40	64	‘	01100000	60	96
A	01000001	41	65	a	01100001	61	97
B	01000010	42	66	b	01100010	62	98
C	01000011	43	67	c	01100011	63	99
D	01000100	44	68	d	01100100	64	100
E	01000101	45	69	e	01100101	65	101
F	01000110	46	70	f	01100110	66	102
G	01000111	47	71	g	01100111	67	103
H	01001000	48	72	h	01101000	68	104
I	01001001	49	73	i	01101001	69	105
J	01001010	4A	74	j	01101010	6A	106
K	01001011	4B	75	k	01101011	6B	107
L	01001100	4C	76	l	01101100	6C	108
M	01001101	4D	77	m	01101101	6D	109
N	01001110	4E	78	n	01101110	6E	110
O	01001111	4F	79	o	01101111	6F	111
P	01010000	50	80	p	01110000	70	112
Q	01010001	51	81	q	01110001	71	113
R	01010010	52	82	r	01110010	72	114
S	01010011	53	83	s	01110011	73	115
T	01010100	54	84	t	01110100	74	116
U	01010101	15	85	u	01110101	75	117
V	01010110	56	86	v	01110110	76	118
W	01010111	57	87	w	01110111	77	119
X	01011000	58	88	x	01111000	78	120
Y	01011001	59	89	y	01111001	79	121
Z	01011010	5A	90	z	01111010	7A	122
[	01011011	5B	91	{	01111011	7B	123
\	01011100	5C	92	|	01111100	7C	124
]	01011101	5D	93	}	01111101	7D	125
^	01011110	5E	94	~	01111110	7E	126
_	01011111	5F	95	□	01111111	7F	127

Tabella 3.5: Tabella del codice ASCII (dal carattere 64 al 127).

È opportuno sottolineare ancora una volta come i caratteri siano biunivocamente associati dalla codifica ASCII ai primi 128 numeri naturali. Ciascun codice ASCII può essere pertanto interpretato come carattere o come numero a seconda del contesto.

Si noti che la disposizione dei caratteri nella tabella rispetta l'ordinamento esistente tra le cifre decimali e tra le lettere dell'alfabeto inglese. Inoltre, l'associazione con i numeri induce sull'insieme dei caratteri un ordinamento totale per il quale lo spazio precede le cifre, che precedono le lettere maiuscole, che precedono le lettere minuscole. Per questo motivo quando si chiede ad un programma di ordinare stringhe di caratteri (per esempio un elenco di nominativi) generalmente l'ordinamento risultante considera stringhe "più piccole" quelle che iniziano con degli spazi, seguite da quelle che iniziano con cifre, seguite da quelle che iniziano con lettere maiuscole, seguite da quelle che iniziano con lettere minuscole.

Infine, osservando che le cifre sono elencate consecutivamente nella tabella, se si sottrae il codice associato a ciascuna cifra al codice associato alla cifra 0, entrambi interpretati come numeri naturali, si ottiene il valore numerico della cifra. Per esempio, sottranedo dal codice 55, associato alla cifra 7, il codice 48, associato alla cifra 0, si ottiene il numero sette.

Poiché tutti i calcolatori utilizzano normalmente stringhe di bit di lunghezza multipla di 8, la scelta di usare 7 bit per rappresentare i caratteri lascia inutilizzato l'ottavo bit (quello più a sinistra) che, per rispettare il codice ASCII, si assume sempre uguale a 0. Tuttavia, considerando anche le stringhe di 8 bit con il bit più a sinistra uguale a 1, la tabella può essere estesa fino a comprendere altri 128 "caratteri". La cosa ha un certo interesse pratico in quanto, oltre ai caratteri previsti dal codice ASCII originale, in tutte le lingue sono previste lettere non presenti nell'alfabeto inglese (per esempio in italiano vi sono le vocali accentate con due diveri accenti). Può essere inoltre utile avere a disposizione altri simboli matematici come il "minore o uguale", il "maggior o uguale", il "diverso", ecc.

Sono state pertanto introdotte delle estensioni al codice ASCII, che però non sono altrettanto universali perchè, ad esempio, molti dei caratteri presenti nell'estensione sono specifici di una particolare lingua. Tuttavia, questi codici, detti ASCII estesi, possono pertanto differire da sistema a sistema e possono, ad esempio, dipendere dalle versioni del software installato. Inoltre esistono programmi, come ad esempio i compilatori di cui parleremo più avanti, che non riconoscono correttamente i caratteri rappresentati da stringhe di bit che hanno un 1 nella posizione più a sinistra.

Pertanto nel seguito useremo esclusivamente caratteri appartenenti al codice ASCII originale e consigliamo di fare altrettanto se non si padroneggia adeguatamente il problema della rappresentazione dei caratteri che, nella sua formulazione più generale non è affatto banale.

A conclusione del paragrafo vogliamo soltanto accennare a un nuovo standard recentemente introdotto dagli organismi internazionali, enominato UNICODE, che usa stringe di 16 e più bit per rappresentare i caratteri e che risolve in modo più radicale e completo il problema delle specificità delle lingue, dei simboli matematici, ecc. Lo standard UNICODE prevede come caso particolare il codice ASCII per la rappresentazione dei caratteri fondamentali, che restano quelli precedentemente elencati. Lo standard ASCII è pertanto destinato ad essere usato ancora a lungo dalla maggior parte dei sistemi che non richiedono un uso sofisticato dei caratteri.

3.10 Rappresentazione in complemento dei numeri relativi

Le rappresentazioni di insiemi numerici diversi dai numeri naturali sono sempre basate sulla rappresentazione posizionale dei numeri naturali. In altri termini la funzione iniettiva che associa gli elementi dell'insieme numerico da rappresentare alle stringhe di bit rappresentanti è la composizione della funzione che associa i numeri da rappresentare a un opportuno intervallo dei numeri naturali, con la funzione usata per rappresentare tali numeri.

Un esempio tipico di questo modo di procedere è quello dei numeri relativi dove, volendo utilizzare stringhe di bit di lunghezza n, per rappresentare un numero relativo x appartenente all'intervallo $[-2^{n-1}, 2^{n-1} - 1]$ si usa:

- la funzione iniettiva:

$$f(x) = \begin{cases} x & \text{se } x \geq 0 \\ 2^n - |x| & \text{se } x < 0 \end{cases} \tag{3.4}$$

  per associare ad ogni numero relativo x da rappresentare un numero naturale $y = f(x)$ nell'intervallo $[0, 2^n - 1]$;

- la rappresentazione posizionale di y come rappresentante di x.

Si noti che nel caso di numeri negativi, il valore da rappresentare viene associato al complemento a 2^n del suo valore assoluto. Per tale motivo questo metodo di rappresentazione dei numeri relativi viene detto *in complemento su n bit*. Si noti anche che l'intervallo di numeri rappresentati è asimmetrico rispetto allo zero.

Accettando per il momento senza giustificazione questo modo di procedere, vale la pena notare che gli estremi dell'intervallo di numeri relativi da rappresentare sono scelti in modo che l'intervallo contenga esattamente 2^n numeri, che, come abbiamo già più volte visto, è il massimo numero di valori che possono essere rappresentati con n bit. Notiamo inoltre che anche l'intervallo di numeri naturali usato per associare le stringe di bit contiene esattamente 2^n numeri, come peraltro è naturale a motivo del fatto che la (3.4) è biiettiva.

Esempio 3.9

Ad esempio, volendo usare 8 bit, l'intervallo di numeri relativi che può essere rappresentato con questo metodo è $[-128, 127]$. Ciascun numero in tale intervallo viene quindi associato mediante la (3.4) ad un numero nell'intervallo $[0, 255]$, e precisamente:

- a se stesso se è un numero positivo o nullo (ad esempio il numero *relativo* 12 è associato al numero *naturale* 12);
- al *complemento* del suo valore assoluto a 256 se è un numero negativo (ad esempio il numero relativo -12 è associato al numero naturale $256 - 12 = 244$.

Poichè, secondo la rappresentazione posizionale, il rappresentante di 12 è la stringa di 00001100, e il rappresentante di 244 è la stringa 11110100, i numeri *relativi* 12 e -12 sono rappresentati rispettivamente da queste due stringhe.

Osservazione 3.12

Si noti che nell'impiegare la rappresentazione in complemento è importante specificare il numero di bit e dunque, implictamente, l'intervallo di valori che si possono rappresentare. Se infatti si vuole rappresentare il numero -12 in complemento su 10 bit, il numero naturale associato non è più 244, ma $2^{10} - 12 = 1012$.

Pertanto la rappresentazione in complemento su 10 bit di -12 è 1111110100 e non 0011110100, che su 10 bit rappresenta il numero positivo 244. È interessante notare che la rappresentazione in complemento di -12 su 10 bit si ottiene da quella su 8 bit *estendendo la stringa di di 8 bit a sinistra con degli* 1. Tale proprietà è del tutto generale e non dipende dal numero da rappresentare né dal numero di bit utilizzato.

Osservazione 3.13

Venendo al motivo per cui si utilizza un metodo apparentemente così poco naturale per rappresentare i numeri relativi, si può facilmente osservare che la (3.4) coincide con il *resto della divisione intera per* 2^n. Indicando tale operazione il simbolo $|x|_{2^n}$ dove x è un un numero relativo, vale la seguente proprietà generale:

$$||x|_{2^n} + |y|_{2^n}|_{2^n} = |x + y|_{2^n}. \tag{3.5}$$

In altre parole, dati due numeri relativi x e y qualsiasi, il resto della somma dei resti di x e y, è uguale al resto di $x + y$. Ora, poiché con il metodo di rappresentazione in complemento, i rappresentanti dei numeri relativi sono i rispettivi resti della divisione intera per 2^n, per effettuare la somma (algebrica) tra due numeri basta fare la somma dei rappresentanti e poi calcolarne il resto della divisione intera per 2^n. Per la (3.5), il risultato di questa operazione è il resto della divisione intera per 2^n del risultato della somma algebrica, che ne è anche il rappresentante.

Esempio 3.10

Nella rappresentazione in complemento su 8 bit il numero -53 è rappresentato dalla stringa 11001011 e il numero 71 è rappresentato dalla stringa 01000111. Sommando le due stringhe si ottiene la stringa di 9 bit 100010010 che rappresenta il numero naturale 274. Il resto modulo 256 di tale numero è 18 che è proprio il risultato della somma algebrica $-53 + 71$.

Si noti che l'operazione di resto modulo 256 equivale molto banalmente a prendere gli 8 bit più destra dalla stringa prodotta dalla somma, scartando l'eventuale riporto (la stringa 00010010 nel nostro caso). Pertanto l'intera operazione di somma algebrica può essere completamente effettuata manipolando i rappresentanti senza bisogno di considerare esplicitamente i valori rappresentati.

L'ultima affermazione dell'esempio, tuttavia, è vera solo se non si verifica un overflow e cioè se il risultato della somma appartiene ancora all'intervallo $[-2^{n-1}, 2^{n-1} - 1]$. In caso contrario, pur essendo ancora vera la (3.5), il resto a secondo membro *non è più il rappresentante di* $x + y$.

Ad esempio, se n è ancora 8, e se $x = 89$ e $y = 93$, la quantità $|89 + 93|_{256} = 182$ è ancora il resto della divisione intera di 182 per 256, ma tale numero è il rappresentante di -74 (è infatti $256-74 = 182$) e non di 182. La cosa d'altra parte è del tutto naturale potendo rappresentare con 8 bit solo i numeri relativi appartenenti all'intervallo $[-128, 127]$.

Osservazione 3.14

Il vantaggio della rappresentazione in complemento sta proprio nella semplicità con cui si può effettuare la somma algebrica direttamente sulle stringhe rappresentanti. Si tratta infatti di sommare tali stringhe interpretandole come normali numeri naturali, salvo poi controllare che non si sia verificato un overflow. Tale controllo, tuttavia, non presenta alcuna difficoltà (cfr. successiva osservazione 3.16). Anche il passaggio dal valore da rappresentare al suo rappresentante non è particolarente oneroso. Se il numero da rappresentare è positivo, il problema è lo stesso dei numeri naturali, se è negativo si può sfruttare il fatto che sostituendo ciascun bit con il suo complemento (gli "0" con gli "1" e viceversa) e sommando 1 alla stringa risultante, si ottiene direttamente il complemento a 2^n a partire dalla stringa di bit, senza dover effettuare sottrazioni.

Ad esempio, volendo ricavare il rappresentante di -12, si può calcolare la stringa che rappresenta 12 (come per quanlunque numero naturale operando divisioni successive per 2), invertirne i bit e sommare 1. Si ottiene in tal modo la stringa 11110100, che, come abbiamo visto in un esempio precedente è il numero naturale 244, rappresentante di -12 su 8 bit.

Osservazione 3.15

Un altro vantaggio della rappresentazione in complemento sta nella possibilità di determinare immediatamente il segno del valore rappresentato, senza bisogno di riconvertire la stringa di bit.

Ricordando che, nella rappresentazione in complemento, l'insieme da rappresentare è del tipo $[-2^{n-1}, 2^{n-1} - 1]$, l'insieme dei rappresentanti è del tipo $[0, 2^n - 1]$, e la corrispondenza tra i due è data dalla 3.4, si può osservare facilmente che:

- i valori negativi (nell'intervallo $[-2^{n-1}$,-1]) corrispondono ai valori maggiori o uguali a 2^{n-1} (nell'intervallo $[2^{n-1}, 2^n - 1]$);
- i valori positivi o nulli (nell'intervallo $[0, 2^{n-1} - 1]$) corrispondono se stessi.

Di conseguenza, i rappresentanti dei numeri negativi coincidono con la rappresentazione posizionale dei numeri maggiori o uguali a 2^{n-1} *che hanno sempre il bit più a sinistra (quello associato appunto alla potenza* 2^{n-1}*) uguale a* 1, mantre i numeri positivi coincidono con la rappresentazione posizionale dei numeri minori di 2^{n-1} *che hanno sempre il bit più a sinistra uguale a* 0.

Questo permette di dedurre immediatamente il segno del valore rappresentato dal primo bit del rappresentante.

Osservazione 3.16

Le precedenti considerazioni permettono di verificare facilmente se c'e' stato overflow in una somma algebrica tra numeri relativi rappresentati in complemento. Il metodo si basa sull'osservazione del bit più a sinistra degli operandi e del risultato che, come abbiamo visto, indica il segno del numero rapprsentato. Se il segno degli operandi è diverso non può esserci overflow come è facile verificare osservando che in questo caso, se gli operandi appartengono all'intervallo di rappresentazione, anche il risultato vi appartiene. Se viceversa gli operandi hanno lo stesso segno, occorre verificare che anche il risultato abbia lo stesso segno perché in caso contrario si è evidentemente avuto un overflow.

Ad esempio, operando sempre in complemento su 8 bit, se gli operandi sono 01101011 (rappresentante del numero 107) e 00111000 (rappresentante del numero 56), la somma produce la stringa 10100011 che non viene modificata operando su di essa il resto modulo 256. Tale stringa rappresenta però un numero negativo, mentre gli operandi sono entrambi positivi e dunque l'operazione ha prodotto un overflow e il risultato è errato. Come controprova si osservi che $107 + 56 = 153$ che è fuori dell'intervallo di rappresentazione.

Osservazione 3.17

Un'ultima osservazione è la seguente. Nella somma algebrica di numeri relativi rappresentati in complemento, un eventuale riporto finale diverso da 0 non indica overflow e viene eliminato dall'operazione di resto modulo 2^n. Al contrario, nella somma tra numeri naturali rappresentati secondo la rappresentazione posizionale un eventuale riporto finale diverso da 0 indica overflow.

3.11 Rappresentazione in segno e modulo dei numeri relativi

Una rappresentazione alternativa dei numeri relativi è basata sull'idea di separare il segno del numero da rappresentare dal suo *modulo* (cioè dal suo valore assoluto). Per questo motivo essa è denominata *rappresentazione in segno e modulo*.

Nella rappresetazione in segno e modulo, il modulo del numero, che è un numero naturale, è rappresentato con il solito metodo, mentre il segno, che può assumere solo due possibili valori, è rappresentato con un solo bit. Solitamente si associa la cifra 0 al segno positivo e la cifra 1 al segno negativo e si colloca il bit di segno a sinistra della stringa che rappresenta il modulo. Si noti che tale convenzione è coerente con la proprietà della rappresentazione in complemento descritta nell'osservazione 3.15.

Esempio 3.11

Volendo usare stringhe di 8 bit, il numero 45 risulta codificato con la stringa 00101101, mentre il numero -45 risulta codificato con la stringa 10101101. Si noti che anche nella rappresentazione in segno e modulo occorre specificare il numero di bit impiegati. Infatti, volendo codificare -45 su 10 bit si deve usare la stringa 1000101101. Si noti anche che non ha senso in questa rappresentazione interpretare le stringhe associate ai numeri negativi come numeri naturali, perché tali stringhe sono solo la giustapposizione di due elementi indipendenti: il segno e il modulo.

Nonostante la sua semplicità, la rappresentazione in segno e modulo non è solitamente usata per rappresentare i numeri relativi perché essa richiede un procedimento più complicato (dal punto di vista di un esecutore automatico) per effettuare le somme algebriche. Inoltre la rappresentazione in segno e modulo ha la caratteristica di prevedere due rappresentazioni diverse per il numero 0: la stringa composta da tutti zeri e la stringa composta da tutti zeri tranne il primo bit uguale a uno. Tale caratteristica contribuisce a complicare la manipolazione della rappresentazione in segno e modulo perché, ad esempio, per stabilire se due stringhe rappresentano lo stesso numero non si può semplicemente verificare se sono uguali o no[3].

[3] In realtà, a ben vedere, questa difficoltà deriva dal fatto che la rappresentazione in segno è modulo non verifica tutte le proprietà generali di una rappresentazione. Essa infatti non è una funzione in quanto associa due rappresentanti allo zero (a questo proposito vedi anche l'osservazione 3.6).

Un vantaggio della rappresentazione in segno e modulo è peraltro il fatto che l'intervallo di numeri rappresentati è simmetrico rispetto allo zero. Nel caso di impiego di stringhe di n bit, l'itervallo di rappresentazione è $[-2^{n-1}, 2^{n-1}]$[4]. Per questo ed altri motivi la rappresentazione in segno e modulo viene solitamente usata per la mantissa nella rappresentazione in virgola mobile dei numeri reali (cfr. paragrafo successivo).

3.12 Rappresentazione in virgola moblie dei numeri reali

La rappresentazione dei numeri reali mediante stringhe di bit di lunghezza fissata pone un problema. Abbiamo infatti visto che per rappresentare i valori di un insieme è necessario che la sua cardinalità sia finita. Nel caso dei numeri naturali e dei numeri relativi questo è stato ottenuto limitando l'intervallo dei valori rappresentati, con la conseguente possibilità di overflow durante la manipolazione dei rappresentanti. Un procedimento analogo nel caso dei numeri reali non è sufficiente a motivo della natura continua di tale insieme: anche un insieme limitato di numeri reali contiene infiniti valori e pertanto non esiste una funzione iniettiva che lo metta in corrispondenza con stringhe di bit di lunghezza assegnata.

Per illustrare come è possibile risolvere il problema, introdurremo direttamente un metodo concreto per associare numeri reali a stringhe di simboli, rinviando le considerazioni di pricipio e l'esame delle proprietà di tale metodo di rappresentazione.

Nel seguito assumiamo di voler rappresentare i numeri reali appartenenti ad un intervallo limitato A simmetrico rispetto allo 0. Assumiamo inoltre di voler impiegare come rappresentanti stringhe di lunghezza n. Per motivi di semplicità espositiva svolgeremo la trattazione dettagliata usando le cifre decimali invece di quelle binarie. In altre parole mostreremo un metodo per associare gli elementi di A a stringhe di n cifre decimali. Accenneremo successivamente come la soluzione illustrata possa facilmente essere estesa al caso di stringhe di bit.

Un qualunque numero $x \in A$ può essere rappresentato da una coppia di numeri:

$$(m, e)$$

tali che:

$$x = m \times 10^e \tag{3.6}$$

Il numero m si dice *mantissa* e il numero e si dice *esponente*. Si noti che l'esponente è un numero intero relativo. Ad esempio, il numero reale 346.09801 può essere rappresentato dalla coppia $(346.09801, 0)$.

È facile rendersi conto che esistono infinite coppie (mantissa,esponente) che rappresentano x. Ad esempio il numero 346.09801, oltre che dalla coppia $(346.09801, 0)$, è rappresentato anche dalle coppie $(3460.9801, -1)$, $(34609.801, -2)$, $(346098.01, -3)$, $(3460980.1, -4)$, ecc.

Per evitare questo molteplicità di rappresentanti, si può scegliere tra le infinite coppie quella che risulta *normalizzata* rispetto ad un criterio fissato, che solitamente è quello di imporre che il numero m abbia una sola cifra intera diversa da zero. Ad esempio, nel caso del numero 346.09801 si sceglie come unica coppia rappresentante la coppia normalizzata $(3.4609801, 2)$. Si noti che, assumendo una rappresentazione normalizzata, il punto decimale può essere omesso essendo nota a priori la sua posizione nella mantissa. Si noti inoltre che il numero 0 richiede una rappresentazione speciale perchè la sua mantissa deve essere nulla e dunque non ha cifre intere. Si conviene pertanto di rappresentare il numero 0 con la coppia $(0, 0)$.

Fatta questa premessa, per rappresentare su n cifre i numeri contenuti in A, si fissa il numero h di cifre usate per rappresentare l'esponente, e si usano le cifre rimanenti (cioè $n - h$) per rappresentare la mantissa, assumendo implicitamente il punto decimale dopo la prima di queste cifre. Poiché tanto la mantissa che l'esponente hanno un segno, per evitare ambiguità, in tutti gli esempi seguenti indicheremo sempre esplicitamente il segno di entrambi, anche quando è positivo. I due segni si aggiungono pertanto alle n cifre nel formare il rappresentante di un elemento di A. Si noti che per la 3.6, il segno del numero rappresentato coincide con quello della mantissa.

[4] Si noti che tale intervallo contiene $2^n - 1$ numeri, uno in meno rispetto al numero di stringhe disponibili proprio perché lo zero è rappresentato da due stringhe diverse

Ad esempio, se $n = 6$, e usiamo due cifre per rappresentare l'esponente e quattro cifre per rappresentare la mantissa, il numero 346.09801 viene rappresentato dalla coppia $(+3460, +02)$, dove si è omesso il punto decimale dopo il 3. Si noti che in questo modo le ultime quattro cifre siginificative del numero vengono perdute dando luogo a *una rappresentazione approssimata del numero reale*[5]. Il fatto non deve peraltro meravigliare perché stiamo rappresentando un numero infinito di valori (i numeri in A) attraverso un numero finito di stringhe. La cosa, che a stretto rigore sarebbe impossibile, si può fare solo a prezzo di introdurre un *errore* nella rappresentazione.

Si noti che, sebbene per migliorare la leggibilità del testo si sia deciso di usare la notazione $(+3460, +02)$ per indicare il rappresentante di un numero, una volta decisa la posizione realativa dei segni, delle cifre dedicate all'esponente e di quelle dedicate alla mantissa, la rappresentazione è di fatto una stringa di $n + 2$ simboli: un segno, $n - h$ cifre decimali per la mantissa, un segno, h cifre decimali per l'esponente, realizzando così l'obiettivo di rappresentare un numero reale con una stringa finita di simboli.

Prima di esaminare i vantaggi di questo modo di procedere, può essere utile esaminare diversi esempi di rappresentazione di elementi di A con il metodo illustrato:

valore	rappresentante
34609.801	$(+3460, +04)$
-34609.801	$(-3460, +04)$
34609801	$(+3460, +07)$
0.0034609801	$(+3460, -03)$
1	$(+1000, +00)$
10	$(+1000, +01)$
0.01	$(+1000, -02)$
-1000	$(-1000, +03)$

Si noti come, i numeri che differiscono per un fattore potenza di 10 hanno sempre la stessa mantissa. Per questo motivo, il tipo di rappresentazione appena descritto viene indicato con il termine *virgola mobile*. Il punto decimale può infatti spostarsi senza effetti sulla rappresentazione della mantissa.

Fermiamoci ora a considerare alcune importanti proprietà della rappresetazione in virgola mobile. Senza perdita di generalità, consideriamo ancora una rappresentazione che usi sei cifre decimali di cui due destinate all'esponente più i due segni. A ciascun rappresentante corrisponde un ben preciso numero reale. Per esempio, alla coppia $(+9999, +99)$ corrisponde il numero reale:

$$9.999 \times 10^{99}$$

che è anche il numero più grande che possa essere rappresentato esattamente con quattro cifre di mantissa e due di esponente, mentre alla coppia $(+1000, -99)$ corrisponde il numero reale:

$$1.0 \times 10^{-99}$$

che è anche il numero positivo più piccolo più prossimo allo zero) che possa essere rappresentato esattamente [6]. Analogamente, considerando mantisse negative, si osserva immediatamente che il numero reale:

$$-9.999 \times 10^{99}$$

è il numero più piccolo che possa essere rappresentato esattamente, mentre il numero reale:

$$1.0 \times -10^{-99}$$

[5] In realtà dovendo rinunciare alle cifre finali, invece di un semplice *troncamento* delle cifre in più è opportuno operare un *arrotondamento* che cerchi di minimizzare l'errore. La questione non è tuttavia essenziale ai fini della comprensione della rappresentazine in virgola mobile e dunque in tutti gli esempi di questo paragrafo, per non complicare inutilmente la trattazione, useremo sempre il troncameto per eliminare le cifre in eccesso dalla mantissa.

[6] Si ricordi che la prima cifra rapresenta la parte intera della mantissa e che pertanto deve essere diversa da 0.

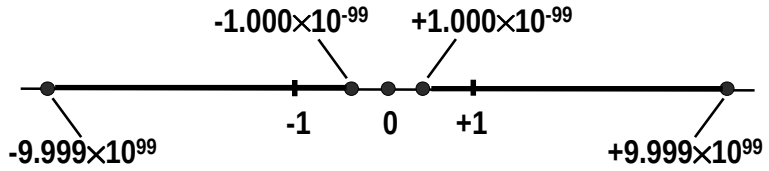


Figura 3.3: L'insieme di reali rappresentabili con quattro cifre di mantissa e due di esponente.

è il numero negativo più grande (più prossimo allo zero) che possa essere rappresentato esattamente.

Pertanto, usando la rappresentazione in virgola mobile con stringhe di sei cifre decimali, di cui due dedicate all'esponente, è possibile rappresentare tutti i numeri reali appartenenti all'intervallo:

$$[-9.999 \times +10^{+99}, +9.999 \times +10^{+99}]$$

Alcuni dei numeri appartenenti a questo intervallo vengono rappresentati esattamente. Ad esempio il numero 2235000 è rappresentato esattamente dalla coppia $(+2235, +06)$. Altri numeri vengono approssimati al numero più vicino che può essere rappresentato esattamente. Ad esempio, il numero 2235043 viene rappresentato in modo approssimato dalla stessa coppia $(+2235, +06)$, il che equivale a dire che viene approssimato a 2235000. In particolare, i numeri più piccoli in valore assoluto di 1.0×10^{-99} vengono approssimati a 0.

Quest'ultima osservazione richiede alcune ulteriori considerazioni. Abbiamo visto che la rappresentazione approssimata dei numeri reali è inevitabile e più avanti vedremo come la rappresentazione in virgola mobile consenta di ridurre gli effetti negativi di tale fenomeno. Tuttavia, l'approssimazione di un numero piccolo a 0 non è una normale approssimazione e va considerata un fatto anomalo. In generale, infatti, quando in un calcolo un operando viene approssimato, l'effetto sul risultato è proporzionale all'errore introdotto. Ad esempio, mentre il risultato esatto della moltiplicazione $(-1.045901 \times 10^{-95}) \times (8.00067 \times 10^{97})$ è $-8.36790875367 \times 10^2$, usando i rappresentanti (approssimati) dei fattori si ottiene il valore -8.3600×10^2, che differisce di poco (esattamente 0.790875367) dal risultato esatto. Se però l'approssimazione consiste nel sostituire lo zero ad un numero diverso da zero, per quanto piccola possa essere stata l'approssimazione, il risultato può essere completamente stravolto. Ad esempio il risultato esatto della moltiplicazione $(-1.045901 \times 10^{-103}) \times (8.00067 \times 10^{97})$ è $-8.36790875367 \times 10^{-4}$, ma se usiamo i rappresentanti, poichè il primo fattore viene approssimato a zero, si ottiene come risultato zero, che ha un siginificato completamente diverso dal numero -0.000836790875367!

Il fenomeno che corrisponde alla approssimazione a zero di un numero troppo piccolo per essere rappresentato diversamente viene indicato con il termine *underflow* e corrisponde ad una situazione anomala, per certi versi simile all'overflow. In effetti, a meno che non si stiano effettuando solo somme algebriche, il verificarsi di un underflow implica l'impossibilità di effettuare il calcolo in modo accettabile. Quando non si possono escludere operazioni diverse dalla somma algebrica, i numeri che producono undeflow vanno considerati non rappresentabili, e per essi possono essere ripetute considerazioni analoghe a quelle fatte nel paragrafo 3.7.

La figura 3.3 mostra graficamente l'intervallo di numeri reali rappresentabile nell'ipotesi di usare quattro cifre per la mantizza e due per l'esponente. Nella figura viene evidenziato il fatto che i numeri intorno allo zero danno luogo a un underflow e non vengono pertanto sempre rappresentati validamente.

Osservazione 3.18

Si noti che l'ampiezza dell'intevallo, e quindi la possibilità di overflow, dipende dal numero di cifre destinate all'esponente. In altre parole, se all'esponente fosse assegnata solo una cifra (e cinque cifre alla mantissa), l'intervallo di rappresentazione si ridurrebbe a $[-9.9999 \times +10^{+9}, +9.9999 \times +10^{+9}]$, considerevolmente più piccolo del caso precedente. Il numero di cifre dell'esponente influenza in modo del tutto analogo l'intorno dello zero in cui si verifica l'underflow: nel caso in cui si usa una sola cifra per l'esponente il numero viene approssimato a zero non appena il suo valore assoluto scende sotto 10^{-9}.

Osservazione 3.19

Dopo aver considerato i limiti di rappresentabilità e la loro dipendenza dal numero di cifre assegnate all'esponente, occupiamoci del grado di approssimazione introdotto dalla rappresentazione in virgola mobile. A tal fine proviamo a elencare tutti i numeri reali rappresentati esattamente, limitandoci, per semplicità, ai numeri positivi (analogo discorso si può fare per i negativi). I numeri positivi rappresentati esattamente si ottengono considerando nell'ordine tutte le possibili combinazioni di sei cifre decimali, partendo da esponenti negativi. L'elenco che si ottiene è il seguente:

$$\begin{aligned} &1.000 \times 10^{-99}, 1.001 \times 10^{-99}, 1.002 \times 10^{-99}, \quad \ldots \quad 9.997 \times 10^{-1}, 9.998 \times 10^{-1}, 9.999 \times 10^{-1}, \\ &1.000 \times 10^{0}, 1.001 \times 10^{0}, 1.002 \times 10^{0}, \quad \ldots \quad 9.997 \times 10^{99}, 9.998 \times 10^{99}, 9.999 \times 10^{99}, \end{aligned}$$

Provando a sottrarre le coppie di valori contigui, otteniamo una lista di differenze crescenti, da valori molto piccoli come 0.001×10^{-99} a valori molto grandi come 0.001×10^{99}, passando per valori intermedi come 0.001×10^{0}. In altre parole, i numeri rappresentati esattamente non sono distribuiti in modo uniforme su tutto l'intervallo di rappresentazione, ma, limitandoci ai numeri positivi, sono molto "fitti" nella zona vicina allo zero e sono molto "radi" nella zona vicina all'estremo superiore dell'intervallo.

Poiché il grado di approssimazione (cioè l'errore assoluto introdotto) dalla rappresentazione dipende dalla distanza tra due numeri consecutivi rappresentati esattamente, si deduce che *la rappresentazione in virgola mobile introduce errori piccoli per numeri piccoli ed errori grandi per numeri grandi*. Questa proprietà è molto positiva perché è evidente che su un numero molto grande può essere accettabile un errore anche grande in termini assoluti, mentre su numeri piccoli sono accettabili solo errori molto piccoli. In altre parole, è conveniente avere un *errore relativo* costante e sufficientemente piccolo, piuttosto che errori assoluti costanti e piccoli.

Ricordando che l'errore relativo è il rapporto tra l'errore assoluto e il numero rappresentato, possiamo immediatamente valutare l'errore relativo introdotto dalla rappresentazione in virgola mobile sulla base delle quantità riportate in precedenza. Si verifica facilmente che l'errore relativo è in tutti i casi compreso tra 10^{-3} e 10^{-4} e dunque risulta sostanzialmente costante. Il risultato non è casuale. In generale, assumendo sempre di usare n cifre decimali, di cui h assegnate all'esponente, l'errore relativo è pari approssimativamente a $10^{-(n-h)}$. In altre parole, l'errore relativo è inversamente proporzionale al numero di cifre della mantissa.

In definitiva, la rappresentazione in virgola mobile permette di rappresentare i numeri reali con una precisione relativa costante. L'intervallo di rappresentazione è tanto maggiore quante più cifre vengono assegnate all'esponente, mentre la precisione (relativa) è tanto maggiore quante più cifre vengono assegnate alla mantissa. Fortunatamente, anche un numero limitato di cifre assegnate all'esponente garantiscono un un'elevata ampiezza dell'intervallo, per cui, disponendo di un numero adeguato di cifre non è difficile ripartirle tra mantissa ed esponente in modo da garantire un intervallo ampio senza compromettere eccessivamente la precisione.

Per concludere, vogliamo accennare a cosa accade quando si passa dalle cifre decimali ai bit. Tutte le considerazioni fatte si possono ripetere con le seguenti specificità:

- l'esponente viene determinato in modo che sia $x = m \times 2^{e}$ e cioè si moltiplica la mantissa per una potenza di 2;
- l'unica cifra prima del punto decimale è sempre un 1 (le cifre sono solo 0 e 1);
- i due segni possono pure essere rappresentati da un bit dando luogo a una rappresentazione completamente binaria.

Sebbene in principio sia possibile scegliere arbitrariamente il numero di bit da usare, così come la loro ripartizione tra mantissa ed esponente, oggi è universalmente accettato lo standard IEEE 754 per la rappresentazione in virgola mobile. Tale standard prevede due versioni, denominate rispettivamente, *in singola precisione* e *in doppia precisione* che si distinguono per il numero complessivo di bit utilizzati. La tabella 3.6 riassume le principali caratteristiche dello standard IEEE 754 che può risultare utile conoscere nell'attività di programmazione. In particolare, è utile avere

versione	#bit totali	#bit esponente (incl. segno)	#bit mantissa (escl. segno)	intervallo di rappresentazione	underflow	precisione (errore rel.)
singola precisione	32	8	24	$[-10^{38}, +10^{38}]$	$< 10^{-38}$	10^{-7}
doppia precisione	64	11	53	$[-10^{308}, +10^{308}]$	$< 10^{-308}$	10^{-16}

Tabella 3.6: Rappresentazione IEEE 754: caratteristiche principali.

presente il contenuto delle ultime tre colonne della tabella e le differenze tra le due versioni dello standard. Si noti che i dati sull'intervallo di rappresentazione e sulla precisione sono indicativi, essendo riportati in termini di potenze di 10. Una trattazione dettagliata dello standard, incluse le modalità per passare da un numero reale alla stinga di bit e viceversa, vanno comunque al di là dei nostri obiettivi.

Capitolo 4

Macchina astratta (bozze, v. 2.0)

In questo capitolo e nei successivi ci occuperemo di illustrare le caratteristiche di un esecutore reale in grado di elaborare automaticamente informazioni. Il punto di vista da cui intendiamo studiare l'esecutore è quello di chi vuole formulare algoritmi e pertanto la nostra attenzione sarà essenzialmente concentrata sul linguaggio di programmazione usato per formulare gli algoritmi stessi. In particolare, in questo capitolo illustreremo le operazioni elementari che il linguaggio di programmazione permette di esprimere in termini generali, rimandando ai successivi gli aspetti di dettaglio.

Per la corretta comprensione del capitolo si assume che siano stati acquisiti i concetti di algoritmo ed esecutore e che sia chiaro il rapporto di sostanziale identità tra l'esecutore e il linguaggio usato per formulare l'algoritmo.

4.1 Scelta del linguaggio di programmazione e natura dell'esecutore

L'esecutore che descriveremo è quello corrispondente ai linguaggi C e C++. Si tratta di due linguaggi strettamente legati in quanto il primo è un sottoinsieme del secondo. Di fatto nei prossimi capitoli faremo riferimento al linguaggio C come base, con l'aggiunta di costrutti specifici del C++ dove tali aggiunte consentono una illustrazione più chiara dei concetti. Nel seguito il termine *linguaggio di programmazione* o, più brevemente *linguaggio*, indica questo linguaggio C esteso con costrutti C++.

Tutta la descrizione dell'esecutore è fatta esclusivamente attraverso il linguaggio di programmazione e prescinde della natura fisica dell'esecutore reale. Tale esecutore risulta pertanto una *macchina astratta*, capace di fare tutto e solo quello che può essere espresso mediante il linguaggio e dunque completamente definita da esso.

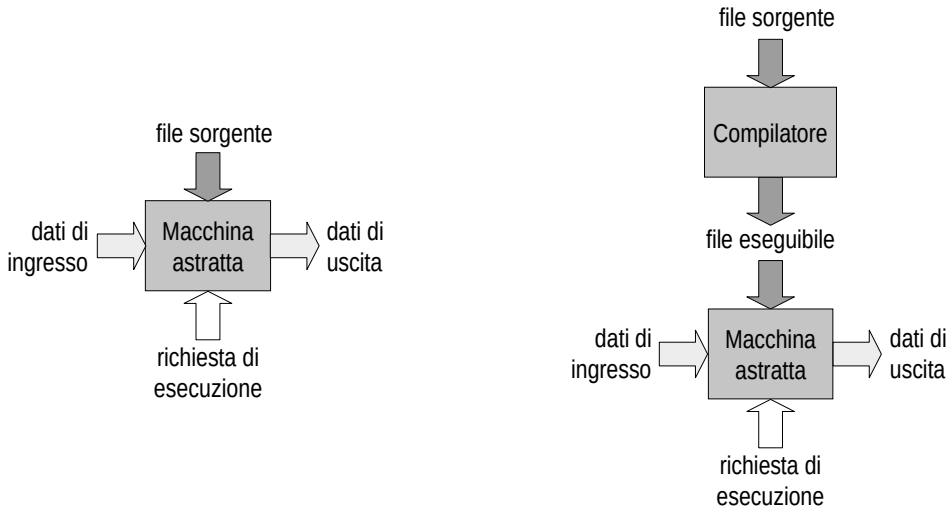
Prima di addentrarci nella descrizione della macchina astratta è tuttavia opportuno accennare a come in pratica essa può essere *implementata*[1].

L'esecutore reale è una macchina fisica che può acquisire un programma scritto nel linguaggio di programmazione e che può eseguire il corrispondente algoritmo quando l'utente lo richiede. Senza entrare nel dettaglio della struttura fisica del'esecutore reale, e in particolare delle modalità concrete con cui il programma viene acquisito e di come l'utente chiede di avviare l'esecuzione[2], un primo punto da chiarire è se la macchina è in grado di "comprendere" direttamente il linguaggio o no.

Nel primo caso, quando l'operatore chiede l'esecuzione del programma, l'esecutore "legge" il testo del programma e lo esegue.

[1] Il verbo *implementare*, derivato dall'inglese, è un sinonimo di realizzare. Esso è tuttavia largamente usato nella letteratura perché risulta meno generico di tale termine italiano; il termine implementare indica infatti l'azione di tradurre in termini fattuali la descrizione astratta di un sistema.

[2] Ad esempio, disponendo di un moderno personale computer, il programma viene fornito sotto forma di un file di testo e la richiesta di esecuzione può consistere in uno più "click" del mouse o nella pressione di uno o più tasti della tastiera.



(a) lettura diretta del programma (b) traduzione del programma

Figura 4.1: Strategie di implementazione della macchina astratta.

Nel secondo caso, per operare, l'esecutore deve ricevere le istruzioni in una forma diversa (solitamente più semplice) da quella usata per formulare l'algoritmo. In questo caso si procede pertanto a una *traduzione* del programma scritto nel linguaggio di programmazione (denominato *programma sorgente*) in un *equivalente programma* scritto in un linguaggio *direttamente interpretabile dall'esecutore reale* (detto *programma eseguibile*). Il linguaggio di arrivo della traduzione, usato per formulare il programma eseguibile, viene detto pertanto *linguaggio della macchina*.

Come abbiamo detto, la traduzione del programma sorgente nel programma eseguibile deve produrre un programma perfettamente equivalente. Questo siginifica che quando il programma eseguibile viene eseguito dall'esecutore reale, *l'effetto prodotto deve essere lo stesso che si sarebbe avuto se il programma sorgente fosse stato eseguito dalla macchina astratta descritta nei paragrafi successivi, come se essa fosse in grado di interpretare direttamente il programma sorgente*.

La traduzione viene effettuata da un apposito programma, chiamato *compilatore* [3], che partendo dal programma sorgente contenuto in un file (detto sorgente) genera il programma eseguibile in un altro file (detto eseguibile).

In figura 4.1 sono mostati in forma grafica le due soluzioni. Il passaggio attraverso una fase di traduzione, sebbene possa sembrare laborioso, è molto utilizzato nella pratica e in particolare è normalmente impiegato con il linguaggio C/C++. Alcuni motivi di tale modo di procedere e i suoi vantaggi e svantaggi saranno discussi nel capitolo 10.

4.2 Elementi costitutivi dell'esecutore

Per facilitare l'illustrazione delle operazioni elementari che il linguaggio consente di esprimere partiamo da una descrizione astratta dell'esecutore di tipo strutturale, cioè delle parti che lo compongono, delle loro funzioni e delle loro relazioni reciproche.

L'esecutore che intendiamo descrivere è formato dalle seguenti parti:

- Un *interprete-attuatore*, o, più brevemente, *interprete*, che è l'entità attiva dell'esecutore in grado di compren-

[3] Per ora con questo termine intendiamo riferirci all'insieme di programmi che intervengono nel processo di traduzione senza distinguerli.

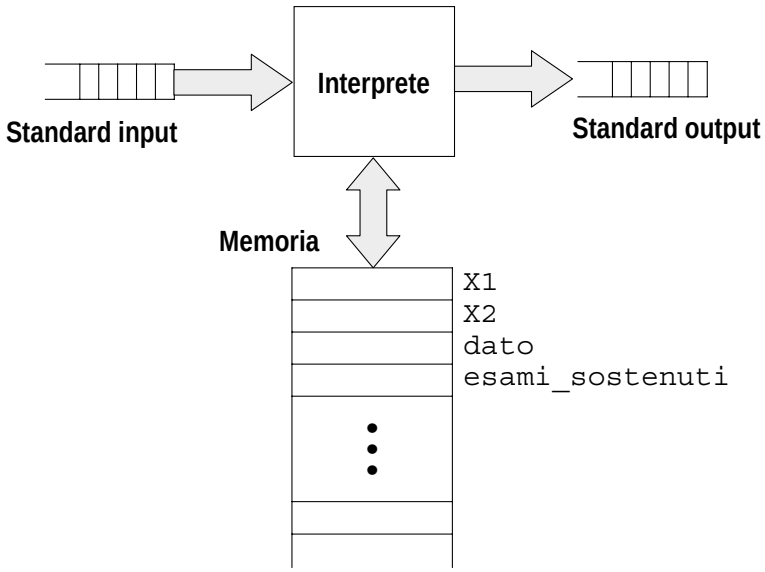


Figura 4.2: Struttura dell'esecutore.

dere le istruzioni del linguaggio e di svolgere le azioni che corrispondono a tali istruzioni, eventualmente agendo sulle altre parti dell'esecutore, di seguito elencate.

- Una *memoria*, che è l'entità in cui possono essere rappresentati in modo permanente i valori delle informazioni da elaborare. In ogni istante, la memoria è costituita da un insieme finito di celle ognuna delle quali può contenere la rappresentazione di un valore. A ciascuna cella è normalmente associato un nome che la individua in modo univoco, ed è sempre associato un tipo che determina i valori che in tale cella possono essere rappresentati e la modalità con cui sono di fatto rappresentati.

- Uno *standard input*, che è l'entità attraverso cui l'esecutore può ricevere informazioni dall'esterno. Per il momento definiamo lo standard input come una sequenza di valori generata esternamente all'esecutore. I meccanismi con cui vengono generati i valori che costituiscono lo standard input non fanno parte dell'esecutore.

- Uno *standard output*, che è l'entità attraverso cui l'esecutore può inviare informazioni all'esterno. Per il momento definiamo lo standard output come una sequenza di valori generata dall'esecutore e trasferiti esternamente ad esso. I meccanismi con cui vengono resi disponibili all'esterno dell'esecutore i valori che costituiscono lo standard output non fanno parte dell'esecutore.

Questi quattro componenti sono poi logicamente collegati tra loro in modo che i valori delle informazioni possano essere trasferiti tra di essi. In particolare i valori possono essere trasferiti dalla memoria all'interprete, dall'interprete alla memoria, dallo standard input alla memoria e dall'interprete allo standard output. Questa struttura è raffigurata nella figura 4.2.

È importante sottolineare che la descrizione appena fornita ha unicamente lo scopo di consentire nei prossimi paragrafi una spiegazione più agevole delle operazioni elementari che l'esecutore è in grado di svolgere e non corrisponde necessariamente alla struttura di un calcolatorie elettronico reale. In particolare, sebbene in un calcolatore reale sia presente un'interprete (normalmente indicato con il termine *unità di controllo*), una memoria e dei dispositivi perifierici per l'ingresso e l'uscita dei dati, così come un organo di collegamento tra queste componenti denominato bus di

sistema, la natura degli organi riportati in figura 4.2 è del tutto astratta e include solo alcuni aspetti dei corrispondenti organi reali di un calcolatore elettronico.

4.3 L'interprete-attuatore

Come è stato già detto l'interprete è l'entità che compie le operazioni specificate attraverso il linguaggio di programmazione. Durante l'esecuzione di un programma l'interprete esegue le operazioni indicate secondo una particolare sequenza dinamica che viene determinata dall'interprete stesso calcolando la verità o la falsità delle condizioni specificate nei passi decisionali dell'algoritmo.

Per quanto riguarda le capacità operative dell'interprete esso è in grado di (ulteriori dettagli sono riportati anche nei paragrafi successivi):

- *leggere* il contenuto di una cella della memoria: acquisire cioè il valore in essa rappresentato;
- *scrivere* un valore in una cella della memoria: trasferire cioè un valore in una cella in grado di rappresentarlo; il valore viene rappresentato nella cella secondo le modalità di rappresentazione associate a quella cella;
- *estrarre* il primo valore dello standard input e *scriverne* la rappresentazione in una cella di memoria opportuna;
- *aggiungere* un nuovo valore, comunque generato, allo standard output;
- *generare* nuovi valori attraverso un numero finito di operatori predefiniti, utilizzando se necessario valori acquisiti o generati in precedenza, inclusi valori costanti.

4.4 La memoria

La memoria è un insieme finito di celle in grado di contenere valori. Il linguaggio di programmazione permette di creare e distruggere celle di memoria attraverso apposite istruzioni.

Quando una cella viene creata, essa viene associata a un *tipo* che determina quali valori essa è in grado di rappresentare e le modalità con cui tali valori sono di fatto rappresentati [4]. Sono disponibili un insieme di tipi predefiniti, detti *primitivi*, ed è possibile definire, attraverso apposite istruzioni del linguaggio nuovi tipi, detti *tipi definiti dall'utente*.

In ogni caso i tipi disponibili sono indicati attraverso un nome o *identificatore*. Un identificatore è una sequenza di lettere e di cifre iniziata da una lettera e può contenere il carattere speciale _ (underscore). Sono pertanto esempi di identificatori i seguenti:

```
dato
Dato
X1
X2
esami_sostenuti
```

non sono invece esempi di identificatori i seguenti:

```
3dato
x$
esami sostenuti
```

[4] Si ricorda che un insieme di valori può essere rappresentato in generale in più modi e che dunque per determinare il valore a partire dal rappresentante occorre conoscere il tipo dell'informazione che, oltre all'insieme dei valori rappresentabili, include anche la legge che associa ciascun valore possibile al suo rappresentante.

Si noti in particolare che gli identificatori non possono iniziare con una cifra e che non devono contenere spazi; inoltre le lettere maiuscole sono considerate diverse da quelle minuscole (gli identificatori `dato` e `Dato` sono pertando due identificatori distinti).

Quando una cella viene creata, essa, oltre che ad un tipo, viene associata normalmente ad un identificatore che la individua. Ad esempio, nella figura 4.2 si supone che la memoria comprenda le celle associate agli identificatori: `X1`, `X2`, `dato`, `esami_sostenuti`. Sebbene l'associazione di una cella ad un identificatore non sia né necessaria né univoca (lo stesso identificatore può essere associato a più celle), assumeremo per ora che *tutte le celle siano associate ad un identificatore e che tale associazione sia univoca*.

Una cella di memoria, che come abbiamo detto è sempre associata a un tipo e a un identificatore, viene indicata con il termine *variabile* e ha le seguenti proprietà fondamentali:

- Una volta creata contiene sempre la rappresentazione di un valore tra quelli che possono essere rappresentati in base al tipo ad associato. Una variabile si dice *definita* se il valore in essa rappresentato è stato scritto nella variabile mediante il trasferimento di un valore dall'interprete o dallo standard input esplicitamente previsto dall'algoritmo, si dice *indefinita* altrimenti. Si noti che in base a tale definizione una variabile appena creata è indefinita.

- Una variabile definita contiene un'informazione solo se associamo concettualmente un attributo al nome della variabile. In tal caso infatti il tipo della variabile determina il valore rappresentato, mentre il nome associa al valore un attributo e dunque un siginificato. Si noti che, in mancanza del tipo e del nome, il contenuto di una cella di memoria da solo non determina né un valore, né tantomeno un'informazione.

- Una variabile indefinitia non contiene un'informazione perché la rappresentazione in essa contenuta è indeterminata. Di conseguenza non è mai lecito leggere una variabile indefinita (cioè trasferire il valore in essa rappresentato all'interprete o allo standard output) perchè il risultato dell'operazione e delle operazioni successive che usassero tale valore sarebbe a sua volta indeterminato.

L'ultimo punto merita ancora un commento. L'operazione di lettura di una variabile viene indicata tecnicamente con il termine *uso* della variabile. Quanto detto si traduce pertanto nella seguente regola del tutto generale e di importanza fondamentale nella programmazione:

Regola 4.1 *In un algoritmo, una variabile non può mai essere usata prima di essere stata definita*

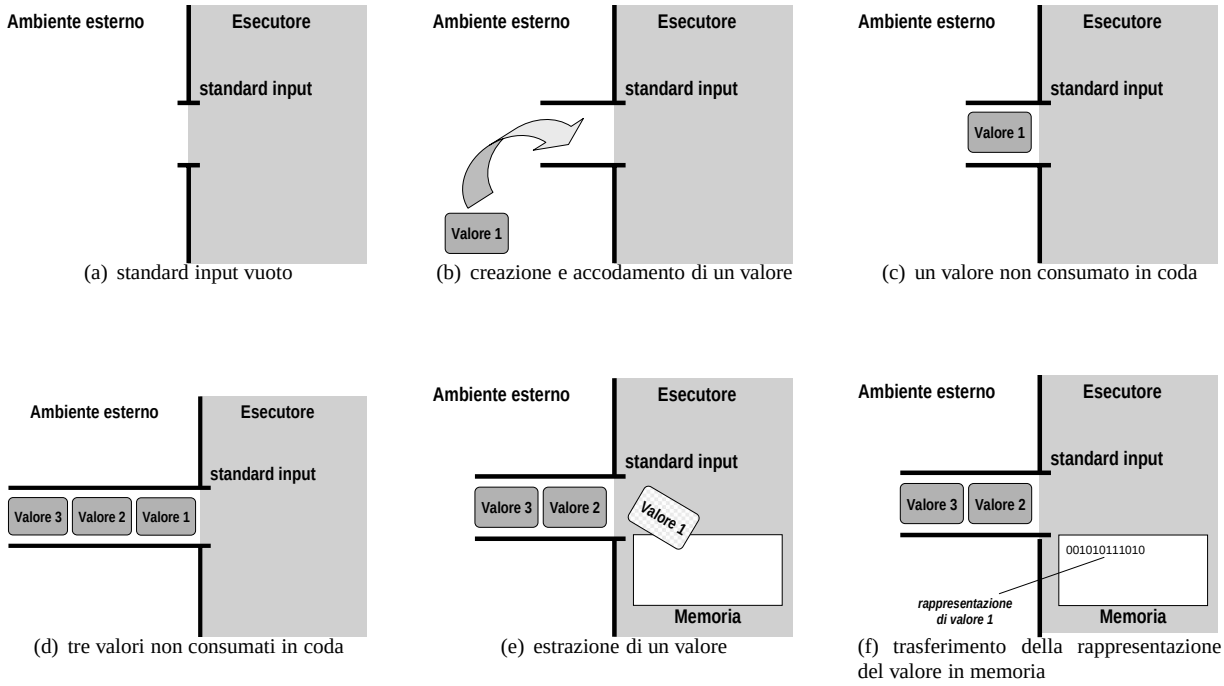
4.5 Lo standard input

Come abbiamo accennato, lo standard input è l'entità dell'esecutore attraverso cui possono essere acquisite informazioni dall'ambiente esterno.

Prima di descrivere con maggiore dettaglio il funzionamento dello standard input, vale la pena illustrare brevemente le modalità con cui possono essere generate le informazioni all'esterno dell'esecutore perchè possano poi essere da questi acquisite. Le informazioni giungono all'esecutore attraverso dispositivi fisici che prendono il nome di *periferiche di input*. Esempi di periferiche di input sono: la tastiera, le porte seriale, parallela e USB, il floppy disk, l'hard disk. Tutti questi dispositivi, che non sono parte dell'esecutore, sono in grado di generare valori nei modi più svariati; ad esempio, la tastiera genera informazioni corrispondenti ai tasti premuti dall'utente, la porta seriale genera le informazioni corrispondenti alla sequenza di bit che riceve dal dispositivo ad essa connesso (per es. un modem), ecc.

Durante l'esecuzione di un algoritmo da parte dell'esecutore, uno di questi dispositivi è connesso con lo standard input[5]. I valori generati dalla periferica connessa con lo standard input vengono trasferiti all'esecutore attraverso lo

[5]Le modalità con cui viene realizzata tale connessione non riguarda l'esecutore, ma l'ambiente operativo nel quale l'esecutore opera. Maggiori dettagli su questo aspetto verranno pertanto forniti nel capitolo 9.



(a) standard input vuoto (b) creazione e accodamento di un valore (c) un valore non consumato in coda

(d) tre valori non consumati in coda (e) estrazione di un valore (f) trasferimento della rappresentazione del valore in memoria

Figura 4.3: Lo standard input.

standard input uno dopo l'altro, nell'ordine con cui sono stati generati. Per questo motivo lo standard input può essere immaginato come una sequenza di valori generata esternamente all'esecutore.

Nella figura 4.3 sono mostrati in forma grafica gli aspetti principali del funzionamento dello standard input. Qualora non vi siano valori generati, lo standard input è vuoto (figura 4.3-a). Quando viene generato un valore, esso viene aggiunto allo standard input (figura 4.3-b) in modo da formare una sorta di coda di valori in attesa di essere acquisiti dall'esecutore (figure 4.3-c e 4.3-d).

Quando l'algoritmo richiede l'acquisizione di un valore attraverso lo standard input, il primo valore in coda viene estratto dalla coda che scorre di una posizione (figura 4.3-e). Il valore estratto viene trasferito in memoria sotto forma di una stringa di bit che dipende dal tipo della cella usata[6] (figura 4.3-f).

Da quanto detto, l'operazione che l'interprete esegue sullo standard input può essere descritta nel modo seguente:

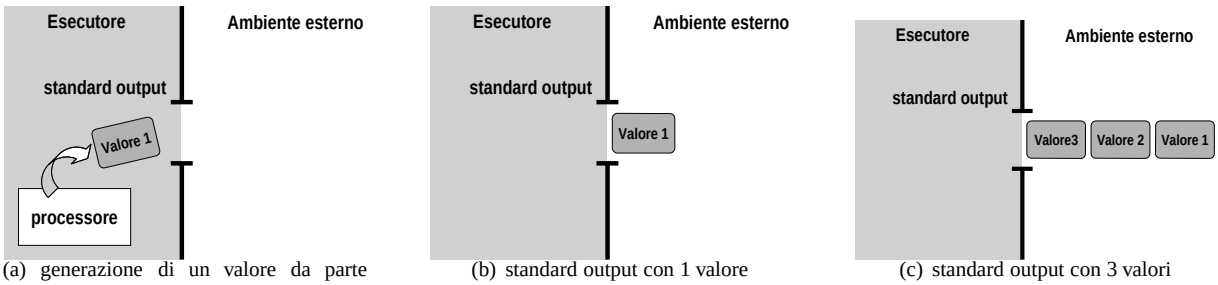
> *il primo valore presente nello standard input viene estratto e trasferito nella memoria dove viene mantenuto in una rappresentazione opportuna.*

È dunque evidente che le operazioni sullo standard input devono sempre specificare:

- il tipo da usare per determinare la rappresentazione del valore acquisito;
- la cella di memoria in cui trasferire tale rappresentazione

Va infine sottolineato che il buon esito delle operazioni sullo standard input non dipende esclusivamente dall'esecutore e dalla correttezza dell'algoritmo, ma anche dal verificarsi di opportune circostanze *all'esterno dell'esecutore e dunque indipendenti da esso*. Tali circostanze sono principalmente di due tipi.

[6] Ricordiamo che il tipo determina la rappresentazione da usare, come illustrato nel capitolo 3.



(a) generazione di un valore da parte dell'interprete e standard output vuoto

(b) standard output con 1 valore

(c) standard output con 3 valori

Figura 4.4: Lo standard output.

Un primo aspetto riguarda la corrispondenza tra il valore acquisito e il tipo specificato nell'operazione di acquisizione. Come sappiamo, un tipo fa riferimento ad un insieme ben definito di valori e dunque è necessario che il valore acquisito appartenga a tale insieme perchè la sua rappresentazione possa essere determinata correttamente. Tuttavia, dal momento che il valore è generato esternamente all'esecutore, nulla garantisce che esso verifichi tale requisito. Perchè questo avvenga occorre che chi manovra la periferica, lo faccia in modo da generare i valori forniti all'esecutore nell'ordine "giusto", cioè nell'ordine con cui l'algoritmo se li aspetta. In caso contrario l'acquisizione del valore produrrà un risultato indefinito (per esempio un errore, la memorizzazione di una rappresentazione errata, ecc.).

Un secondo aspetto riguarda la necessità di *sincronizzare* l'escuzione dell'algoritmo con l'ambiente esterno. Può infatti accadere che, quando l'esecutore esegue le operazioni di acquisizione di un valore dallo standard input, questi sia vuoto, sia perchè il dispositivo non ha ancora generato alcun valore, sia perchè i valori generati fino a quel momento sono stati tutti già acquisiti. In questo caso, l'esecutore si blocca in attesa che venga generato un valore e l'esecuzione dell'algoritmo riprende solo dopo che si sia verificata tale eventualità. Se per qualche ragione dall'esterno non vengono generati nuovi valori l'esecutore rimane bloccato indefinitamente *non producendo alcun risultato*.

4.6 Lo standard output

Lo standard output è l'entità dell'esecutore atteaverso cui le informazioni possono essere inviate all'esterno. Analogamente al caso dello standard input, le informazioni sono trasferite all'esterno dell'esecutore attraverso dispositivi fisici che prendono il nome di *periferiche di output*. Esempi di tali periferiche sono: il video, le porte serale, parallela e USB, il floppy disk, l'hard disk. Si noti come molti dispositivi siano, per la loro natura, allo stesso tempo periferiche di input e di output.

Durante l'esecuzione di un algoritmo, uno di questi dispositivi è connesso con lo standard output, e i valori generati dall'interprete vengono trasferiti alla periferica collegata attraverso lo standard output uno dopo l'altro, nell'ordine con cui sono stati generati. Per questo motivo, lo standard output può essere immaginato come una sequenza di valori generata dall'esecutore verso l'esterno.

Nella figura 4.4 sono mostrati in forma grafica gli aspetti principali del funzionamento dello standard output. L'interprete, eseguendo una istruzione di output genera un valore (il valore può anche essere semplicemente letto da una variabile in memoria) e lo trasferisce all'esterno attraverso lo standard output (figura 4.4-a). Una volta trasferito all'esterno, il valore viene aggiunto allo standard output in modo da formare una sequenza accessibile all'esterno dell'esecutore secoondo modalità che dipendono dalla natura della periferica utilizzata (figure 4.4-b e 4.4-c).

Da quanto detto, l'operazione che l'interprete esegue sullo standard output può essere descritta nel modo seguente:

il valore generato dall'interprete viene aggiunto allo standard output.

Le operazioni sullo standard output devono pertanto semplicemente indicare il valore da aggiungere, o meglio come l'interprete deve generare tale valore. Vedremo nel paragrafo 4.7 che questo può essere fatto attraverso un'espressione.

A differenza di quanto accade per le operazioni di input, il buon esito delle operazioni di output è molto meno dipendente dall'ambiente esterno all'esecutore. In tali operazioni, infatti, l'attività è tutta svolta dall'esecutore e, in particolare:

- il valore prodotto viene immediatamente trasferito sullo standard output provvedendo automaticamente ad una modifica di rappresentazione, se richiesta dalla natura della periferica;
- non vi sono problemi di sincronizzazione in quanto il valore viene aggiunto allo standard output indipendentemente dal fatto che qualcuno lo utilizzi una volta trasfrito all'esterno.

4.7 Descrizione delle operazioni di calcolo

Come abbiamo detto l'interprete è in grado di generare nuovi valori a partire da valori precedentemente acquisiti o generati. Il linguaggio di programmazione permette di descrivere quali valori generare e in che modo generarli attraverso la scrittura di *espressioni*.

Un'espressione è pertanto una notazione che descrive la generazione di un valore attraverso un opportuno meccanismo. È opportuno distinguere tra espressioni *semplici* ed espressioni *composte*, e tra espressioni *costanti* e *non costanti*. Le due classificazioni sono indipendenti per cui si possono avere: espressioni costanti semplici, espressioni costanti composte, espressioni non costanti semplici ed espressioni non costanti composte.

4.7.1 Espressioni costanti semplici

Le espressioni costanti semplici, indicate anche comunemente col termine *costanti*, sono notazioni che specificano direttamente un valore. Le costanti possono essere numeri interi o reali, nel qual caso si usano sequenze di cifre composte con alcuni caratteri speciali secondo le usuali convenzioni, valori logici, caratteri singoli o stringhe di caratteri. In tabella 4.1 sono ripotati esempi di costanti appartenenti a questi cinque tipi. In particolari linguaggi possono essere previsti anche altri tipi di costanti, ma quelli illustrati nella tabella sono comuni a tutti i linguaggi, eventualmente con qualche piccola differenza nella notazione utilizzata.

4.7.2 Espressioni costanti composte

Le espressioni costanti composte, indicate anche comunemente col termine *espressioni costanti*, sono espressioni costituite da più parti.

L'elemento che consente di formare espressioni costanti è sempre un *operatore*, cioè un simbolo che rappresenta una legge che trasforma un numero finito di valori detti *argomenti* in un valore detto *risultato*. La nozione di operatore coincide pertanto con la nozione matematica di funzione nel suo significato più generale.

Ad esempio, se il simbolo + rappresenta l'operatore binario di somma tra numeri, l'espressione:

```
1254 + 36
```

rappresenta il valore `1290`, che è il risultato che si ottiene sommando i numeri interi `1254` e `36` secondo le normali regole aritmetiche.

Il significato e la struttura di un'espressione costante dipende dall'operatore usato per costruirla. L'operatore determina infatti:

- il numero degli argomenti e il loro tipo (se sono numeri, valori logici, caratteri, ecc.);

Costante	Tipo	Note
`1343`	intero	numero intero positivo
`-254`	intero	numero intero negativo
`+136.098`	reale	numero reale, il segno + può essere omesso
`-0.001`	reale	numero reale, la parte intera può essere omessa
`-105.00`	reale	numero reale, la parte decimale può essere omessa
true	logico	valore logico “vero”
false	logico	valore logico “falso”
`'a'`	singolo carattere	la prima lettera minuscola dell’alfabeto inglese
`' '`	singolo carattere	lo spazio bianco
`'!'`	singolo carattere	il punto esclamativo
`'2'`	singolo carattere	la terza cifra decimale
`"cane"`	stringa di caratteri	una singola parola
`"Paolo e' al mare"`	stringa di caratteri	una breve frase
`"Q"`	stringa di caratteri	una stringa formata da un singolo carattere
`""`	stringa di caratteri	la stringa *vuota* (o *nulla*)

Tabella 4.1: Esempi di espressioni costanti semplici.

- il valore e il tipo del risultato;
- la forma testuale dell’espressione.

Nella tabella 4.2 sono riportati esempi di espressioni costanti che usano operatori di diversa natura. Il significato dei simboli usati nelle espressioni di tipo logico è spiegato nelle tabelle 4.4 e 4.5.

Si noti che l’argomento di un operatore (che è un valore) può essere sia una costante (cioè un’espressione semplice) sia ancora un’espressione composta. In questo secondo caso, le regole di precedenza nell’applicazione degli operatori o l’impiego esplicito di parentesi determinano l’ordine con cui devono essere applicati i deversi operatori che compaiono nell’espressione per giungere al risultato finale, che è poi il valore rappresentato dall’intera espressione.

In ogni caso comunque perchè un’espressione composta sia costante, è necessario che gli argomenti di tutti gli operatori siano a loro volta espressioni costanti (sia semplici che composte) in modo da rendere possibile il calcolo del risultato a partire dal testo dell’espressione.

4.7.3 Espressioni non costanti semplici

Le espressioni non costanti semplici sono espressioni formate unicamente dall’identificatore di una variabile e rappresentano il valore contenuto nella variabile *nel momento in cui l’espressione viene presa in considerazione dall’esecutore*.

Il valore di tali espressioni dunque, a differenza di quanto accade con le espressioni costanti, non può essere determinato a partire dal testo dell’espressione. Tale testo infatti indica solo *come fa l’interprete a ottenere il valore dell’espressione*: leggendo la cella di memoria associata all’identificatore indicato nell’espressione. Il valore è determinato dal contenuto della cella in quel momento e dal tipo associato ad essa.

Ad esempio, se `X` è l’identificatore associato ad una variabile che contiene la stringa di bit `01101101`, l’espressione non costante semplice:

```
X
```

Espressione	risultato	Note
`1254 + 36`	1290	operatore di somma, due argomenti entrambi interi, risultato intero
`1254.03 - 36.807`	1217.223	operatore di sottrazione, due argomenti entrambi reali, risultato reale
`1254.03 * -36`	45145.08	operatore di moltiplicazione, due argomenti uno reale e l'altro intero, risultato reale
`15.091 + 12.3 * 3`	51.991	operatore di somma, il secondo argomento è a sua volta un'espressione composta con l'operatore di moltiplicazione, risultato reale
`(15.091 + 12.3) * 3`	82.173	operatore di somma e uso delle parentesi per specificare l'ordine di applicazione degli operatori, il primo argomento è a sua volta un'espressione composta con l'operatore di somma, risultato reale
`0 <= 7`	**true**	operatore di relazione minore o uguale, due argomenti dello stesso tipo ordinato, risultato logico
false `&&` **true**	**false**	operatore logico AND, due argomenti di tipo logico, risultato logico
`3 != 4` `||` **false**	**true**	operatore logico OR, usoe delle regole di precedenza per specificare l'ordine di applicazione degli opertori, il primo argomento è a sua volta un'espressione composta con l'operatore di relazione di disuguaglianza, risultato logico

Tabella 4.2: Esempi di espressioni costanti composte.

ha valore `109` se il tipo della variabile corrisponde a una rappresentazione in complemento a 2 su 8 bit di numeri relativi. Se invece il tipo di `X` corrispondesse alla rappresentazione di caratteri secondo il codice ASCII, il valore dell'espressione sarebbe `'m'`, cioè la tredicesima lettera minuscola dell'alfabeto inglese.

Si noti che se `X` contenesse la stringa `00100001`, il valore della stessa espressione sarebbe `33`, nel caso di rappresentazione in complemento a 2 su 8 bit di numeri relativi, e il carattere punto esclamativo nel caso di rappresentazione di caratteri secondo il codice ASCII.

4.7.4 Espressioni non costanti composte

Le espressioni non costanti composte, infine, sono espressioni costituite da parti, analogamente alle espressioni costanti composte. La differenza è che, nel caso di espressioni non costanti composte, gli identificatori delle variabili (che sono espressioni non costanti semplici) possono essere usati come operandi degli operatori presenti nell'espressione.

Le espressioni non costanti composte sono il tipo più generale di espressione e possono avere strutture anche molto complesse. Tuttavia anche nei casi più complessi le espressioni non costanti composte vengono sempre formate a partire da uno o più operatori. In particolare, il significato e la struttura di un'espressione dipende dall'operatore usato per costruirla che, ricordiamo, determina:

- il numero degli argomenti e il loro tipo;
- il valore e il tipo del risultato;
- la forma testuale dell'espressione.

Per quanto riguarda l'ultimo punto è opportuno distinguere tra operatori *prefissi* e operatori *infissi*. I primi sono quelli per i quali deve essere usata una notazione in cui il simbolo dell'operatore precede gli argomenti che sono elencati tra parentesi e separati da virgole. Ad esempio, l'operatore `pow`, che ha due argomenti reali e che calcola il valore reale che si ottiene elevando il primo argomento e al secondo, è di tipo prefisso per cui può essere usato per costruire la seguente espressione:

```
pow(x,2.0)
```

dove `x` è il nome di una variabile che contiene un numero reale. Gli operatori infissi sono quelli in cui il simbolo dell'operatore viene collocato tra gli argomenti. Il caso tipico è quello degli operatori binari aritmetici (somma, sottrazione, ecc.) per i quali il simbolo viene scritto tra i due operandi.

Il valore di un'espressione non costante composta, analogamente al caso delle espressioni non costanti semplici, dipende dal valore che hanno le variabili presenti in essa nel momento in cui l'esecutore la prende in considerazione. Con riferimento all'esempio precedente, se la variabile `x` contiene il valore `5.02` quando l'espressione viene valutata, il valore di `pow(x,2.0)` è `25.2004`.

4.7.5 Commenti riepilogativi sulle espressioni

Le espressioni sono l'unico strumento per descrivere la trasformazione di informazioni in un linguaggio di programmazione. Come abbiamo visto il principio di costruzione delle espressioni è molto semplice ed è sostanzialmente riconducibile al concetto di funzione matematica. Tale apparente semplicità non deve tuttavia trarre in inganno, perchè in realtà è possibile formare espressioni molto complesse sia dal punto di vista strutturale (numero e varietà degli operatori e degli operandi), sia dal punto di vista del *significato*, cioè delle regole che l'esecutore deve seguire per calcolarne il valore.

Il punto cruciale è che le espressioni possono essere formate componendo ripetutamente operatori diversi e usando anche espressioni composte come operandi per formare nuove espressioni composte più complesse. In questo procedimento occorre seguire le regole di composizione di ciascun operatore (numero e tipo degli operandi, tipo del risultato e forma testuale dell'operatore), e tali regole determinano il significato dell'espressione finale.

Il numero degli operatori disponibili in un qualunque linguaggio di programmazione è solitamente molto elevato, e sebbene in genere tutti gli operatori abbiano un numero limitato di operandi (quasi sempre uno o due), essi sono spesso caratterizzati da molteplici varianti che pur differendo nella forma testuale per aspetti marginali, possono avere un significato anche molto diverso.

Tutto questo rende il linguaggio delle espressioni molto flessibile e capace di descrivere un numero sconfinato di trasformazioni utili, ma comporta allo stesso tempo la necessità di acquisire un numero notevole di nozioni per poterlo usare con efficacia.

Sebbene non sia possibile in modo sintetico fornire tutte le nozioni importanti per costruire nei vari linguaggi di programmazione espressioni corrette sia sotto il profilo strutturale, sia sotto quello del siginificato, è possibile tuttavia indicare sinteticamente alcuni principi generali che possono guidare nell'applicazione delle regole di composizione delle espressioni riportate dettagliatamente sul manuale di ciascun linguaggio specifico:

1. Ciascuna espressione è la descrizione di come generare un valore che è detto il suo risultato; un'espressione pertanto rappresenta un valore e nel seguito useremo il termine *denotare* per indicare tale associazione tra un'espressione e il suo risultato.

2. Il procedimento attraverso cui l'esecutore genera il valore denotato da un'espressione si dice *valutazione* o *calcolo* dell'espressione.

3. Ciascuna espressione *ha* un tipo, che è il tipo del valore che essa denota.

4. La valutazione di un'espressione semplice, e di conseguenza la deretminazione del suo tipo, è banale.

5. Ciascuna espressione composta contiene uno o più operatori; la valutazione di un'espressione composta e la conseguente determinazione del suo tipo richiede l'applicazione delle regole associate a ciascun operatore che compare nell'espressione; parentesi e regole di precedenza devono essere usate per stabilire l'ordine con cui devono essere considerati gli operatori nel caso ce ne sia più di uno.

6. In un'espressione composta, a ciascun operatore corrisponde una *sotto espressione*, cioè una parte dell'espressione, che comprende: il simbolo dell'operatore, tutti i suoi operandi, eventuali caratteri ausiliari (ad esempio le parentesi e le virgole nel caso di operatori prefissi, le parentesi che indicano precedenza, ecc.).

7. La sotto espressione corrispondente ad ogni operatore deve essere *ben formata*, occorre cioè che il numero e il tipo degli operandi corrisponda alle regole che caratterizzano l'operatore; l'applicazione di tali regole determinano come viene valutata la sotto espressione e quale è il tipo associato ad essa; le regole possono ammettere varianti e ciascuna variante ha una sua forma testuale caratteristica che può comportare differenze tra gli operandi e i risultati.

8. La valutazione dell'intera espressione deve essere fatta partendo dagli operatori che devono essere considerati per primi e applicando ad essi le regole di valutazione; una volta valutati i valori di tali sotto espressioni e determinati i loro tipi si ripete il procedimento per gli operatori che devono essere valutati successivamente, fino a giungere all'ultimo operatore da valutare che fornisce il risultato dell'espressione e il suo tipo.

Come si è detto al punto 7, per poter applicare le regole che caratterizzano ciascun operatore, ogni parte dell'espressione deve essere ben formata. In tal caso l'espressione denota un valore e ha un tipo. Nel caso invece che l'espressione non sia ben formata, e cioè nel caso in cui vi siano operatori che non hanno il numero di operandi richiesto, o che hanno operandi di tipo non previsto, o la cui forma testuale non sia tra quelle previste, l'espressione contiene degli errori e *non* può essere valutata.

Diamo ora alcuni esempi di applicazione dei principi sopra esposti nella comprensione del significato di alcune semplici espressioni.

In tabella 4.3 sono riportati i principali operatori binari (somma, sottrazione, prodotto, divisione) che operano su informazioni di tipo numerico. Si noti che nel caso della divisione fra operandi entrambi interi (divisione intera) sono

Operatore	numero e tipo degli operandi	tipo e valore del risultato	Note
+	2 operandi di tipo intero	il risultato è di tipo intero ed è la somma algebrica degli operandi	può prodursi overflow se il risultato non è rappresentabile
+	2 operandi di cui almeno uno di tipo reale	il risultato è di tipo reale ed è la somma algebrica degli operandi	può prodursi overflow se il risultato non è rappresentabile o underflow se viene approssimato a 0
-	2 operandi di tipo intero	il risultato è di tipo intero ed è la somma algebrica del primo operando con il secondo cambiato di segno	può prodursi overflow se il risultato non è rappresentabile
-	2 operandi di cui almeno uno di tipo reale	il risultato è di tipo reale ed è la somma algebrica del primo operando con il secondo cambiato di segno	può prodursi overflow se il risultato non è rappresentabile o underflow se viene approssimato a 0
*	2 operandi di tipo intero	il risultato è di tipo intero ed è il prodotto degli operandi	può prodursi overflow se il risultato non è rappresentabile
*	2 operandi di cui almeno uno di tipo reale	il risultato è di tipo reale ed è il prodotto degli operandi	può prodursi overflow se il risultato non è rappresentabile o underflow se viene approssimato a 0
/	2 operandi di tipo intero	il risultato è di tipo intero ed è il *quoto* della divisione intera tra gli operandi	non può prodursi overflow
%	2 operandi di tipo intero	il risultato è di tipo intero ed è il *resto* della divisione intera tra gli operandi	non può prodursi overflow
/	2 operandi di cui almeno uno di tipo reale	il risultato è di tipo reale ed è il quoziente degli operandi	può prodursi overflow se il risultato non è rappresentabile o underflow se viene approssimato a 0

Tabella 4.3: Esempi di espressioni costanti composte.

previsti due operatori, uno che calcola il quoto e uno che calcola il resto. In molti linguaggi il simbolo `/` è usato sia per il calcolo del quoto della divisione intera, sia per il calcolo del quoziente della divisione tra reali. Il siginificato di un'espressione che contiene tale simbolo dipende quindi dal tipo dei suoi operandi ed è necessario applicare con attenzione le regole di valutazione come suggerito dai principi generali sopra riassunti (cfr. in particolare esempi 4.2 e 4.3.

Esempio 4.1

Valutare l'espressione:

```
-9.8 * sqr(T) + 25 * T + 12.4
```

assumendo che la variabile `T` sia di tipo reale e abbia valore `1.0`.

Gli operatori vanno applicati nel seguente ordine:

1. `sqr(T)` che ha come risultato `1.0`;
2. `-9.8 * 1.0` e `25 * 1.0` in un ordine qualsiasi, che hanno come risultato `-9.8` e `25.0`, rispettivamente;
3. `-9.8 + 25.0 + 12.4` che ha come risultato `27.6` (in realtà si tratta di due operatori di somma distinti che per la proprietà associativa possono essere applicati in un ordine qualunque).

Si noti che nell'esempio il tipo degli operandi non gioca un ruolo particolare in quanto le regole che caratterizzano gli operatori considerati coincidono con quelle dei corrispondenti operatori aritmetici.

Esempio 4.2

Valutare l'espressione:

```
3.6 * (2 / 3)
```

Le parentesi specificano che gli operatori vanno applicati nel seguente ordine:

1. `2 / 3` che ha come risultato il numero intero `0` in quanto entrambi gli operandi sono interi;
2. `3.6 + 0` che ha come risultato `3.6`.

Si noti che questa volta la presenza di entrambi gli operand interi attribuisce al simbolo `/` il significato di divisione intera.

Esempio 4.3

Valutare l'espressione:

```
(3.6 * 2) / 3
```

Le parentesi specificano che gli operatori vanno applicati nel seguente ordine:

1. `3.6 * 2` che ha come risultato il numero reale `7.2`;
2. `7.2 / 3` che ha come risultato `2.4`, essend il primo operando un numero reale.

operatore	notaz. cap. 2	notaz. linguaggio prog.
uguale	$=$	`==`
diverso	$\neq$	`!=`
maggiore	$>$	`>`
mibìnore	$<$	`<`
maggiore o uguale	$\geq$	`>=`
minore o uguale	$\leq$	`<=`

Tabella 4.4: Simboli usati nel linguaggio di programmazione per gli operatori di relazione.

operatore	notaz. cap. 2	notaz. linguaggio prog.
not	NOT	`!`
not	AND	`&&`
not	OR	`||`

Tabella 4.5: Simboli usati nel linguaggio di programmazione per gli operatori logici.

Si noti come questa volta il differente ordine di applicazione degli operatori produca un risultato differente perchè la presernza di un operando reale attribuisce al simbolo `/` il significato di divisione tra reali.

È anche interessante notare che, da un punto di vista puramente matematico, l'espressione `3.6 * 2 / 3` non richiederebbe la specificazione delle parentesi e avrebbe in ogni caso come risultato il numero `2.4`. L'apparente paradosso mette in luce come occorra fare attezione all'interpretazione delle espressioni in un linguaggio di programmazione a motivo del fatto che esse sono in realtà solo la descrizione del procedimento che deve seguire l'esecutore automatico per produrre il risultato.

Nelle tabelle 4.4 e 4.5 sono riportati rispettivamente i principali operatori di relazione e logici usati nel linguaggio per descrivere condizioni. Per ogni operatore, nella seconda colonna è riportato il simbolo usato nel capitolo 2 e nella terza il simbolo usato nel linguaggio di programmazione. Tutti gli operatori sono già stati introdotti nel capitolo 2 e, a parte le differenze di notazione, tutte le proprietà di questi operatori sono quelle già discusse. In particolare si ricordi che gli operatori di relazione hanno precedenza su quelli logici.

Esercizi proposti

1. Valutare l'espressione:

   ```
   x + (z/10 - 1) * 2.0
   ```

   assumendo che le variabili abbiano il tipo e il valore riportato nella seguente tabella:

Identificatore	Tipo	Valore
`x`	reale	`23.09`
`z`	intero	`25`

2. Riscrivere le espressioni logiche del capitolo 2 usando i simboli del linguaggio di programmazione riportati nelle tabelle 4.4 e 4.5

Capitolo 5

Costrutti base del linguaggio di programmazione (bozze, v. 2.0)

In questo capitolo illustreremo con maggiore dettaglio le operazioni elementari che il linguaggio di programmazione permette di esprimere e le modalità con cui tali operazioni possono essere composte in modo da formare un'algoritmo.

Per la corretta comprensione del capitolo devono essere conosciute le seguenti nozioni: dati di ingresso e di uscita di un algoritmo, sequenza statica e dinamica di un algoritmo, valore e attributo di un'informazione, rappresentazione e tipo di un'informazione.

5.1 Redazione di programmi

Per per formulare un algoritmo in forma effettivamente eseguibile, occorre aggiungere alle istruzioni vere e proprie delle "istruzioni" particolari che servono, per esempio, a definire con precisione dove inizia e dove termina il programma, a specificare che il programma fa uso di particolari istruzioni di input e di output, ecc.

Prima di procedere a introdurre le istruzioni concrete che devono essere usate per programmare la macchina astratta è pertanto opportuno introdurre alcune di queste istruzioni aggiuntive in modo da essere subito in grado di scrivere programmi effettivamente utilizzabili. La descrizione che segue ha carattere essenzialmente pratico e prescrittivo. Una spiegazione completa della natura e del siginificato delle istruzioni introdotte verrà fornita successivamente.

Un programma completo consiste in una sequenza di istruzioni terminate dal carattere `;` (punto e virgola), racchiusa tra parentesi graffe e preceduta dagli identificatori **int** `main`. In fondo alla sequenza di istruzioni deve poi essere aggiunta l'istruzione:

```
return 0;
```

anch'essa terminata dal carattere `;`. Un programma ha pertanto la seguente forma generale:

```
int main () {
   istruzione 1;
   istruzione 2;
      ...
   istruzione n;
   return 0;
}
```

Si noti la disposizione delle parentesi graffe e l'allineamento della sequenza di istruzioni leggermente sfalsate rispetto all'inizio della prima e dell'ultima riga. Questa disposizione del testo non è casuale. La disposizione più

avanzata delle istruzioni viene detta *indentazione* e ha lo scopo di mettere in evidenza, anche dal punto di vista di chi legge il programma, quale sia la lista delle istruzioni e che esse sono interne al *blocco* delimitato dalle parentesi graffe. Anche l'allineamento verticale della parentesi graffa chiusa con l'inizio della prima riga ha lo scopo di aumentare l'impatto visivo della struttura del programma.

In tutti gli esempi riportati in questo testo, si farà largo uso dell'indentazione e di altri accorgimenti di tipo grafico per rendere sempre il più evidente possibile le relazioni tra le istruzioni e la struttura del programma. Sebbene tali accorgimenti siano di tipo convenzionale (il siginificato del programma non dipende da tali accorigmenti che possono essere omessi in tutto o in parte) e possano differire da programmatore a programmatore, tuttavia essi sono necessari per scrivere programmi facilmete leggibili da parte di operatori umani e non devono mai essere omessi.

Un ultima aggiunta allo schema generale di programma appena illustrato, consiste nel premettere ad esso la frase:

```
# include <iostream.h>
```

tutte le volte (in pratica sempre nel caso dei semplici esempi che considereremo nel seguito) che il programma fa uso di istruzioni di input e/o di output.

5.2 Istruzione di output

L'*istruzione di output* ha la forma:

`cout <<` *espressione*`;`

dove *espressione* deve essere un'espressione ben formata.

L'effetto di un istruzione di output è quello di generare il valore denotato dall'espressione, rappresentato secondo il tipo dell'espressione stessa, e di aggiungerlo allo standard output.

Spesso tale trasferimento viene indicato con il termine *stampa*[1] perchè in passato lo standard output era tipicamente associato ad un dispositivo di stampa su supporto cartaceo. Nei moderni sistemi di elaborazione lo standard output è invece normalmente associato ad una finestra del dispositivo grafico di visualizzazione, che consiste in un numero finito di righe (per es. 25), ciascuna contenenti un numero finito di caratteri (per es. 80).

Osservazione 5.1

Con riefrimento al concetto di uso di una variabile introdotto nel paragrafo 4.4, in una istruzione di output le variabili che compaiono nell'espressione vengono *usate*.

Esempio 5.1

Il programma:

```
# include <iostream.h>

int main () {
  cout << 1254 + 36;
  return 0;
}
```

aggiunge allo standard output il valore 1290.

Esempio 5.2

[1] Per esempio è tipica una frase del tipo "l'istruzione (di output) stampa il risultato".

Il programma:

```
# include <iostream.h>

int main () {
  cout << (15.091 + 12.3) * 3;
  return 0;
}
```

aggiunge allo standard output il valore 82.173.

Esempio 5.3

Il programma:

```
# include <iostream.h>

int main () {
  cout << 'a';
  return 0;
}
```

aggiunge allo standard output il valore corrispondente alla prima lettera minuscola dell'alfabeto inglese.

Esempio 5.4

Il programma:

```
# include <iostream.h>

int main () {
  cout << "Paolo e' al mare";
  return 0;
}
```

aggiunge allo standard output il valore corrispondente alla frase:

Paolo e' al mare

Esempio 5.5

Il programma:

```
# include <iostream.h>

int main () {
  cout << "Prima riga";
  cout << endl;
  cout << "Seconda riga";
  cout << endl;
  return 0;
}
```

Tipo	Insieme dei valori	rappresentazione
bool	valori logici appartenenti all'insieme {**false**,**true**}	**false** è rappresentato da una stringa nulla
unsigned	numeri naturali appartenenti all'intervallo $[0, N-1]$ (valori tipici di N: $2^{16} = 65536$, $2^{32} = 4294967296$	rappresentazione posizionale
int	numeri relativi appartenenti all'intervallo $[-Z, Z-1]$ (valori tipici di Z: $2^{15} = 16384$, $2^{31} = 2147483648$	rappresentazione in complemento a 2
char	caratteri comprendenti: cifre decimali, lettere maiuscole e minuscole dell'alfabeto inglese, segni di interpunzione e altri caratteri speciali per un totale di 256 simboli	rappresentazione ASCII
float	numeri reali appartenenti a un intervallo $[-2^{128}, -2^{128}]$	rappresentazione IEEE 754 (singola precisione)
double	numeri reali appartenenti a un intervallo $[-2^{1024}, -2^{1024}]$	rappresentazione IEEE 754 (doppia precisione)

Tabella 5.1: I principali tipi primitivi.

aggiunge allo standard output il valore corrispondente alle frasi:

Prima riga
Seconda riga

su due linee diverse, andando a capo dopo la seconda frase. L'andata a capo è prodotta aggiungendo allo standard output un carattere speciale di "fine linea", che si indica con l'identificatore `endl`.

5.3 Creazione di variabili

Come abbiamo detto nel paragrafo 4.4, le variabili devono essere esplicitamente create. Per farlo il linguaggio prevede apposite istruzioni che vengono dette *dichiarazioni*.

La dichiarazione di una variabile ha la forma generale:

tipo variabile ;

dove *tipo* è un identificatore che specifica un tipo primitivo o definito dall'utente, e *variabile* è un identificatore non usato in precedenza per altri scopi (questo vincolo non è in realtà necessario, ma per il momento assumeremo per semplicità che lo sia).

L'effetto della dichiarazione è quello che viene creata una cella di memoria associata al tipo e all'identificatore specificati.

Ad esempio, assumendo di disporre dei tipi primitivi elencati nella tabella 5.1, la dichiarazione:

int `x;`

aggiunge alla memoria una cella identificata dal nome `x` e in grado di contenere la rappresentazione di un intero relativo in complemento.

5.4 Istruzione di input

L'*istruzione di input* ha la forma:

```
cin >> variabile;
```

dove *variabile* deve essere l'identificatore di una variabile creata da una precedente dichiarazione.

L'effetto di un istruzione di input è quello di estrarre il primo valore presente nello standard input e di memorizzarlo nella variabile specificata. Il valore viene memorizzato usando la rappresentazine corrispondente al tipo della variabile. Se il valore letto non è compreso nell'insieme dei valori corrispondente al tipo della variabile, il risultato della memorizzazione è indefinito. Se lo standard input è vuoto, l'istruzione di input rimane sospesa indefinitamente fino a che non viene aggiunto un valore allo standard input tramite la periferica collegata.

Osservazione 5.2

Con riferimento al concetto di definizione di una variabile introdotto nel paragrafo 4.4, in una istruzione di input la variabile che compare in un'istruzione di input viene *definita*.

Esempio 5.6

Il programma:

```
# include <iostream.h>

int main () {
  int x;
  cin >> x;
  cout << 2 * x;
  return 0;
}
```

crea una variabile `x` di tipo **int**, estrae il primo valore dallo standard input e ne trasferisce la rappresentazione in complemento a 2 in `x`, aggiune allo standard output il doppio del valore trasferito in `x`. Per esempio, se il primo valore presente sullo standard input è 123, allo standard output viene aggiunto il valore 246.

Esempio 5.7

Il programma:

```
# include <iostream.h>

int main () {
  double x;
  double y;
  cin >> x;
  cin >> y;
  cout << x * y;
  cout << x / y;
  return 0;
}
```

crea due variabili di tipo **double**, estrae due valori dallo standard input e trasferisce la loro rappresentazione in formato IEEE 745 (doppia precisione) nelle due variabili, aggiune allo standard output il prodotto dei due valori estratti seguito dal loro rapporto. Per esempio, se i valori presenti sullo standard input sono 5.24 e 2, allo standard output vengono aggiunti (nell'ordine) i valori 10.48 e 2.62.

Esercizi proposti

1. Scrivere un programma che legge un numero dallo standard input e lo stampa sullo standard output preceduto dalla didascalia:

   Il numero letto è:

2. Scrivere un programma che legge due numeri interi dallo standard input e stampa sullo standard output il quoto e il resto della divisione intera del primo numero per il secondo.

5.5 Istruzione di assegnazione

Abbiamo già spiegato come l'interprete sia in grado di generare nuovi valori calcolando il risultato di espressioni. Abbiamo anche visto come i valori generati dalla valutazione di espressioni possano essere trasferiti all'esterno dell'esecutore mediante le istruzioni di output.

Ci occupiamo ora di un altro tipo di istruzione che contiene espressioni: l'istuzione di *assegnazione*. Analogamente all'istruzione di output, anche l'istruzione di assegnazione comprende un'espressione, ma il suo effetto, oltre alla valutazione dell'espressione, è quello di trasferire il valore calcolato dall'interprete in una variabile in memoria.

La forma generale di un'istruzione di assegnazione è la seguente:

variabile = *espressione*;

dove *variabile* rappresenta l'identificatore di una variabile creata da una dichiarazione precedente, il simbolo = rappresenta *l'operatore di assegnazione*, e *espressione* è un'espressione. Il tipo della variabile a sinistra dell'operatore di assegnazione e dell'espressione alla sua destra devono coincidere.

Quando l'esecutore esegue un'istruzione di assegnazione, l'interprete compie nell'ordine le seguenti operazioni:

- valuta l'espressione a destra dell'operatore di assegnazione;
- sulla base del tipo dell'espressione genera la rappresentazione del risultato ottenuto;
- trasferisce tale rappresentazione nella variabile a sinistra dell'operatore di assegnazione.

Per chiarire quanto detto presentiamo alcuni esempi. Se la variabile `X` è di tipo **int** l'istruzione:

```
X = 23;
```

valuta l'espressione di tipo **int** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in complemento) del valore ottenuto (il numero ventitrè) nella variabile `X`. Se la variabile `w` è di tipo **char** l'istruzione:

```
w = 'a';
```

valuta l'espressione di tipo **char** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (ASCII) del valore ottenuto (la prima lettera minuscola) nella variabile `w`. Se le variabili `y` e `z` sono di tipo **int**, e il valore di `z` è 514, l'istruzione:

```
y = z;
```

valuta l'espressione di tipo **int** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in complemento) del valore ottenuto (il numero 514) nella variabile `y`. Se le variabili `alfa`, `x` e `y` sono di tipo **double**, il valore di `x` è 16.56, e il valore di `y` è -5.011, l'istruzione:

```
alfa = x + y;
```

valuta l'espressione di tipo **double** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in virgola mobile, doppia precisione) del valore ottenuto (il numero 11.549) nella variabile `alfa`. Infine, se la variabile `x` è di tipo **unsigned** e il suo valore è 10, l'istruzione:

```
x = x + 1;
```

valuta l'espressione di tipo **unsigned** a destra dell'operatore di assegnazione e trasferisce la rappresentazione (in rappresentazione posizionale) del valore ottenuto (il numero 11) nella stessa variabile `x`.

Nell'ultimo caso è evidente come, nel linguaggio dell'esecutore, il simbolo `=` non vada confuso con l'operatore di uguaglianza. L'operatore di assegnazione indica il trasferimento in memoria della rappresentazione di un valore, e dunque, scrivere `x = x + 1` indica semplicemente che:

- viene *usata* la variabile `x`;
- il suo valore (ad esempio 10) viene sommato a 1;
- la rappresentazione del risultato viene trasferita nella stessa variabile `x` *sovrascrivendo evidentemente il precedente valore contenuto in* `x`.

Osservazione 5.3

Con riferimento ai concetti di definizione e uso di una variabile introdotti nel paragrafo 4.4, in una assegnazione le variabili che compaiono nell'espressione a destra dell'operatore di assegnazione vengono *usate*, mentre la variabile che compare a sinistra dell'operatore di assegnazione viene *definita*. Si noti che, come accade nell'ultimo esempio, una stessa variabile può essere usata e definita nella stessa istruzione di assegnazione. In questo caso la variabile viene prima usata e poi definita.

Osservazione 5.4

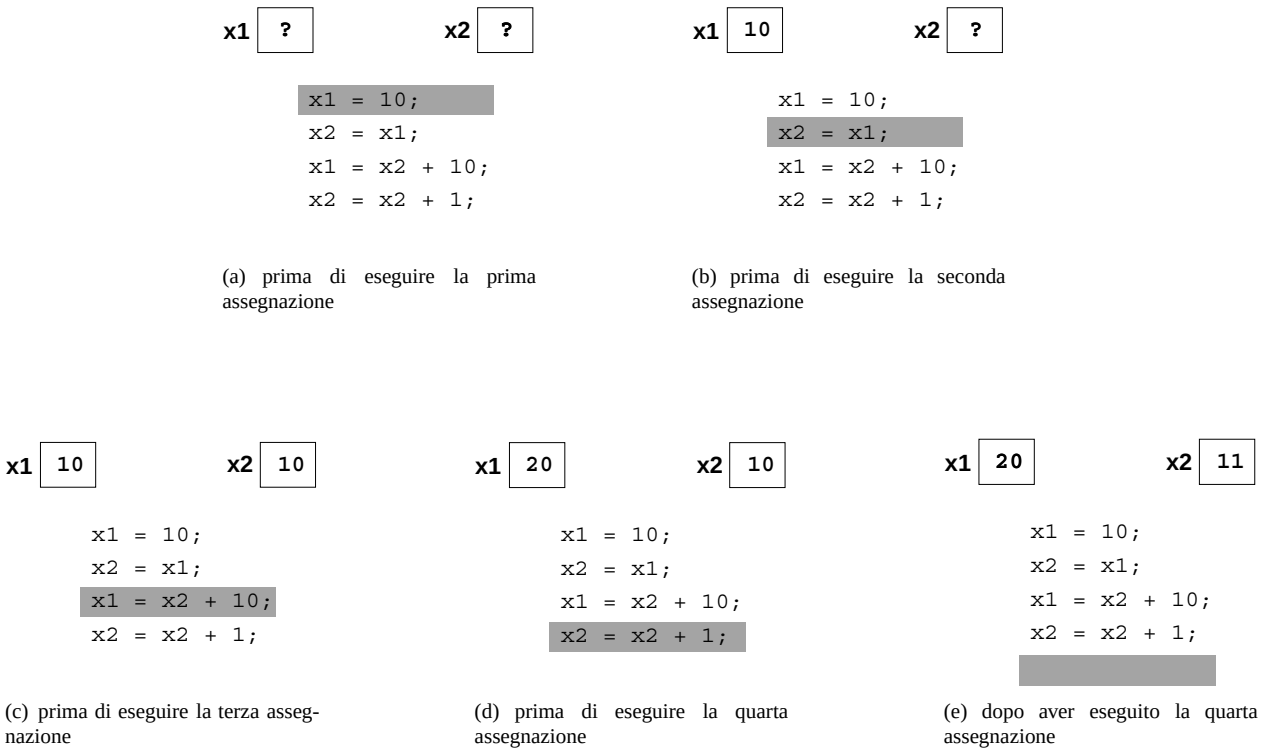
Se il tipo dell'espressione non coincide con il tipo della variabile, la stringa di bit generata, che rappresenta il valore del risultato dell'espressione sulla base del tipo di quest'ultima, viene "convertita" in una stringa di bit che possa essere trasferito nella variabile (che sia cioè coerente con il suo tipo) prima di effettuare il trasferimento. La conversione può consistere in una trasformazione della stringa secondo regole previste per il caso specifico, può consistere nel troncare la stringa o nell'estenderla nel caso fosse troppo lunga o troppo corta, o può consistere semplicemente nel trasferire la stringa senza effettuare modifiche.

Ad esempio, se il tipo dell'espressione fosse **int** e il tipo della variabile fosse **float**, la stinga prodotta, che corrisponde a un numero relativo rappresentato in complemento, viene trasformata in una stringa di opportuna lunghezza che rappresenta lo stesso numero in virgola mobile. Viceversa, se il tipo dell'espressione fosse **int** e il tipo della variabile fosse **unsigned**, la stringa prodotta viene trasferita senza modifiche. Si noti che questo significa che il numero calcolato e quello contenuto nella variabile dopo l'assegnazione sono uguali solo se il primo valore è positivo, altrimenti sono diversi.

Esempio 5.8

Per chiarire ulteriormente il siginificato dell'istruzione di assegnazione, supponendo che `x1` e `x2` siano due variabili di tipo **int** appena dichiarate (e cioè non ancora definite), consideriamo la seguente sequenza di assegnazioni:

```
x1 = 10;
x2 = x1;
x1 = x2 + 10;
x2 = x2 + 1;
```



(a) prima di eseguire la prima assegnazione

(b) prima di eseguire la seconda assegnazione

(c) prima di eseguire la terza assegnazione

(d) prima di eseguire la quarta assegnazione

(e) dopo aver eseguito la quarta assegnazione

Figura 5.1: Effetti prodotti da una sequenza di assegnazioni.

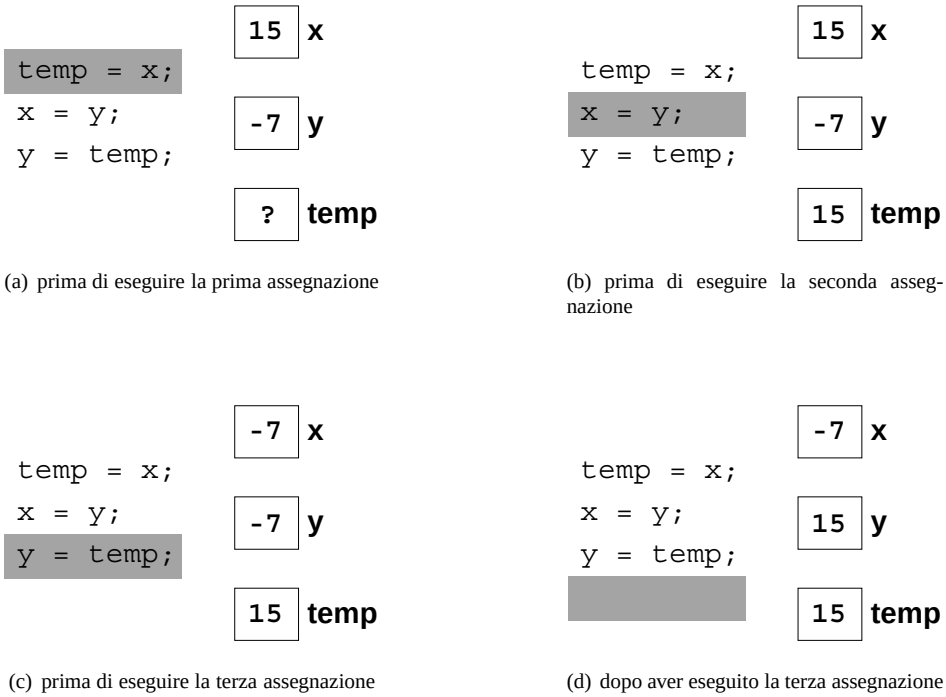


Figura 5.2: Scambio tra due variabili.

In figura 5.1 viene mostrato graficamento l'effetto dell'esecuzione in sequenza delle quattro assegnazioni. Le variabili `x1` e `x2` sono rappresentate mediante dei rettangoli, all'interno dei quali è indicato il loro valore *prima* dell'esecuzione dell'istruzione evidenziata. Il punto interrogativo indicato nella figura 5.1-a all'interno dei due rettangoli sottolinea il fatto che le due variabili sono inzialmente indefinite.

Si noti in particolare come una nuova assegnazione cancelli il valore precedentemente contenuto nella variabile che compare a sinistra dell'operatore di assegnazione.

Esempio 5.9

Un ulteriore esempio di uso dell'assegnazione è quello riportato nella figura 5.2 che mostra come risolvere il seguente problema tipico:

date due variabili definite e dello stesso tipo scambiare i valori in esse contenuti.

Il problema richiede l'impiego di una terza variabile di appoggio dedicata a questo scopo per poter "salvare" il valore di una delle due variabili mentre la si ridefinisce con il valore contenuto nell'altra. Ne deriva la seguente sequenza di assegnazioni in cui, si `x` e `y` sono le variabili di cui si vogliono scabiare i valori e `temp` è la variabile di appoggio:

```
temp = x;
x = y;
y = temp;
```

Si noti che inizialmente la variabile `temp` è indefinita (anche se fosse stata definita precedentemente, ai fini dello scambio il suo valore è irrilevante e dunque, da un punto di vista concettuale, va considerata indefinita). Analogamente

il suo valore finale non è siginificativo perché dipende da quale delle due variabili da scambiare viene salvata in `temp`, cosa che, come abbiamo detto, è irrilevante (se nella sequenza di assegnazioni `x` e `y` si scambiano i ruoli il risultato finale è lo stesso).

Questo comportamento della variabile di appoggio `temp` è tipico come risulterà evidente quando discuteremo le proprietà dei segmenti di programma (cfr. capitolo 6).

Esercizi proposti

1. Riscrivere il programma dell'esempio 5.6, assegnando il risultato dell'espressione a una variabile `ris` e stampando il valore di tale variabile.
2. Riscrivere il programma dell'esempio 5.7, assegnando i risultati delle due espressioni a due variabile `ris1` e `ris2` e stampando i valori di tali variabili.
3. Riscrivere l'esercizio 2, usando una sola variabile `ris` a cui assegnare i risultati delle due espressioni prima di stamparli.
4. Usando 4 assegnazioni e una variabile di appoggio, scrivere una sequenza di istruzioni che, date tre variabili `x`, `y` e `z` dello stesso tipo, porti il valore di `y` in `x`, quello di `z` in `y`, e quello di `x` in `z`.

5.6 Blocco di istruzioni e variabili locali

Nel paragrafo 5.1 sono state introdotte in modo informale la notazione per indicare una sequenza di istruzioni e la nozione di *blocco* di istruzioni. Riprendendo l'argomento in modo sistematico, una sequenza di istruzioni viene indicata elencando le istruzioni da sinistra verso destra e dall'alto verso il basso, terminando ciascuna istruzione con il separatore `;`. Questa notazione indica che l'esecutore esegue una dopo l'altra le istruzioni della sequenza, nell'ordine con cui le istruzioni sono elencate. Ricordiamo che in questo caso la sequenza dinamica di ogni esecuzione coincide con la sequenza statica.

Un *blocco* è una sequenza di istruzioni (incluse eventuali dichiarazioni di variabili) racchiusa tra parentesi graffe. Un blocco di istruzioni può essere usato come un'unica *istruzione composta* per costruire istruzioni più complesse (vedi paragrafi successivi). Eventuali dichiarazioni di variabili all'interno di un blocco producono come sempre la loro creazione da parte dell'esecutore. Tali variabili, dette *locali* al blocco, vengono distrutte dopo che l'esecutore ha completato l'ultima istruzione del blocco. Le variabili locali possono pertanto essere definite o usate solo da istruzioni interne al blocco in cui sono state create.

Esempio 5.10

Il seguente codice causa l'acquisizione di due valori interi dallo standard input e la stampa di due valori interi sullo standard output:

```
{
  int x;
  int y;
  cin >> x;
  {
    cin >> y;
    cout << x+y;
  }
  cout << x-y;
  cout << 2*x;
}
```

Il codice è corretto perché tutte le variabili che le istruzioni definiscono e usano sono locali. Si noti che le variabili locali a un blocco sono locali anche ai blocchi interni ad esso.

Il seguente codice non è corretto perché la variabile `y`, locale al blocco più interno, non esiste al di fuori di tale blocco:

```
{
  int x;
  cin >> x;
  {
    int y;
    cin >> y;
    cout << x+y;
  }
  cout << x-y;
  cout << 2*x;
}
```

Volendo mantenere la variabile `y` locale al blocco più interno bisognerebbe riscrivere il codice nel modo seguente:

```
{
  int x;
  cin >> x;
  {
    int y;
    cin >> y;
    cout << x+y;
    cout << x-y;
  }
  cout << 2*x;
}
```

dove le istruzioni che usano `y` sono state spostate all'interno dello stesso blocco.

5.7 Controllo della sequenza dinamica

Nel capitolo 1 abbiamo visto che in un algoritmo sono normalmente presenti passi decisionali che servono a controllare la particolare sequenza dinamica che deve essere eseguita dall'esecutore per risolvere uno specifico caso di ingresso. Si può dimostrare che per descrivere qualunque algoritmo sono sufficienti solo due tipologie di passi decisionali a cui corrispondono nel linguaggio di programmazione due tipi di istruzioni: l'istruzione di *selezione* e l'istruzione *iterativa*. Si tratta in entrambi i casi di istruzioni composte da istruzioni più semplici che, sulla base del valore di verità di una condizione, consentono rispettivamente di scegliere tra due istruzioni alternative e di ripetere un istruzione. Le due tipologie sono discusse nei dettagli nei due paragrafi successivi.

5.8 Istruzione di selezione

L'istruzione di selezione ha la seguente forma generale:

if (*condizione*)
  parte-then ;
else
  parte-else ;

dove *condizione* è una condizione, mentre *parte-then* e *parte-else* sono due istruzioni qualsisasi (semplici o composte). L'esecuzione di un'istruzione di selezione produce nell'ordine:

1. la valutazione della condizione;
2. l'esecuzione della parte-then se il valore logico della condizione è **true** oppure l'esecuzione della parte-else se il valore logico della condizione è **false**.

Si noti che l'istruzione di selezione produce l'esecuzione *solo* della parte-then oppure *solo* della parte-else.

Esempio 5.11

Se le variabili `a` e `b` hanno lo stesso valore, l'istruzione:

```
if ( a == b )
  x = 0;
else
  x = 1;
```

causa prima la valutazione della condizione `( a == b )`, che risulta vera, e poi l'esecuzione dell'istruzione `x = 0;`. Viceversa, se le variabili `a` e `b` avessero valore diverso, l'istruzione, dopo la valutazione della condizione, causerebbe l'esecuzione dell'istruzione `x = 1;`.

Osservazione 5.5

L'istruzione di selezione considerata corrisponde quindi alla seguente descrizione:

> se *il valore contenuto nella variabile* `a` *è uguale a quello contenuto nella variabile* `b` **allora** *assegna il valore* 0 *alla variabile* `x`, **altrimenti** *assegna il valore* 1 *alla variabile* `x`

Si noti che nella descrizione sono state evidenziate le parole:

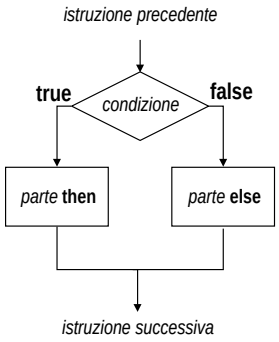
- **se** che sottolinea il fatto che si opera una scelta; per questo motivo l'istruzione di selezione viene anche chiamata istruzione **if** (che significa "se" in inglese);
- **allora** che inizia la descrizione della parte-then ("then" in inglese significa "allora");
- **altrimenti** che inizia la descrizione della parte-else ("else" in inglese significa "altrimenti").

Osservazione 5.6

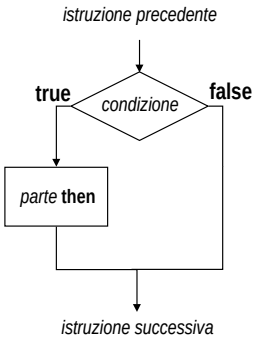
Il comportamento delle istruzioni per il controllo della sequenza dinamica può essere descritto anche graficamente attraverso un *diagramma di flusso*, che è un diagramma in cui le condizioni sono rappresentate da rombi e le istruzioni sono rappresentate da rettangoli. I rombi e i rettangoli sono poi collagati da linee orientate per indicare l'ordine seguito dall'esecutore.

Il diagramma di flusso dell'istruzione di selezione considerata è rappresentato nella figura 5.3-a.

Osservazione 5.7



(a) parte-then e parte-else non vuoti



(b) parte-else vuota

Figura 5.3: Diagramma di flusso dell'istruzione di selezione.

La parte-then o la parte-else di un'istruzione di selezione può corrispondere all'*istruzione nulla*, cioè all'istruzione che non produce alcun effetto e che si indica con il separatore `;`. In questo caso si dice che la parte-then o else è *vuota*.

Si può pertanto scrivere:

```
if ( a == b )
  ;
else
  x = 1;
```

che non produce alcun effetto se `a` e `b` hanno lo stesso valore, mentre causa l'assegnazione di 1 a `x` se `a` e `b` hanno valore diverso, oppure si può scrivere:

```
if ( a == b )
  x = 0;
else
  ;
```

che causa l'assegnazione di 0 a `x` se `a` e `b` hanno lo stesso valore, mentre non produce alcun effetto se `a` e `b` hanno valore diverso.

Se la parte-else coincide con l'istruzione nulla, la si può completamente omettere. Per esempio il secondo caso considerato potrebbe essere più semplicemente scritto come segue:

```
if ( a == b )
  x = 0;
```

Si noti però che in tutti questi casi non significa che non ci sia la parte-then o la parte-else, ma solo che la loro esecuzione non ha alcun effetto come indicato graficamente dal diagramam di flusso riportato in figura 5.3-b. L'osservazione è importante perché *nel progettare un algoritmo occorre considerare esplicitamente cosa accade quando viene eseguita una parte vuota di una istruzione di selezione*. Torneremo comunque sulla questione nel capitolo 6.

Osservazione 5.8

La parte-then e la parte-else possono essere istruzioni semplici (come nell'esempio 5.11) o composte come nel seguente esempio:

```
if ( a == b ) {
  x = 0;
  y = 1;
}
else {
  x = 1;
  y = 0;
}
```

dove sia la parte-then che la parte-else sono blocchi contenenti una sequenza di due istruzioni. Se le variabili `a` e `b` hanno lo stesso valore, l'istruzione considerata causa l'assegnazione di `0` a `x` e di `1` a `y`, se hanno valore diverso vengono assegnati i valori invertiti.

Osservazione 5.9

Si noti che se si omettono le parentesi graffe nella parte-else il significato cambia. Si consideri ad esempio, il codice:

```
if ( a == b ) {
  x = 0;
  y = 1;
}
else
  x = 1;
y = 0;
```

L'istruzione di selezione causa l'assegnazione di `0` a `x` e di `1` a `y`, se le variabili `a` e `b` hanno lo stesso valore, mentre se hanno valore diverso causa solo l'assegnazione di `1` a `x`; l'assegnazione di `0` a `y` è in sequenza all'istruzione di selezione e pertanto *viene eseguita dall'esecutore in ogni caso, indipendentemente dai valori di* `a` *e* `b`.

Si noti anche che se si omettono le parentesi graffe della parte-then si ottiene un codice scorretto senza significato.

Osservazione 5.10

Si noti l'impiego dell'indentazione in tutti gli esempi di istruzione di selezione forniti. Essa viene sempre usata per mettere in evidenza *la struttura dell'istruzione* e cioè: la riga che contiene la condizione, le istruzioni che appartengono alla parte-then, le istruzioni che appartengono alla parte-else.

È tuttavia importante notare come l'indentazione *è solo una convenzione usata dai programmatori per migliorare la leggibilità dei programmi da parte di un lettore umano*. L'esecutore non ha in alcun modo bisogno dell'indentazione perché la presenza delle parole **if** ed **else**, unitamente all'uso corretto delle parentesi tonde e graffe, chiarisce in ogni caso il ruolo di ciascuna parte dell'istruzione. Anzi, è essenziale avere ben presente che sono queste cose a determinare *il significato del codice* e non l'indentazione utilizzata.

Esempio 5.12

La parte-then e la parte-else possono a loro volta contenere istruzioni di selezione dando luogo a scelte in cascata che permettono di descrivere algoritmi che devono operare una scelta tra più di due alternative. Ad esempio, supponendo che le variabili `a`, `b` e `c` di tipo **int** siano già state definite, per assegnare alla variabile `min`, anch'essa di tipo **int**, il valore minimo contenuto nelle tre variabili si può usare il codice:

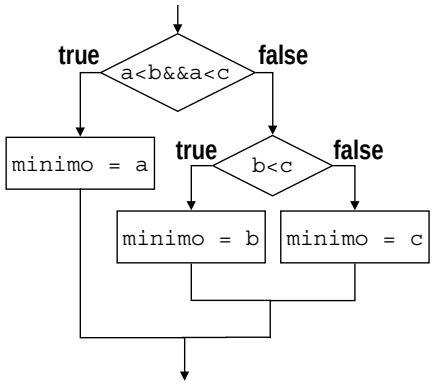


Figura 5.4: Diagramma di flusso dei istruzioni di selezione innestate.

```
if ( a<b && a<c )
  min = a;
else
  if ( b<c )
    min = b;
  else
    min = c;
```

che prevede una istruzione di selezione nella parte-else di un'altra istruzione di selezione. Per indicare questo tipo di composizione "a scatole cinesi" di istruzioni una dentro l'altra si usa il termine *annidamento* (dall'inglese *nesting*).

Applicando le regole illustrate fino ad ora è semplice comprendere il significato del codice, evidenziato anche dal diagramma di flusso della figura 5.4. Si noti che l'annidamento di due istruzioni **if** corrisponde a effettuare la scelta tra *tre* alternative. Non è difficile comprendere come usando ulteriori livelli di annidamento sia possibile operare scelte tra un qualunque numero di alternative.

Esercizi proposti

1. Scrivere un programma che legge tre numeri naturali che rappresentano i lati di un triangolo e stampa un messaggio che indica se il triangolo è equilatero, isoscele o scaleno (*suggerimento: usare uno degli algoritmi presentati nel paragrafo 2.6*).

2. Disegnare il diagramma di flusso corrispondente al seguente codice (la numerazione a sinistra non fa parte del codice):

```
1     min = a;
2     max = a;
3     if ( b < min )
4       min = b;
5     else
6       if ( b > max )
7         max = b;
8     if ( c < min )
9       min = c;
10    else
11    if ( c > max )
12        max = c;
```

3. Con riferimento al codice dell'esercizio 2, usando la numerazione delle istruzioni riportata sulla sinistra, indicare le sequenze dinamiche seguite dall'esecutore nei seguenti casi:
   - `a` vale 2, `b` vale 4, `c` vale 7;
   - `a` vale 7, `b` vale 4, `c` vale 2;
   - `a` vale 4, `b` vale 7, `c` vale 2;
4. Scrivere un programma che legge dallo standard input quattro numeri e stampa sullo standard output il valore minimo tra quelli letti.
5. Scrivere un programma che legge dallo standard input quattro numeri e stampa sullo standard output il valore minimo e il valore massimo tra quelli letti.
6. Scrivere un programma che legge dallo standard input due numeri e stampa sullo standard output i numeri letti in ordine crescente.
7. Scrivere un programma che legge dallo standard input tre numeri e stampa sullo standard output i numeri letti in ordine crescente.
8. Scrivere un programma che legge dallo standard input quattro numeri e stampa sullo standard output i numeri letti in ordine crescente.

5.9 istruzione iterativa

L'istruzione iterativa ha la seguente forma generale:

while (*condizione*)
  corpo ;

dove *condizione* è una condizione, mentre *corpo* è una istruzione qualsisasi (semplice o composta).
l'esecuzione di un'istruzione iterativa produce nell'ordine:

1. la valutazione della condizione;
2. il termine dell'istruzione se il valore logico calcolato è **false** oppure l'esecuzione del corpo e la ripetizione del punto 1 se il valore logico calcolato è **true**.

Si noti quindi che l'istruzione di selezione produce l'esecuzione del corpo *fintanto che il valore logico della condizione è* **true**. Si noti anche che se già la prima volta il valore logico della condizione è **false**, *il corpo non viene eseguito neppure una volta*. Il comportamento descritto giustifica il termine *condizione di uscita* normalmente usato per indicare la codizione dopo la parola **while** e il termine *ciclo* per indicare l'intera istruzione iterativa.

Esempio 5.13

Se le variabili `i` e `n` contengono rispettivamente i valori 0 e 4, l'istruzione:

```
while ( i < n )
  i = i + 1;
```

causa prima la valutazione della condizione `( i < n )` che risulta vera, poi l'esecuzione dell'istruzione `i = i + 1;` che trasferisce in `i` il valore 1, poi di nuovo la valutazione della condizione `( i < n )` che risulta vera, poi ancora l'esecuzione dell'istruzione `i = i + 1;` che trasferisce in `i` il valore 2, e così via per altre due volte fino a quando la quarta esecuzione dell'istruzione `i = i + 1;` trasferisce in `i` il valore 4; a questo punto viene nuovamente valutata la condizione `( i < n )` che risulta falsa, causando la terminazione dell'istruzione iterativa.

Si noti che, se inizialmente le variabili `i` e `n` contessero rispettivamente i valori 3 e 4, il corpo dell'istruzione iterativa verrebbe eseguito solo una volta. Se invece contessero rispettivamente i valori 5 e 4, il corpo dell'istruzione iterativa non verrebbe eseguito affatto.

Osservazione 5.11

L'istruzione iterativa considerata corrisponde quindi alla seguente descrizione:

> **fintanto che** *il valore contenuto nella variabile* `i` *è minore di quello contenuto nella variabile* `n`*, la variabile* `i` *viene incrementata di* 1

Si noti che nella descrizione sono state evidenziate le parole **fintanto che** che sottolineano il fatto che si esegue ripetutamente una operazione; per questo motivo l'istruzione iterativa è anche chiamata istruzione **while** (che significa appunto "fintanto che" in inglese).

Osservazione 5.12

Analogamente a quanto visto per la selezione, il diagramma di flusso dell'istruzione iterativa considerata è rappresentato nella figura 5.5.

Osservazione 5.13

Il corpo di un ciclo può essere un istruzione semplice (come nell'esempio 5.13) o composta come nel seguente esempio:

```
while ( i < n ) {
  i = i + 1;
  cout << i;
}
```

dove il corpo è un blocco contenente una sequenza di due istruzioni. Se inizialmente le variabili `i` e `n` valgono rispettivamente 0 e 4, l'istruzione iterativa considerata causa la stampa sullo standard output dei numeri: 1, 2, 3, 4.

Osservazione 5.14

Si noti che se si omettono le parentesi graffe che racchiudono il corpo il significato cambia. Si consideri ad esempio, il codice:

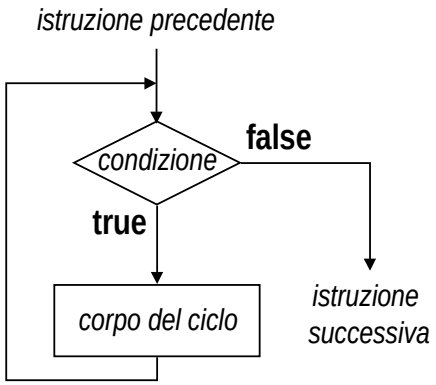


Figura 5.5: Diagramma di flusso dell'istruzione iterativa.

```
while ( i < n )
  i = i + 1;
cout << i;
```

Se inizialmente le variabili `i` e `n` valgono rispettivamente 0 e 4, l'istruzione iterativa ripete quattro volte *solo* l'incremento di `i` prima di terminare; l'istruzione di output è in sequenza all'istruzione iterativa e pertanto *viene eseguita dall'esecutore in ogni caso una sola volta*.

Osservazione 5.15

Si noti l'impiego dell'indentazione in tutti gli esempi di istruzione iterativa forniti. Essa viene sempre usata per mettere in evidenza *la struttura dell'istruzione* e cioè: la riga che contiene la condizione di uscita e le istruzioni che appartengono al corpo del ciclo.

Anche in questo caso l'indentazione *è solo una convenzione usata dai programmatori per migliorare la leggibilità dei programmi da parte di un lettore umano*. L'esecutore non ha in alcun modo bisogno dell'indentazione perché la presenza della parola **while**, unitamente all'uso corretto delle parentesi tonde e graffe, chiarisce in ogni caso il ruolo di ciascuna parte dell'istruzione.

Osservazione 5.16

Il corpo di un ciclo può contenere sia istruzioni di selezione, sia altri cicli dando luogo alla descrizone di algoritmi complessi. Anche nel caso dei cicli, per indicare questo tipo di composizione "a scatole cinesi" di istruzioni una dentro l'altra si usa il termine *annidamento*. In particolare si parla di *cicli annidati* (in inglese *nested*) per indicare la presenza di un ciclo nel corpo di un altro ciclo.

Osservazione 5.17

Un'ulteriore importante osservazione riguarda la terminazione di un'istruzione iterativa.

Poiché si assume in generale che un esecutore impiega un tempo finito per eseguire una qualunque istruzione semplice, è chiaro che l'esecutore impiega certamente un tempo finito per eseguire istruzioni composte formate solo da sequenze di assegnazioni e da istruzioni di selezione. Infatti in questo caso la lunghezza della sequenza dinamica è necessariamente finita e pertanto il tempo di esecuzione è una somma finita di termini finiti.

Questa conclusione non è però necessariamente vera allor quando l'esecutore deve eseguire un'istruzione iterativa. In questo caso la sequenza dinamica ha lunghezza finita *solo se dopo un numero finito di ripetizioni la condizione di*

uscita del ciclo diventa falsa. In caso contrario l'esecutore, costretto a rispettare il significato dell'istruzione, non può che ripetere il corpo del ciclo indefinitamente e la corrispondente sequenza dinamica ha lunghezza infinita.

È evidente che tale comportamento non è di alcuna utilità pratica e che un algoritmo che ammettesse una tale sequenza dinamica tra quelle possibili conterrebbe un errore.

Purtroppo, stabilire se un ciclo non possa dar luogo a sequenze dinamiche di lunghezza infinita non è cosa in generale agevole. Con argomenti di buon senso è tuttavia possibile stabilire tale proprietà nella maggior parte dei casi più semplici.

Esempio 5.14

Con riferimento al codice riportato nell'esempio 5.13, il ciclo non può dar luogo a sequenze dinamiche di lunghezza infinita. Infatti, poiché ad ogni ripetizione il valore di `i` viene incrementato, qualunque siano i valori iniziali di `i` e `n` dopo un numero finito di ripetizioni il valore di `i` avrà raggiunto o superato quello di `n` (eventualmente zero ripetizioni se `i` fosse maggior o uguale a `n` fin dall'inizio).

Esempio 5.15

Con riferimento al seguente codice:

```
while ( i != n ) {
  i = i + 2;
  cout << i-1;
  cout << i;
}
```

si osserva che il ciclo può dar luogo a sequenze dinamiche di lunghezza infinita. Infatti, se i valori iniziali di `i` e `n` fossero rispettivamente 0 e 3, a motivo degli incrementi ripetuti, la variabile `i` assumerebbe i valori: $0, 2, 4, \ldots$ e non diverrebbe pertanto mai uguale a `n`. Si tratta quindi di un codice errato. Nella pratica della programmazione è possibile che venga formulato per errore un ciclo che può non terminare. Si tratta di situazioni non sempre facili da rilavare nei casi concreti.

Esercizi proposti

1. Scrivere un programma che stampa sullo standard output la tabella delle moltiplicazioni tra le cifre decimali.
2. Scrivere un programma che stampa sullo standard output i primi 10 numeri naturali.
3. Scrivere un programma che legge un numero naturale N e stampa sullo standard output i primi N numeri naturali.
4. Scrivere un programma che legge un numero naturale N e stampa sullo standard output le prime N potenze di 2.
5. Scrivere un programma che legge due numeri naturali b e N e stampa sullo standard output le prime N potenze di b.

5.10 Array

In tutti gli esempi visti finora, le informazioni da elaborare, quasi sempre di natura numerica, sono state caratterizzate da valori *atomici*, non scomponibili cioè in componenti più semplici. Esistono però nella pratica anche informazioni

che per loro natura sono *strutturate*, i cui valori sono composti da valori più semplici, organizzati secondo particilari *strutture*.

Un tipico esempio di informazione strutturata è un numero complesso, formato da una coppia ordinata di numeri reali. La parte reale e la parte immaginaria sono gli elementi semplici componenti, mentre la struttura dell'informazione è quella di una *coppia ordinata*. Un altro esempio è quello di un vettore in uno spazio a n dimensioni, composto da una n-pla ordinata di valori numerici. Un terzo esempio è rappresentato da un insieme finito, composto da un numero finito di valori (non necessariamente numerici), la cui struttura questa volta non presuppone alcun ordinamento tra i componenti.

Per essere manipolate da un esecutore, le informazioni strutturate richiedono l'impiego di *variabili strutturate*, capaci cioè di contenere valori composti da elementi più semplici. Il numero di componenti e la struttura secondo cui sono organizzati i valori sono determinati dal *tipo* della variabile, che deve essere a sua volta un *tipo strutturato*.

Possono essere definiti diversi tipi strutturati, nessuno dei quali però è primitivo, cioè predefinito dal linguaggio. Il linguaggio, tuttavia, mette a disposizione dei *meccanismi di strutturazione* per definire tipi strutturati. Ciascun meccanismo determina le caratteristiche comuni ai tipi strutturati che possono essere definiti attraverso di esso. In particolare ciascun meccanismo determina se i componenti sono tutti dello stesso tipo (*componenti onogenei*) o no, e la cosiddetta *funzione di accesso*, cioè le modalità con cui si possono individuare le singole informazioni componenti a partire dall'informazione strutturata complessiva.

L'esempio di meccanismo di strutturazione di gran lunga più utilizzato nella programmazione è l'*array*, che permette di definire tipi strutturati con le seguenti caratteristiche:

- le informazioni componenti sono tutte dello stesso tipo *T*;
- il numero N delle informazioni componenti è finito e determinato all'atto della scrittura del programma;
- la funzione di accesso è basata su una corrispondenza biunivoca tra le informazioni componenti e i primi N numeri naturali, e permette di individuare un singolo valore componente dell'array attraverso un numero naturale appartenente all'intervallo $[0, N-1]$, che prende il nome di *indice*.

Queste caratteristiche sono essenzialmente quelle delle n-ple ordinate di valori omogenei che, come vedremo, trovano applicazione nelle più diverse situazioni.

Per dichiarare una variabile strutturata come array si può usare una dichiarazione del tipo:

T nome [*EXT*] ;

dove *T*, detto il *tipo base dell'array*, è un identificatore che indica il tipo dei singoli componenti, *nome* è l'identificatore associato alla variabile strutturata, ed *EXT* è l'*estensione* dell'array, cioè il numero (finito) delle componenti. La variabile strutturata, come tutte le variabili, ha un tipo associato che viene detto *array di tipo T*[2]. Si noti che, nella dichiarazione di una variabile strutturata come array, EXT deve essere un'espressione *costante* di tipo intero. Si noti anche che, per quanto detto in generale sugli array, se indichiamo con n il valore dell'espressione costante intera *EXT*, gli n componenti dell'array sono individuati dagli indici da 0 a $n-1$.

La forma usata nel linguaggio per indicare una particolare componente di una variabile strutturata come array è:

nome [*ind*] ;

dove *nome* è l'identificatore associato alla variabile, e *ind* è un'espressione *qualsiasi* di tipo intero. In particolare, *ind* può essere un'espressione non costante. La notazione appena illustrata corrisponde a indicare il componente della variabile strutturata che ha come indice il *valore* dell'espressione *ind*.

Esempio 5.16

[2] Il tipo della variabile strutturata non è genericamente *array*, ma *array di tipo T*, per sottolineare che l'array è solo un meccanismo di strutturazione che ha bisogno di un tipo base per generare un tipo strutturato.

Volendo manipolare informazioni che rappresentano punti nello spazio tridimensionali, poiché ciascun punto è individuato da tre numeri reali, si può usare una variabile di tipo array di *double* con estensione 3, dichiarata nel modo seguente:

```
double punto3D[3];
```

Secondo quanto detto in precedenza, la variabile `punto3D` è composta da tre informazioni atomiche, ciascuna di tipo **double**, associate ai tre numeri naturali 0, 1, 2. Volendo leggere dallo standard input le coordinate di un punto nello spazio per conservarle nella variabile strutturata `punto3D` si potranno usare tre istruzioni di ingresso, una per ogni componente della variabile stessa:

```
cin >> punto3D[0];
cin >> punto3D[1];
cin >> punto3D[2];
```

Alternativamente, volendo evitare di usare tre istruzioni distinte per la lettura delle tre componenti, si può usare il codice:

```
i = 0;
while ( i<3 ) {
  cin >> punto3D[i];
  i = i + 1;
}
```

Osservazione 5.18

Ciascun componente di una variabile strutturata come array può essere usato come se fosse una normale variabile del tipo base dell'array. In particolare, se il componente compare in un'espressione, il suo tipo e il suo contenuto vengono *usati* per determinare il tipo e il valore dell'espressione secondo le normali regole degli operatori coinvolti. Se invece il componente compare in una istruzione di input o a sinistra dell'operatore di assegnazione, il componente viene *definito*, viene cioè trasferito nella cella di memoria ad esso corrispondente il valore acquisito dallo standard input o calcolato sulla base dell'espressione a destra dell'assegnazione.

Osservazione 5.19

L'espressione usata per selezionare la componente di un array può essere non costante. In tal caso, le variabili presenti nell'espressione sono *usate* nell'istruzione in cui compare la componente. Si noti che questo è vero sia quando la componente è usata sia quando è definita. Ad esempio, nell'istruzione:

```
punto3D[j+1] = punto3D[i];
```

vengono usate le variabili `j`, `i`, e la componente di `punto3D` associata all'indice `i`, mentre viene definita la componente di `punto3D` associata all'indice `j+1`.

Osservazione 5.20

Al momento dell'esecuzione di una istruzione in cui compare la componente di un array, l'espressione usata per selezionare tale componente deve avere un valore che corrisponda effettivamente a una componente. In altre parole, se l'array ha n componenti, l'espressione deve avere un valore compreso tra 0 e $n-1$ inclusi. Con riferimento all'esempio riportato nell'osservazione 5.19, se `punto3D` ha tre componenti, il valore della variabile `j` deve essere compreso tra -1 e 1 inclusi, e il valore della variabile `i` deve essere compreso tra 0 e 2 compresi.

Osservazione 5.21

Il meccanismo di strutturazione come array è comune alla maggior parte dei linguaggi di programmazione. In particolare, in tutti i linguaggi che consentono la definizione di array, tali strutture sono omogenee e hanno i componenti associati a un indice. Tuttavia, da linguaggio a linguaggio possono variare aspetti secondari del meccanismo come ad esempio:

- la possibilità di usare espressioni variabili per indicare l'estensione;
- le limitazioni sulla natura degli indici;
- la possibilità di operare sull'intera variabile strutturata o su sue sottoparti diverse dai singoli componenti.

Come già visto, nel linguaggio che usiamo in questo contesto, rispetto a questi punti, gli array hanno le seguenti caratteristiche:

- l'estensione deve essere specificata mediante un'espressione costante;
- se n è l'estensione dell'array, gli indici che individuano le sue componenti sono sempre numeri naturali da 0 a $n-1$;
- è possibile dichiarare l'array coma variabile unica (sebbene strutturata), ma poi si può operare esclusivamente sulle sue componenti prese singolarmente (le elaborazioni su tutto o su parte dell'array richiedono l'uso esplicito di cicli).

5.11 Dichiarazione di costanti

Quando è richiesto di usare valori costanti all'interno di un programma (per esempio per indicare costanti universali nelle espressioni, l'estensione nelle dichiarazioni di array, ecc.) può essere conveniente usare un identificatore associato al valore costante invece di riportare esplicitamente il valore all'interno delle espressioni. Il vantaggio di questo modo di procedere è che se si vuole modificare il valore della costante (ad esempio perché si vuole aumentare il numero di cifre siginificative nel caso di costanti universali, o perché si vuole variare l'estensione di uno o più array) è sufficiente modificare solo l'istruzione che associa l'identificatore alla costante invece di modificare una per una tutte le occorrenze del valore costante presente nel programma.

L'associazione di un identificatore a un valore costante ha una forma simile a una dichiarazione di variabile, con l'aggiunta della parola riservata **const** prima del tipo e del valore costante da associare preceduto dal simbolo `=`. Si noti che non è consentito modificare una costante così definita mediante un'assegnazione successiva.

Esempio 5.17

Volendo definire una costante `PIGRECO` di tipo reale si deve usare la dichiarazione:

```
const double PIGRECO = 3.1415;
```

Ancora, volendo definire due array di interi di dimensione uno il doppio dell'altro si possono usare le dichiarazioni:

```
const int MAX_ELEMS = 100;
int a_singlo[MAX_ELEMS];
int a_doppio[2*MAX_ELEMS];
```

dove si noti che la seconda estensione è ancora un'espressione costante pur non essendo semplice.

Osservazione 5.22

L'impiego del simbolo = nella dichiarazione di una costante prende il nome di *inizializzazione* e ha proprietà diverse ripsetto all'assegnazione. Per questo motivo, la parte di dichiarazione che va dal simbolo = in poi vine detto *inizializzatore*

Si noti che gli inizializzatori possono essere usati anche nelle dichiarazioni di variabili. In tal caso la variabile viene creata già *definita* con il valore dell'inizializzatore. Ad esempio, la dichiarazione:

```
int i = 0;
```

crea una variabile associata all'identificatore `i` e al tipo **int**, già definita con il valore 0. Si noti che in questo caso, essendo `i` una variabile, il suo valore può essere successivamente modificato senza limitazioni.

Le differenze tra un inizializzatore e un'assegnazione non si possono cogliere pienamente nel contesto dei meccanismi linguistici illustrati fino a questo momento. Rinviamo pertanto la spiegazione di tali differenze.

5.12 Operatori di autoincremento e autodecremento

Le operazioni di incremento e decremento di una variabile intera sono molto frequenti nella formulazione di algoritmi. Nel linguaggio sono statoi pertanto introdotti due operatori che permettono di specificare l'incremento e il decermento di una variabile in modo sintetico.

L'*operatore di autoincremento* vine indicato con il simbolo ++ e, applicato a una variabile, la incrementa di uno. In altri termini, l'istruzione:

```
i++;
```

è *del tutto equivalente all'assegnazione*:

```
i = i + 1;
```

Analogamente, l'*operatore di autodeceremento* viene indicato con il simbolo -- e, applicato a una variabile, la decrementa di 1. Anche in questo caso c'è la piena identità con l'assegnazione equivalente.

Osservazione 5.23

Per quanto detto le istruzioni:

```
i++;
```

e:

```
i--;
```

sono equivalenti a delle assegnazioni che contengono la variabile `i` sia a destra che a sinistra dell'operatore di assegnazione. Di conseguenza tali istruzioni prima *usano* e poi *definiscono* la variabile riferita; di questo si deve tenere conto nell'applicazione della regola sull'uso delle variabili (cfr. paragrafo 4.4).

Osservazione 5.24

Gli operatori di autoincremento e di autodecremento possono essere indicati sia prima che dopo l'identificatore della variabile a cui devono essere applicati. Esiste una differenza di significato nei due casi, ma tale differenza non produce alcun effetto se l'autoincermento e l'autodecremento vengono usati esclusivamente come istruzioni isolate. Nel seguito ci limiteremo a tale uso, peraltro consigliabile in generale per motivi di leggibilità, e pertanto rimandiamo il lettore interessato ad approfondire la questione al manuale del linguaggio.

5.13 Ciclo a conteggio (for)

Nella formulazione di algoritmi è molto frequente il caso di dover far uso di un ciclo da ripetere un numero di volte noto *prima dell'inizio del ciclo*. In questi casi, si dovrebbe usare un codice del tipo:

```
i = 0;
while ( i < espressione ) {
  corpo
  i = i + 1;
}
```

dove *espressione* è un'espressione di tipo intero (anche non costante). Se le istruzioni rappresentate da *corpo* non modificano mai il valore di `i` e di *espressione*, il ciclo viene ripetuto un numero di volte esattamente pari al valore di *espressione*. Nel codice, la variabile `i` ha pertanto l'unico scopo di contare il numero di ripetizioni e viene pertanto detta *variabile di contaggio*.

Per semplificare la formulazione degli algoritmi, nel linguaggio è disponibile un'istruzione apposita per indicare un ciclo del tipo di quello appena mostrato, che prende il nome di *ciclo a conteggio* o *ciclo* **for** dal nome della parola riservata che lo caratterizza.

La forma del ciclo **for** è la seguente:

```
for ( i=0; i<espressione; i++ )
  corpo
```

e il suo significato è *esattamente lo stesso del ciclo* **while** *riportato sopra*. Si noti che le istruzioni rappresentate da *corpo* non devono modificare né il valore di `i` né quello di *espressione* [3].

5.14 Commenti

Per migliorare la leggibilità dei programmi, il linguaggio prevede la possibilità di riportare nel codice dei *commenti*. Tali commenti *non fanno parte del programma* e sono completamente ignorati dall'esecutore. Il programmatore può pertanto utilizzarli per spiegare ad eventuali lettori umani aspetti del programma *non immediatamente evidenti dal codice*.

Esistono due forme di commenti. La prima, più generale, richiede che il testo del commento sia racchiuso tra i simboli:

```
/* ... */
```

Tali commenti possono essere posti in un punto qualunque del programma e possono occupare anche più righe di testo. La seconda forma di commento inizia con il simbolo:

```
// ...
```

e può essere posto solo nella parte finale di una linea di codice, in quanto è automaticamente terminato dalla fine della linea.

[3] In realtà, poiché il ciclo **for** è solo una forma equivalente del ciclo **while**, nel linguaggio non è presente tale limitazione. Tuttavia noi la rispetteremo sempre volendo usare il ciclo **for** esclusivamente per i casi in cui il numero di ripetizione del ciclo sia noto prima dell'inizio del ciclo stesso.

5.15 Impiego di un ambiente di prgrammazione

Dopo aver introdotto i costrutti base del linguaggio ed essere dunque in grado di scrivere semplici programmi completi, è opportuno descrivere brevemente come occorre procedere per giungere all'esecuzione di un programma su un esecutore reale.

Nel paragrafo 4.1 abbiamo accennato alla possibilità che l'esecutore reale comprenda il linguaggio e lo esegua direttamente e alla possibilità alternativa che il programma debba essere tradotto prima che possa essere eseguito.

Generalmente l'esecutore descritto in queste pagine richiede l'uso di un compilatore secondo il procedimento sintetizzato dalla figura 4.1-b. Tuttavia, per semplificare l'attività di programmazione, da diversi anni si sono diffusi degli *ambienti di sviluppo integrati* che, attraverso un'unica interfaccia verso il programmatore di tipo visuale, consentono di effettuare:

- la redazione del testo del programma (*editing*);
- la traduzione del programma in linguaggio macchina e la corrispondente generazione del programma eseguibile (*compilazione*);
- l'esecuzione del programma.

L'uso degli ambienti di sviluppo semplifica molto l'uso pratico di un linguaggio per formulare algoritmi. Tuttavia la presenza della fase di traduzione non può essere mascherata al programmatore perché durante la traduzione possono essere rilevati e comunicati dal compilatore eventuali errori commessi nella scrittura del programma sorgente. Tale possibilità produce messaggi di errore non sempre facilmente comprensibili per cui le modalità di funzionamento di un compilatore meritano qualche commento ulteriore.

La compilazione viene effettuata passando attraverso quattro fasi: tre di analisi del testo del programma finalizzate alla sua comprensione, e una fase finale di traduzione vera e propria. Le tre fasi di analisi prendono il nome di *analisi lessicale*, *analisi sintattica* e *analisi semantica*, rispettivamente.

L'analisi lessicale effettua un raggruppamento dei caratteri di cui è composto il testo con lo scopo di individuare i *simboli* (detti anche *token*) del linguaggio: le parole chiave (ad esempio: **int**, **if**, **while**, ecc.), le costanti (ad esempio: `-5`, `8.09`, `'a'`, **true**, ecc.), gli identificatori (ad esempio: `X1`, `dato`, `y`, `a`, ecc.), gli operatori (ad esempio: `+`, `*`, `++`, `&&`, ecc.), i separatori (ad esempio: `;`, `[`, {, `'`, ecc.). In questa fase vengono riconosciuti ed eliminati i commenti.

L'analisi sintattica riconosce il tipo di istruzione (ad esempio: dichiarazione, assegnazione, selezione, iterazione, ecc.) e verifica che l'istruzione sia costruita correttamente, seguendo cioè le regole *sinatattiche* del linguaggio (ad esempio se tutte le parentesi aperte sono chiuse, se le istruzioni sono terminate dal punto e virgola, ecc.).

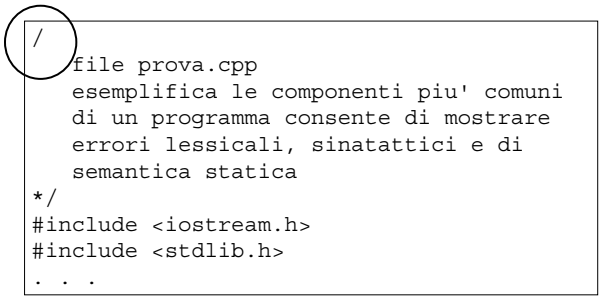
L'analisi semantica verifica se sono rispettate *alcune* regole più complesse di quelle sintattiche (ad esempio se la variaibli sono state dichiarate) e assegna ulteriori proprietà alle istruzioni (ad esempio il tipo a un'espressione).

Le prime tre fasi, possono dar luogo a errori se il programma non rispetta le regole lessicali, sinatattiche e semantiche del linguaggio. Le segalazioni non sono sempre precise e spesso sono così sintetiche da risultare oscure o generiche. È pertanto necessario fare molta pratica per acquisire la scioltezza necessaria a risolvere i casi di errori più comuni.

Se vi sono errori nelle fasi di analisi, infatti, il compilatore non effettua la quarta fase della traduzione che genera il programma eseguibile. Tuttavia, una volta che l'analisi non rileva errori, la traduzione viene sempre effettuata senza ulteriori segnalazioni di errore. Si noti però che questo non significa che il programma non contenga errori perchè la capacità del compilatore di rilevare errori è limitata. In particolare un programa lessicalmente e sintatticamente corretto e che soddisfa alcune regole semantiche basilari corrisponde sempre a un algoritmo eseguibile. Tale algoritmo tuttavia potrebbe non corrispondere affatto alle intenzioni del programmatore ed essere addirittura privo di senso!

A conclusione del capitolo riportiamo alcuni esempi di programmi che contengono errori di tipo lessicale, sintattico e semantico con le corrispondenti segnalazioni di errore generati dall'ambiente di sviluppo DevC++, vesione 4.0.

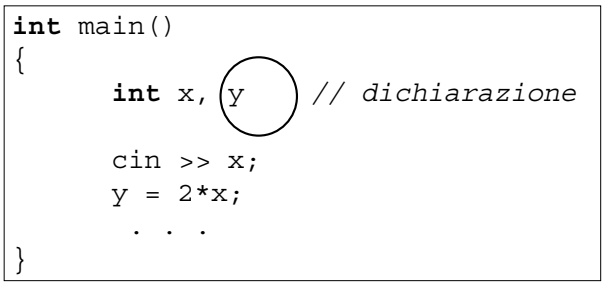
Gli esempi sono riportati nelle figure 5.6, 5.7 e 5.8. In ciascun caso è riportato il testo parziale di un programma contenente un errore evidenziato da un cerchio, e, in basso, le segnalazioni di errore riportate dall'ambiente DevC++, che si appoggia su un compilatore di pubblico dominio.



```
/
   file prova.cpp
   esemplifica le componenti piu' comuni
   di un programma consente di mostrare
   errori lessicali, sinatattici e di
   semantica statica
*/
#include <iostream.h>
#include <stdlib.h>
. . .
```

line	message
3	unterminated character constant

Figura 5.6: Esempio di errore lessicale e relativa segnalazione da parte del compilatore.

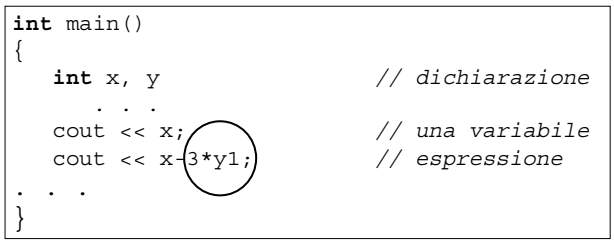


```
int main()
{
      int x, y    // dichiarazione

      cin >> x;
      y = 2*x;
        . . .
}
```

line	message
14	parse error before '> '

Figura 5.7: Esempio di errore sintattico e relativa segnalazione da parte del compilatore.



```
int main()
{
   int x, y                  // dichiarazione
      . . .
   cout << x;                // una variabile
   cout << x-3*y1;           // espressione
. . .
}
```

line	message
17	**'y1' undeclared (first use this function)**
17	**(Each undeclared identifier is reported only once**
17	**for each function it appears in.)**

Figura 5.8: Esempio di errore semantico e relativa segnalazione da parte del compilatore.

Nella figura 5.6, l'errore consiste nell'aver omesso il carattere `*` all'inizio di un commento. Il compilatore pertanto tenta di analizzare il testo del commento e segnala errore in quanto esso non contiene ovviamente codice corretto. Si noti come perlatro la segnalazione di errore non individui correttamente né la causa reale dell'errore, né la sua localizzazione (l'errore è nella linea 1 e non nella linea 3 come segnalato). Nella figura 5.7, l'errore consiste nell'aver omesso il punto e virgola dopo una dichiarazione. Si noti come anche in questo caso la segnalazione di errore non individui correttamente la causa reale dell'errore. Infine, nella figura 5.7, l'errore consiste nell'aver sbagliato il nome di una variabile (`y1` al posto di `y`). In questo caso la segnalazione di errore è abbastanza chiara.

Capitolo 6

Analisi e sintesi dei programmi (bozze, v. 2.0)

La sintesi dei programmi è un'attività di carattere progettuale che non è possibile ridurre a un'insieme di regole da applicare meccanicamente per risolvere un problema. Al contrario, si tratta di un'attività intellettualmente molto impegnativa, che richiede creatività e intuizione, oltre a una conoscenza approfondita degli strumenti tecnici specifici che l'informatica mette a disposizione. Come in tutte le attività di carattere progettuale, l'esperienza gioca un ruolo decisivo nella sintesi dei programmi, dal momento che qualunque problema non banale presenta aspetti di indeterminazione e di novità che possono essere affrontati e superati solo con l'aiuto dell'esperienza. Infine, una particolare difficoltà dovuta alla natura dei sistemi informatici sembra accompagnare l'attività di sintesi dei programmi. Tali sistemi infatti, anche nei casi più semplici, sono caratterizzati da un numero di stati interni straordinariamente elevato e dal fatto che essi sono essenzialemente instabili, nel senso che anche variazioni minime dello stato possono dar luogo un'evoluzione caotica del sistema e addirittura al suo collasso.

Per tutti questi motivi, in questa trattazione introduttiva sulla sintesi dei programmi si è scelto di dare particolare rilievo alle tecniche di *analisi dei programmi* come strumento che possa guidare il programmatore inesperto nella comprensione dei meccanismi fondamentali che presiedono alla loro sintesi. In altri termini, si è ritenuto che il modo migliore di imparare a costruire programmi sia quello di analizzare programmi già pronti, costruiti seguendo regole quanto più possibile ben definite. In questo modo è possibile ricostruire il percorso logico seguito nella sintesi degli esempi presentati e abituarsi ad un modo di ragionare che non risulta naturale a chi si avvicina per la prima volta alla progammazione intesa come attività sistematica.

Il risultato di questo modo di procedere consiste nell'identificazione di un numero limitato di "schemi" risolutivi e di regole di composizione che, con l'aiuto dell'intuizione e dell'esperienza, che non possono mai essere totalmente sostituite, consentono di derivare le soluzioni a problemi nuovi. Sebbene l'insieme di schemi e di regole di composizione discussi nel seguito sia limitato e largamente incompleto, tuttavia esso rappresenta una solida base da cui partire per estendere la propria esperienza, sia attraverso l'analisi delle soluzioni a tali problemi reperibili in letteratura, sia attraverso il tentativo disciplinato di trovare soluzione a nuovi problemi che non sembrano rientrare nei casi già sperimentati.

6.1 Esempi di programmi

Iniziamo con la presentazione di alcuni semplici programmi. In tutti i casi verrà fornita dapprima la specifica del problema che il programma risolve. Viene quindi fornito il codice corrispndente al programma che risolve il problema

posto, seguito da un breve commento che ha lo scopo di attirare l'attenzione su aspetti utili per la discussione sviluppata nel resto del capitolo.

Esempio 6.1

Specifica del problema *Acquisito dallo* `stdin` *un numero reale* x*, aggiungere allo* `stdout` *il valore* $2\,x$*.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  double x, y;
  // input dati di ingresso
  cin >> x;
  // calcolo funzione
  y = 2 * x;
  // output risultato
  cout << y;
}
```

Commento Come si può facilmente notare dai commenti che accompagnano il codice il programma consta di tre parti che indicheremo per ora informalmente con il termine *segmenti di programma*, o, più brevemente, *segmenti*. Il primo segmento ha il compito di acquisire il dato di ingresso dallo `stdin`, il secondo ha il compito di effettuare il calcolo e di salvare il risultato in una variabile definita allo scopo, il terzo ha il compito di produrre il risultato sullo `stdout`. Si noti l'uso di un'espressione per descrivere il calcolo da effettuare e dell'assegnazione per conservare il risultato del calcolo stesso.

Esempio 6.2

Specifica del problema *Acquisiti dallo* `stdin` *tre numeri reali* x*,* y *e* z*, aggiungere allo* `stdout` *il risultato dell'espressione* $x * (y + z)$*.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  double x, y, z, ris;
  // input dati di ingresso
  cin >> x;
  cin >> y;
  cin >> z;
  // calcolo funzione
  ris = x * (y + z);
  // output risultato
  cout << ris;
}
```

Commento Si possono ripetere le stesse considerazioni fatte per l'esempio precedente: i due programmi condividono una stessa struttura di base (cosa peraltro ovvia data la grande somiglianza del problema di partenza).

Esempio 6.3

Specifica del problema *Acquisito dallo* `stdin` *un numero reale* x*, aggiungere allo* `stdout` *il valore:*

$$y = \begin{cases} x+1 & \text{se } x < -1, \\ 0 & \text{se } -1 \leq x \leq 1, \\ x-1 & \text{se } 1 < x \end{cases}$$

Soluzione

```
# include <iostream.h>

int main ( ) {
  double x, y;
  // input dati di ingresso
  cin >> x;
  // calcolo funzione
  if ( x<-1 )
    y = x + 1;
  else if ( 1<x )
    y = x - 1;
  else     // -1 <= x <= 1
    y = 0;
  // output risultato
  cout << y;
}
```

Commento Anche in questo caso valgono le stesse considerazioni fatte per il primo esempio. Si noti peraltro l'uso del costrutto selettivo per il calcolo di una funzione definita per casi (e che non può quindi essere descritta da un'unica espressione). Trattandosi di tre casi sono stati usati due costrutti selettivi, uno interno all'altro (cfr. l'esempio 5.12). Si osservi inoltre che l'ordine con cui sono selezionati i casi è diverso da quello usato nella definizione della funzione. In questo modo le condizioni si semplificano, analogamente a quanto visto nell'esempio 2.2.

Esempio 6.4

Specifica del problema *Acquisita dallo* `stdin` *una lista di numeri reali preceduta dalla sua lunghezza, produrre sullo* `stdout` *la somma dei suoi valori.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  const int MAX = 100;
  int n, i;
  double lista[MAX];
  double somma;
  // input dati di ingresso
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  // calcolo somma
  somma = 0;
  for ( i=0; i<n; i++ )
    somma = somma + lista[i];
  // output risultato
  cout << somma;
}
```

Commento Prima di commentare l'algoritmo, osserviamo brevemente che per memorizzare la lista di valori è stato usato un array. Naturalmente, poiché l'array ha una lunghezza massima, il programma funzionerà correttamente solo se il valore fornito inizialmente come lunghezza della lista e memorizzato in `n` sarà minore o uguale a tale lunghezza massima (nell'esempio 100, ma il punto qui non è il valore concreto scelto, che potrebbe anche essere diverso, quanto piuttosto il fatto che il numero di caselle sia prefissato). Utilizzando quindi in modo coordinato l'array e la variabile intera `n` è possibile manipolare all'interno dell'esecutore una lista di valori di lunghezza variabile. Riprenderemo l'argomento in modo sistematico nel paragrafo 6.2.

Tornando all'analisi dell'algoritmo impiegato per risolvere il problema posto, ancora una volta esso è caratterizzato da una struttura in tre segmenti. Questa volta però si osservano sostanziali differenze nel segmento che acquisisce i dati di ingresso: non essendo costante la lunghezza della lista (e quindi il numero di dati da acquisire), viene dapprima acquisita tale lunghezza; viene poi usato un ciclo per leggere la lista nell'array. Elementi nuovi sono presenti anche nel segmento che effettua il calcolo del risultato: per calcolare la somma di tutti gli elementi della lista viene usato un ciclo a conteggio (la lunghezza della lista è infatti nota e memorizzata nella variabile `n`), combinato con l'impiego della variabile `somma` come "accumulatore" delle somme parziali ottenute sommando gli elementi della lista uno alla volta.

Esempio 6.5

Specifica del problema *Acquisita dallo* `stdin` *una lista di numeri reali preceduta dalla sua lunghezza, aggiungere allo* `stdout` *il numero dei valori nulli in essa contenuti.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  const int MAX = 100;
  int n, i;
  double lista[MAX];
  int n_zeri;
  // input dati di ingresso
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  // calcolo del numero di elementi nulli
  n_zeri = 0;
  for ( i=0; i<n; i++ )
    if ( lista[i]==0 ) n_zeri++;
  // output risultato
  cout << n_zeri;
}
```

Commento Si possono ripetere le osservazioni fatte nell'esempio precedente. Un elemento nuovo è presente nel segmento che effettua il calcolo del risultato: per calcolare il numero di elementi nulli presenti nella lista, viene impiegato una istruzione di selezione all'interno del ciclo a conteggio che scorre la lista, così da incrementare un contatore solo quando un elemento risulta uguale a zero.

Esempio 6.6

Specifica del problema *Acquisito dallo* `stdin` *un numero naturale* n*, aggiungere allo* `stdout` *la lista delle prime* n *potenze di 2.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  const int MAX = 100;
  int n, i;
  int lista[MAX];
  int p;
  // input dati di ingresso
  cin >> n;
  // calcolo delle prime n potenze di 2
  p = 1;
  for ( i=0; i<n; i++ ) {
    lista[i] = p;
    p = 2 * p;
  }
  // output risultato
  for ( i=0; i<n; i++ ) {
    cout << lista[i];
}
```

Commento Ancora una volta si possono ripetere le osservazioni fatte negli esempi precedenti per quanto riguarda la presenza di tre segmenti. Nel segmento che aggiunge la lista di potenze allo `stdout` viene usato un ciclo **for** in modo analogo a come si è fatto per l'input degli esempi precedenti. Si noti anche che nel segmento che effettua il calcolo del risultato viene usato un ciclo **for** per generare una potenza per volta, grazie alla variabile `p` usata in modo simile a come è usata `somma` nell'esempio 6.4.

Esempio 6.7

Specifica del problema *Acquisita dallo* `stdin` *una lista di numeri reali terminata dal valore* -1000*, aggiungere allo* `stdout` *la somma dei suoi valori.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  const int TAPPO = -1000;
  int n, i;
  double lista[MAX];
  double x, somma;
  // input dati di ingresso
  n = 0; cin >> x;
  while ( x != TAPPO ) {
    a[n] = x; n++;
    cin >> x;
  }
  // calcolo somma
  somma = 0;
  for ( i=0; i<n; i++ )
    somma = somma + lista[i];
  // output risultato
  cout << somma;
}
```

Commento Il programma è ancora simile a quelli mostrati nel paragrafo precedente. Tuttavia il ciclo usato per acquisire la lista (ed effettuare contemporaneamente il calcolo della somma) è un ciclo **while** con una logica diversa.

La specifica, infatti, indica che la lunghezza della lista non è fornita esplicitamente al programma prima di acquisire la lista, ma che la sua terminazione deve essere riconosciuta attraverso un'informazione particolare (il valore -1000) aggiunto in fondo alla lista stessa. Si parla in tal caso di un'informazione *tappo*, che non fa parte dei dati di ingresso, ma ne segnala solo la fine.

Il segmento che acquisisce la lista terminata da un'informazione tappo consiste nell'acquisizione "anticipata" di un valore da `stdin`, in un ciclo **while** che termina quando viene trovata l'informazione tappo, e in una nuova acquisizione anticipata al termine del corpo del ciclo, dopo aver memorizzato in un array l'elemento della lista letto. La lettura anticipata prima del ciclo è necessaria perchè non si può valutare la condizione di uscita se non si è acquisito il primo valore da `stdin`. Si noti che nel corpo del ciclo è necessario anche contare i valori letti per disporre alla fine del ciclo **while** della lunghezza della lista letta. A tal fine viene impiegata la variabile `n`, inizializzata a 0 prima di inizializzare la lettura della lista.

Esercizi proposti

1. Per ciascun esempio riportato nel precedente paragrafo, determinare 3 diversi casi di test.

2. Per ciascun esempio riportato nel precedente paragrafo, riportare la soluzione proposta nell'ambiente di sviluppo utilizzato, pervenire ad un programma eseguibile, eseguire i test elaborati svolgendo l'esercizio precedente e controllare su du essi la correttezza del programma.

6.2 Rappresentazione di liste

Prima di commentare ulteriormente gli algoritmi presentati ed introdurrne di nuovi, è opportuno discutere un argomento di natura diversa anche se strettamente collegato a quello della formulazione di algoritmi per un esecutore

automatico. L'argomento è quello della rappresentazione all'interno di un esecutore di informazioni complesse, comunemente indicato anche con il termine *strutture dati*. In effetti in quest sede ci limiteremo a discutere un singolo problema di rappresentazione e presenteremo solo due strutture dati, peraltro abbastanza simili, che lo risolvono. Non si tratta pertanto di una trattazione sistematica dell'argomento, ma solo di fornire gli strumenti necessari a comprendere a fondo la tipologia di problemi che useremo nel seguito.

Il problema che tratteremo è quello della rappresentazione di liste, dal momento che, anche ad un livello introduttivo, l'elaborazione di liste consente di presentare un'ampia varietà di soluzioni algoritmiche. In questo contesto, con il termine *lista* indichiamo un aggregato di informazioni con le seguenti caratteristiche:

- è una sequenza di valori omogenei detti *elementi* della lista;
- ha una lunghezza finita;
- nella lista possono esserci valori ripetuti;
- ciascun elemento della lista occupa una *posizione* o *posto* e i posti sono numerati a partire da 1.

Nel seguito indicheremo le liste elencandone gli elementi tra i simboli $<$ e $>$ separati da virgole. Inoltre, se l è una lista, indicheremo la lunghezza di l con la notazione $\#l$, e l'elemento di l di posto i con la notazione l_i. Ad esempio, se $l =< 0, 1, 0, 2, 0, 3 >$, l è una lista di numeri interi, $\#l = 6$, $l_1 = 0$, $l_2 = 1$, $l_3 = 0$, ecc.

Una lista si può rappresentare all'interno dell'esecutore in molti modi. Una soluzione molto semplice è quella di usare gli array. Tale soluzione ha l'inconveniente che è necessario fissare a priori *la lunghezza massima* della lista da rappresentare. Tale limitazione è tuttavia accettabile in molti casi di interesse pratico ed è ampiamente compensata dalla sua semplicità.

Si noti però che il semplice uso di un array non basta per rappresentare una lista perchè nel concetto di lista è inclusa la variabilità della lunghezza (se la lunghezza fosse costante la lista degenera in una n-pla), mentre un array ha, come abbiamo appena detto, un numero di caselle prefissato e costante. È necessario quindi affiancare all'array un modo per determinare la lunghezza della lista rappresentata, o, con altre parole, per determinare *quali caselle dell'array sono effettivamente usate per memorizzare gli elementi della lista*.

Esistono due modi principali per ottenere questo: mediante una variabile intera associata all'array oppure mediante l'uso di un'informazione tappo.

6.2.1 Rappresentazione mediante array e riempimento

Il modo più semplice di rappresentare una lista utilizzando un array consiste nel memorizzare i valori della lista a partire sempre dalla casella di indice 0 e nell'usare una variabile intera distinta, ma logicamente associata all'array, per memorizzare la lunghezza della lista. In altre parole, se indichiamo con `a` l'array e con `n` la variabile intera associata, il valore di `n` indica sempre che i valori della lista sono memorizzati nelle caselle di `a` dall'indice 0 all'indice `n`-1. Poichè il valore di `n` indica quante caselle di `a` sono effettivamente utilizzate, la variabile intera `n` viene detta *riempimento* dell'array.

Le caselle dell'array di indice maggiore di `n`-1 sono inutilizzate e vanno considerate *indefinite*. Esse sono pertanto sempre disponibili nel caso debbano essere aggiunti valori alla lista. Se il valore del riempimento è pari all'estensione dell'array, allora tutte le caselle sono definite e non possono essere aggiunti altri valori alla lista.

Si noti pertanto che:

- la lista è rappresentata dalla coppia (`a`,`n`);
- l'elemento della lista di posto i è memorizzato nell'elemento di `a` associato all'indice `i`-1;
- se MAX è l'estensione dell'array `a`, la coppia (`a`,`n`) può essere usata per rappresentare liste di lunghezza minore o uguale a MAX.

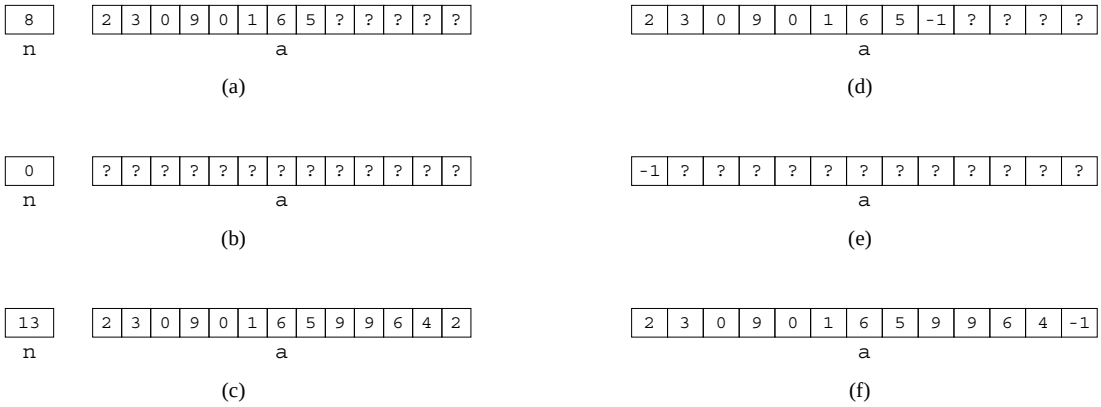


Figura 6.1: Rappresentazione di liste mediante: array e riempimento (a, b, c); array e informazione tappo (d, e, f).

Nella colonna di sinistra della figura 6.1 sono rappresentate tre differenti liste usando un riempimento `n` associato a un array `a` di estensione 13. Nella figura 6.1-(a) è rappresentata la lista $< 2, 3, 0, 9, 0, 1, 6, 5 >$ di lunghezza 8, nella figura 6.1-(b) è rappresentata la lista $<>$ di lunghezza 0 (cioè la *lista vuota*), nella figura 6.1-(c) è rappresentata la lista $< 2, 3, 0, 9, 0, 1, 6, 5, 9, 9, 6, 4, 2 >$ di lunghezza 13 (cioè della lunghezza massima consentita). Nel caso delle figure 6.1-(a) e 6.1-(b) si noti il punto interrogativo nelle caselle non occupate da elementi della lista per evidenziare che tali caselle devono considerarsi indefinite[1].

6.2.2 Rappresentazione mediante array e informazione tappo

La rappresantazione di una lista mediante array e riempimento è molto semplice, ma ha lo svantaggio di richiedere due variabili distinte. Una soluzione alternativa che consente di rappresentare la lista usando *solo l'array* consiste nel *marcare la fine della lista con un valore speciale distinguibile dai valori che possono fare parte della lista stessa*. Tale valore speciale viene indicato con il termine *informazione tappo* o, brevemente *tappo*.

Ad esempio, se vogliamo rappresentare liste di numeri naturali, possiamo usare come informazione tappo un qualsiasi numero negativo (ad esempio -1). La lista viene pertanto rappresentata memorizzando i suoi valori nell'array a partire dalla casella di indice 0, come nel caso precedente, ma aggiungendo -1 dopo l'ultimo valore della lista. La soluzione è illustrata graficamente nella parte destra della figura 6.1 dove l'array `a` di estensione 13 è usato per rappresentare tre liste differenti. Nel caso delle figure 6.1-(a) e 6.1-(b) le liste sono le stesse rappresentate nella parte sinistra della figura. Nel caso della figura 6.1-(c) la lista ha un elemento in meno perché ovviamente con il metodo dell'informazione tappo è disponibile una casella in meno per rappresentare i valori della lista.

Analogamente al caso dell'array con riempimento, se $\#l$ indica la lunghezza della lista rappresentata, gli elementi della lista sono memorizzati negli elementi dell'array `a` associati agli indici da 0 a $\#l - 1$, mentre nell'elemento di indice $\#l$ è memorizzato il valore tappo. Si noti che:

- la lista è rappresentata dal solo array `a`;
- l'elemento della lista di posto i è ancora memorizzato nell'elemento di `a` associato all'indice `i` $- 1$;
- se MAX è l'estensione dell'array `a`, possono essere rappresentate liste di lunghezza minore o uguale a $MAX - 1$.

[1] Si ricorda che una variabile indefinita non è una variabile senza un valore, ma una variabile con un valore qualsiasi e pertanto inutilizzabile in un'espressione (cfr. paragrafo 4.4).

La rappresentazione di una lista con array e informazione tappo ha il vantaggio di evitare l'impiego di due variabili distinte per rappresentare la lista che è concettualmente un un'unica informazione complessa. Vi sono però due svantaggi nell'impiego dell'informazione tappo. Il primo è che non è più disponibile in una variabile la lunghezza della lista. Nel caso serva avere tale informazione occorre scorrere l'array dall'inizio contando le caselle che precedono l'informazione tappo. Il secondo svantaggio è che, a parità di caselle nell'array, vi è una casella in meno per memorizzare gli elementi della lista (cfr. anche l'osservazione precedente sulla figura 6.1-(f)). Sebbene marginale, tale limitazione va tenuta presente nel manipolare una lista rappresentata mediante array e informazione tappo.

6.2.3 Acquisizione e rappresentazione di liste

Riconsiderando gli esempi del paragrafo 6.1, osserviamo che in tutti i casi (e cioè negli esempi 6.4, 6.5, 6.6 e 6.7) la lista è rappresentata all'interno dell'esecutore mediante il metodo dell'array più riempimento.

Tuttavia, fissando l'attenzione sul segmento di codice che effettua l'input dei dati, possiamo osservare delle differenze nella gestione della lista. Negli esempi 6.4 e 6.5, l'acquisizione della lista segue una logica analoga alla rappresentazione con array e riempimento: sullo standard input, prima della lista vera e propria è presente un numero naturale che ne indica la lunghezza. In altre parole la lunghezza della lista è nota a priori e viene fornita separatamente. Una logica dello stesso tipo è seguita anche dall'esempio 6.6, anche se in questo caso viene acquisita dall'esterno solo la lunghezza perché la lista viene generata direttamente all'interno del programma.

Una situazione diversa si riscontra invece nell'esempio 6.7 dove, in fase di acquisizione, la lunghezza della lista non viene fornita esplicitamente, ma viene calcolata dal programma contando i valori letti prima dell'informazione tappo. Questo modo di acquisire la lista è analogo al metodo di rappresentazione basato su array e informazione tappo.

L'esempio 6.7 evidenzia quindi la possibilità che l'*acquisizione* e la *rappresentazione* di una stssa lista segua logiche diverse. In particolare, la logica di acquisizione (con lunghezza esplicitamente fornita prima della lista o con informazione tappo) dipende dalle modalità con cui la lista viene fornita al programma o da esso viene generata, mentre la logica di rappresentazione è sempre una scelta del programmatore, indipendentemente dalle modalità di acquisizione. Per questo motivo, nella maggior parte degli esempi riportati in questo capitolo, il metodo di rappresentazione usato e quello basato su array e riempimento che risulta di più semplice gestione.

6.3 Schemi algoritmici

Gli esempi e le osservazioni svolti nel pragrafo 6.1 mettono in evidenza che i programmi sono costituiti da segmenti di codice e che si incontrano ripetizioni e somiglianze tra tali segmenti sia in programmi diversi, sia all'interno dello stesso programma. Tali ripetizioni e somiglianze sono molto utili nell'analisi dei programmi perché se in un nuovo programma da analizzare si individua un segmento di codice simile o addirittura identico a un segmento già incontrato, la conoscenza già acquisita può essere sfruttata per ridurre lo sforzo richiesto dalla nuova analisi.

Inoltre, l'osservazione che programmi diversi possono condividere, e di fatto condividono con frequenza, segmenti di codice simili può essere sfruttata anche nella sintesi di nuovi programmi. Supponendo infatti di disporre di un repertorio di esempi sufficientemente vasto, si potrebbe pensare di sintetizzare nuovi programmi prendendo segmenti noti e componendoli, eventualmente introducendo qualche "aggiustamento" per adattarli l'uno all'altro e al caso specifico da risolvere.

Esempio 6.8

Specifica del problema *Acquisita dallo* `stdin` *una lista di numeri interi preceduta dalla sua lunghezza, aggiungere allo* `stdout` *la lista dei numeri pari contenuti nella lista letta.*

Soluzione

```
# include <iostream.h>

int main ( ) {
  int n, i;
  int lista[MAX];
  int n_pari;
  int pari[MAX];
  // input dati di ingresso
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  // generazione della lista dei numeri pari
  n_pari = 0;
  for ( i=0; i<n; i++ )
    if ( lista[i]%2 == 0 ) {
      pari[n_pari] = lista[i];
      n_pari++;
    }
  // output risultati
  for ( i=0; i<n_pari; i++ )
    cout << pari[i];
}
```

Commento Come risulta evidente dopo un'analisi del codice, la soluzione è costituita esclusivamente da segmenti già incontrati negli esempi del paragrafo 6.1 con piccole integrazioni nella parte centrale di generazione della lista dei numeri pari.

Più specificamente, la soluzione è costituita da tre segmenti che corrispondono, come in tutti gli esempi visti, alla fase di ingresso dei dati, alla fase di calcolo e alla fase di uscita dei risultati:

- Il primo segmento è identico ai segmenti analoghi degli esempi 6.4 e 6.5, dal momento che anche in quei casi si trattava di acquisire dallo `stdin` una lista di numeri preceduta dalla sua lunghezza. Come nei casi citati la lista viene rappresentata all'interno dell'esecutore con un array e un riempimento.

- Il secondo segmento consiste in un ciclo **for** che scorre interamente la lista acquisita e che effettua un test su ciascuno dei suoi elementi. Tale segmento è molto simile al segmento centrale dell'esempio 6.5, dal momento che anche in quel caso si trattava di individuare gli elementi della lista con una particolare proprietà. La modifica apportata è dovuta alla necessità di generare nell'array `pari` la lista dei numeri selezionati, e consiste nell'usare la variabile `n_pari` come come indice e come contatore su tale array. Anche questa soluzione però era già presente negli esempi del paragrafo 6.1 e precisamente nel primo segmento dell'esempio 6.7. Se si prescinde dalle differenze dovute al fatto che i due segmenti svolgono un compito diverso (ciclo **while** invece di un ciclo **for**, presenza di istruzioni di input che qui mancano), in entrambi i casi una variabile intera viene usata come indice su un array e come contatore degli elementi dell'array effettivamente definiti (rispettivamente la variabile `n` nell'esempio 6.7 e la variabile `n_pari` nell'esempio che stiamo commentando). La cosa non è strana se si riflette che in entrambi i casi si tratta di generare una lista rappresentata mediante array e riempimento. La differenza è che in un caso i valori sono acquisiti dallo `stdin`, mentre nell'altro caso sono presi da un array già definito.

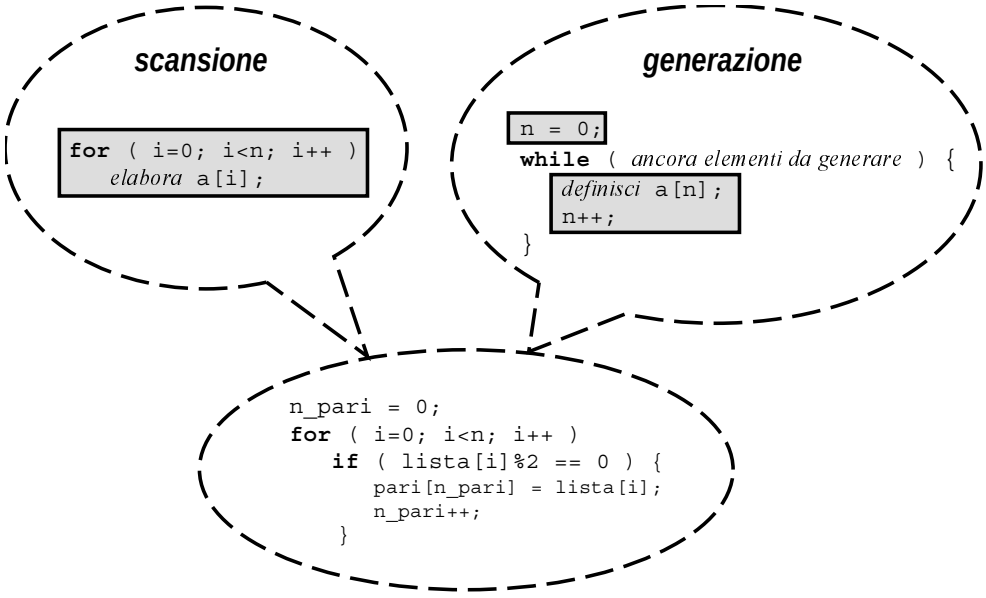


Figura 6.2: Derivazione di un segmento di codice per fusione di due schemi.

- Il terzo segmento, infine, coincide con il segmento analogo dell'esempio 6.6, dal momento che anche in quel caso si tratta di aggiungere allo `stdout` una lista, sempre rappresentata all'interno dell'esecutore da un array e un riempimento.

Sebbene l'esempio precedente mostri chiaramente come l'idea di sintetizzare nuovi programmi partendo da segmenti di codice già noti possa risultare efficace in pratica, tuttavia esso non suggerisce alcun metodo sistematico per selezionare i segmenti utili alla soluzione di un dato problema o per decidere come adattare i segmenti selezionati ai nuovi problemi.

A tal fine è necessario sviluppare ulteriormente lo studio dei segmenti di programma sotto due aspetti complementari. Da un lato è necesario definire meglio il concetto di segmento di programma, caratterizzarne le proprietà e fornire regole per modificare e comporre segmenti già noti al fine di ottenere nuovi segmenti e perfino interi programmi. Dall'altro, è opportuno mettere in evidenza a priori le principali somiglianze tra i segmenti di programma noti, in modo da sostituire un gran numero di segmenti a volte quasi identici con un numero limitato di "segmenti generalizzati", ciascuno dei quali rappresenti una classe di segmenti simili, o che condividono qualche proprietà di interesse. Questi rappresentanti di classi di segmenti, che nel seguito indicheremo con il termine *schemi algoritmici*, evidenziano gli aspetti caratteristici di ciascuna classe e sono in qualche modo già predisposti ad essere modificati. Essi sono pertanto più facilmente applicabili dei segmenti reali ad un vasto insieme di situazioni concrete differenti.

Esempio 6.9

Con riferimento all'esempio 6.8, il segmento che genera la lista dei numeri pari può essere visto come la fusione di due schemi algoritmici (descritti in dettagli nei paragrafi successivi): la *scansione* e la *generazione* di una lista rappresentata con array e riempimento.

Il procedimento di derivazione del segmento è mostrato nella figura 6.2 dove sui due schemi di partenza (nella parte alta della figura) sono evidenziati gli elementi usati per generare il segmento concreto usato nella soluzione dell'esempio 6.8.

La derivazione è evidente se si osserva che nel segmento derivato per fusione (in basso nella figura):

- l'array `lista` svolge il ruolo dell'array `a` nello schema della scansione;
- sempre nello schema della scansione l'elaborazione di `a[i]` si particolarizza nella selezione degli elementi pari dell'array `lista` tramite un'istruzione di selezione;
- la variabile `n_pari` e l'array `pari` svolgono rispettivamente il ruolo di `n` e di `a` nello schema della generazione;
- sempre nello schema della generazione la definizione di `a[i]` si particolarizza nell'assegnazione del valore pari selezionato.

Rinviando una trattazione sistematica dei segmenti di programma e delle loro proprietà al paragrafo 6.4, sviluppiamo prima l'argomento degli schemi algoritmici, introducendo un metodo per descriverli e applicando l'idea ai segmenti incontrati negli esempi fino ad ora presentati.

6.3.1 Definizione e notazione utilizzata

Comnciamo con il fornire una definizione di schema algoritmico.

Definizione 6.1 *Uno schema algoritmico è l'insieme degli elementi strutturali (che ne definiscono la "forma") e logici (che ne definiscono lo "scopo") che caratterizza una classe di algoritmi simili.*

La apparente genericità di questa definizione è motivata dalla natura estremamente eterogenea degli schemi algoritmici di interesse pratico. Si va infatti da schemi che specificano nel dettaglio quasi tutte le istruzioni da utilizzare e il modo di comporle, a schemi che specificano solo alcune variabili da usare e il ruolo che esse devono svolgere, a schemi che rappresentano un insieme limitato di algoritmi concreti, a schemi che rappresentano un numero di varianti e di algoritmi concreti praticamente illimitato.

A fronte di tanta eterogeneità, anche le modalità impiegate per la descrizione degli schemi algoritmici devono necesariamente essere molto varie. La soluzione che adotteremo nel seguito è quella di impiegare descrizioni che fanno largo uso del linguaggio naturale per mantenere la massima flessibilità. Allo stesso tempo però, per evitare il più possibile ambiguità e soprattutto per ottenere descrizioni sintetiche degli schemi, seguiremo il principio di usare gli stessi construtti del linguaggio in tutti quei casi in cui ciò è possibile senza perdere di generalità.

Esempio 6.10

Per chiarire quanto apena detto, consideriamo i seguenti segmenti di codice incontrati negli esempi precedenti:

```
// calcolo somma
somma = 0;
for ( i=0; i<n; i++ )
  somma = somma + lista[i];
```

```
// calcolo del numero di elementi nulli
n_zeri = 0;
for ( i=0; i<n; i++ )
  if ( lista[i]==0 ) n_zeri++;
```

```
// calcolo delle prime n potenze di 2
p = 1;
for ( i=0; i<n; i++ ) {
  lista[i] = p;
  p = 2 * p;
}
```

```
// output risultati
for ( i=0; i<n_pari; i++ )
  cout << pari[i];
```

```
// generazione della lista dei numeri pari
n_pari = 0;
for ( i=0; i<n; i++ )
  if ( lista[i]%2 == 0 ) {
    pari[n_pari] = lista[i];
    n_pari++;
  }
```

In tutti i casi si tratta di un segmento che *scorre* gli n elementi di una lista rappresentata con un array e un riempimento.

Tutti questi segmenti sono particolarizzazioni di uno schema algoritmico molto semplice e di uso molto comune che chiameremo *scansione semplice* o, brevemente *scansione*, e che può essere sintetizzato attraverso il seguente *pseudo-codice*:

```
for ( i=0; i<n; i++ )
  elabora a[i]
```

dove `n` rappresenta una variabile intera che contiene il numero di valori da cui è composta la lista e `a` rappresenta un array di tipo qualsiasi in cui sono memorizzati i valori a partire dalla casella di indice 0.

Il termine *pseudo-codice* indica l'uso di una combinazione arbitraria di costrutti del linguaggio di programmazione e di frasi in linguaggio naturale. I nomi di variabili usati nello pseudo-codice sono puramente indicativi e possono essere sostituiti da qualsiasi altro nome nei segmenti concreti che possono essere derivati dallo schema. Ad esempio, nel quarto segmento dell'esempio 6.10, il ruolo del riempimento `n` è svolto da `n_pari`, e il ruolo dell'array `a` è svolto da `pari`. Si noti che l'uso combinato di costrutti del linguaggio di programmazione e di linguaggio naturale hatra l'altro lo scopo di mettere in evidenza gli elementi dello schema che *devono essere presenti sempre e con lo stesso ruolo* (indicati usando i costrutti del linguaggio) da quelli che *devono essere di volta in volta adattati alle circostanze* (indicati da frasi in linguaggio naturale).

Nel seguito descriveremo alcuni schemi presenti negli esempi del paragrafo 6.1. La descrizione di ciascuno schema sarà organizzata secondo una struttura fissa e comprenderà:

1. Una descrizione a parole dello *scopo* dello schema, cioè della tipologia di compiti svolti dagli algoritmi derivabili dallo schema; ad esempio, nel caso della scansione semplice, lo scopo è, appunto *la scansione degli elementi di una lista per effettuare su ciascuno di essi una elaborazione di natura generica*.

2. Un elenco delle variabili che caratterizzano lo schema e del ruolo che esse hanno negli algoritmi derivabili dallo schema; ad esempio, nel caso della scansione semplice, *l'array* `a` *e il riempimento* `n` *rappresentano la lista e l'indice* `i` *serve a individuare il generico elemento della lista da elaborare nel corpo del ciclo*.

3. Una descrizione delle strutture di controllo (istruzioni iterative e di selezione) presenti nello pseudo-codice e del ruolo che esse hanno negli algoritmi derivabili dallo schema; ad esempio, nel caso della scansione semplice, è sempre presente un *ciclo a conteggio che effettua materialmente la scansione dell'array*.

Si noti che il punto 1 riguarda le proprietà logiche dello schema, mentre i punti 2 e 3 riguardano le sue proprietà strutturali.

Come abbiamo già osservato, le informazioni di cui ai punti 2 e 3 sono normalmente già contenute nello pseudo-codice. Pertanto si farà ricorso a commenti aggiuntivi per illustrare proprietà strutturali di uno schema solo quando lo pseudo-codice risulta insufficiente.

Infine, le descrizioni di schemi algoritmici di seguito presentate non devono essere considerate in ogni caso come qualcosa di rigido e immodificabile. Al contrario, la descrizione di uno schema algoritmico ha l'obiettivo di evidenziare *idee* di natura molto generale che caratterizzano l'organizzazione di una classe di algoritmi. Ad esempio, nel caso della scansione semplice è evidente che la scansione può anche essere effettuata partendo dal fondo e decrementando la variabile `i`, come nel seguente segmento che copia una lista invertendo l'ordine dei suoi elementi:

```
// questo segmento è una scansione di a
for ( i=n-1; i>=0; i-- )
  new_a[n-1-i] = a[i];
```

Analogamente, la scansione può essere parziale, e cioè non riguardare tutti gli elementi della lista, ma solo quelli memorizzati nelle caselle di indice da `h` a `k`, con $0 \leq$`h`$\leq$`k`$<$`n`, come nel seguente segmento di codice che stampa gl elementi della lista esclusi il primo e l'ultimo (si noti che non viene stampato nulla se la lista non ha almeno 3 elementi):

```
// questo segmento è una scansione di a
  for ( i=1; i<n-1; i++ )
    cout << a[i];
```

Dopo aver usato lo schema *scansione di una lista con riempimento* per illustrare la mozione di schema algoritmico e le modalità con cui è possibile introdurre nuovi schemi, nei prossimi paragrafi introduciamo in modo sistematico gli schemi utilizzati negli esempi fin qui presentati.

6.3.2 Generazione di una lista di lunghezza nota

Scopo Lo schema *generazione di una lista di lunghezza nota* ha lo scopo di memorizzare una nuova lista usando un array e un riempimento *nell'ipotesi di conoscere a priori la lunghezza della lista*.

Pseudo-codice

```
genera la lunghezza della lista e assegnala a n
for ( i=0; i<n; i++ )
  genera il valore di posto i + 1 e assegnalo a a[i];
```

Commenti Come già specificato nello scopo, questo schema si può applicare quando il valore da assegnare a `n` si può ottenere senza generare tutta la lista (vedi esempi successivi).

È abbastanza evidente che si tratta di uno schema derivato dalla scansione semplice e che la principale differenza è la presenza di un segmento di codice precedente al ciclo che ha lo scopo di generare la lunghezza della lista.

Esempi Dagli esempi dei pragrafi precedenti possiamo estrarre i seguenti segmenti derivati dallo schema di generazione:

1. Dagli'esempi 6.4, 6.5, 6.8:

   ```
   // input dati di ingresso
   cin >> n;
   for ( i=0; i<n; i++ )
     cin >> lista[i];
   ```

   dove la generazione dei valori viene realizzata attraverso la loro acquisizione dall'esterno tramite istruzioni di input. Si noti come lo schema presupponga che dall'esterno venga fornita la lunghezza della lista prima della lista stessa.

2. Dall'esempio 6.6:

   ```
   // input dati di ingresso
   cin >> n;
   // calcolo delle prime n potenze di 2
   p = 1 ;
   for ( i=0; i<n; i++ ) {
     lista[i] = p;
     p = 2 * p;
   }
   ```

   Si noti come, nel paragrafo 6.3.1, parte di questo segmento sia già stato considerato come derivato dalla scansione semplice. Il presentarlo ora come esempio di generazione di una lista non è una contraddizione, ma piuttosto una conferma che la generazione di una lista è un caso particolare di scansione.

6.3.3 Scansione di una lista con informazione tappo

Scopo Lo schema *scansione di una lista terminata da un'informazione tappo* ha lo scopo di scandire gli elementi di una lista rappresentata con un array e un'informazione tappo per elaborarli uno alla volta. Si noti che la lunghezza della lista non è nota a priori (cioé il suo valore non è contenuto in una variabile).

Pseudo-codice

```
i = 0;
while ( a[i] != TAPPO ) {
  elabora a[i];
  i++;
}
```

Commenti Come già specificato nello scopo, questo schema si applica quando non vi sono variabili che memorizzano la lunghezza della lista, ma tale informazione si può ottenere solo scandendo l'array fino a trovare l'informazione tappo.

Confrontando lo pseudo-codice dello schema con quello della scansione di una lista rappresentata con array e riempimento, si nota l'uso di un ciclo **while** al posto di un ciclo **for** (in questo caso il numero di ripetizioni non è noto a priori).

L'identificatore `TAPPO` che compare nello pseudo-codice rappresenta un'espressione che denota il valore dell'informazione tappo. Si noti che tale espressione è normalmente un'espressione semplice costante o variabile, a seconda

che il valore dell'informazione tappo sia fissato e noto al momento della redazione del programma o che invece possa cambiare e sia disponibile solo al momento dell'esecuzione del programma.

Esempi Non vi sono esempi di questo schema nei paragrafi precedenti. Possiamo però riprendere gli esempi di scansione di una lista rappresentata con array e riempimento usati nell'esempio 6.10 e riscriverli nel caso la lista sia rappresentata con un array e un'informazione tappo[2]. Il lettore è invitato a confrontare i segmenti di codice che nei due casi svolgono la stessa funzione e a notare analogie e differenze.

1. Calcolo della somma:

```
somma = 0;
i = 0;
while ( lista[i] != -1000 ) {
  somma = somma + lista[i];
  i++;
}
```

   dove si è ipotizzato che l'informazione tappo sia il valore costante `-1000`.

2. Calcolo del numero di elementi nulli:

```
n_zeri = 0;
i = 0;
while ( lista[i] != fine_lista ) {
  if ( lista[i]==0 ) n_zeri++;
  i++;
}
```

   dove si è ipotizzato che il valore dell'informazione tappo sia memorizzata nella variabile `fine_lista`.

3. Output risultati:

```
i = 0;
while ( pari[i] != TAPPO ) {
  cout << pari[i];
  i++;
}
```

   dove si `TAPPO` è un'espressione qualsiasi che denota il valore dell'informazione tappo.

4. Un quarto esempio è il segmento di codice che calcola la lunghezza della lista. Come infatti abbiamo chiarito in precedenza, quando la lista è rappresentata con array e informazione tappo, la sua lunghezza non è memorizzata in alcuna variabile, ma bisogna scorrere la lista contandone gli elementi fino a quando non si trova l'informazione tappo:

```
i = 0;
while ( a[i] != TAPPO ) {
  i++;
} lunghezza = i;
```

[2] Vengono presentati solo tre esempi perché gli altri due sono istanze di schemi di generazione piuttosto che semplici scansioni e saranno usati nel paragrafo successivo.

6.3.4 Generazione di una lista con lunghezza da calcolare

Scopo Lo schema *generazione di una lista di lunghezza da calcolare* ha lo scopo di memorizzare una nuova lista *nell'ipotesi che la lunghezza della lista non sia nota a priori, ma debba essere stabilita durante la generazione stessa*. Esistono due versioni dello schema a seconda che la lista generata debba essere rappresentata mediante array e riempimento o mediante array e informazione tappo.

Pseudo-codice *versione per liste rappresentate con array e riempimento*

```
n = 0;
genera un valore e assegnalo a x;
while ( x appartiene alla lista ) {
   a[n] = x;
   n++;
   genera un valore e assegnalo a x;
}
```

Commenti Come già specificato nello scopo, questo schema si applica quando occorre generare la lista per stabilire quando deve terminare la generazione. Confrontando lo pseudo-codice dello schema con quello della generazione di una lista di lunghezza nota, si nota infatti l'uso di un ciclo **while** al posto di un ciclo **for** (il numero di ripetizioni del ciclo non è noto a priori).

È facile osservare come, al termine dello schema algoritmico (cioè dopo laterminazione del ciclo **while**), la variabile `n` contiene il numero di valori generati meno uno. Tale valore corrisponde proprio alla lunghezza della lista generata in quanto l'ultimo valore ha lo scopo si segnalare che la generazione è terminata. La coppia (`a`,`n`) rappresenta quindi la lista sotto forma di array e riempimento, e dunque, se la variabile `n` viene destinata a tale scopo, lo schema fornisce direttamente una rappresentazione della lista generata mediante array e riempimento.

Normalmente la rappresentazione della lista mediante array e riempimento è pienamente soddisfacente e, come è evidente dai paragrafi precedenti, richiede algoritmi di elaborazione più semplici. Se però si vuole rappresentare la lista generata mediante array e informazione tappo si può usare la seguente versione dello schema:
versione per liste rappresentate con array e informazione tappo

```
n = 0;
genera un valore e assegnalo a x;
while ( x appartiene alla lista ) {
   a[n] = x;
   n++;
   genera un valore e assegnalo a x;
}
a[n] = x;
```

che differisce dal precedente solo per il fatto che l'ultimo valore generato, che segnala la fine della lista, viene inserito nell'array dopo l'ultimo elemento della lista (che è quello memorizzato nella casella di `a` di indice $n-1$). Si noti che, al termine dello schema algoritmico, la variabile `x` contiene il valore dell'informazione tappo e può essere usata per controllare l'uscita dal eventuali successive scansioni della lista (cfr. paragrafo 6.3.3).

Osservazione 6.1

Lo schema presentato è del tutto generale, nel senso che non fa ipotesi sul criterio usato per stabilire se l'ultimo valore gnerato appartiene o meno alla lista. Se però il valore che segnala il termine della generazione è noto a priori [3] lo schema di generazione di una lista di lunghezza da calcolare si può semplificare come segue:

[3] Si noti che la conoscenza di tale valore non implica la conoscenza del momento in cui verrà generato, e cioè la conoscenza della lunghezza della lista, e pertanto si tratta sempre di generazione di una lista con lunghezza non nota.

```
n = 0;
genera un valore e assegnalo a a[n];
while ( a[n] != TAPPO ) {
  n++;
  genera un valore e assegnalo a a[n];
}
```

dove, come al solito TAPPO rappresenta un'espressione che denota il valore che segnala il termiina della generazione. Si noti che quando è possibile usare questo schema esso fornisce sia una rappresentazione della lista con array e riempimento (la coppia (a,n)), sia una rappresentazione della lista con array e informazione tappo (la cella di a di indice n è uguale a TAPPO). In altre parole, tale schema comprende entrambe le versioni previste dallo schema.

Esempi Dagli esempi dei paragrafi precedenti possiamo estrarre i seguenti segmenti derivati dallo schema appena discusso:

1. Dall'esempio 6.7:

   ```
   // input dati di ingresso
   n = 0;
   cin >> x;
   while ( x != TAPPO ) {
     a[n] = x;
     n++;
     cin >> x;
   }
   ```

   Si noti come, nell'esempio, il segmento di codice successivo sfrutti il fatto che la generazione della lista fornisce una rappresentazione della stessa mediante array e riempimento. Si noti anche che, essendo verificata l'ipotesi dell'osservazione 6.1, il segmento potrebbe essere semplificato nel seguente:

   ```
   // input dati di ingresso
   n = 0;
   cin >> a[n];
   while ( a[n] != TAPPO ) {
     n++;
     cin >> a[n];
   }
   ```

   Come già osservato, questo segmento ha anche il vantaggio di rendere disponibile la lista in entrambe le rappresentazioni: con array e riempimento (la variabile n) e con array e informazione tappo (il valore TAPPO memorizzato in a[n]).

2. Con riferimento al calcolo delle prime n potenze di 2 riportato nell'esempio 6.10, se invece di assumere la conoscenza di n, si volesse generare la lista delle potenze minori di MAX (cioè la prima potenza maggiore o uguale a MAX segnala il termine della generazione), usando lo schema della generazione di una lista con lunghezza da calcolare si avrebbe:

```
n = 0;
p = 1;
while ( p < MAX ) {
  a[n] = p;
  n++;
  p = 2 * p;
}
```

Si noti che in questo caso, al termine del segmento, la lista viene rappresentata solo mediante array e riempimento (l'uso di un'informazione tappo in questo caso non è comunque di interesse).

3. Con riferimento alla generazione della lista dei numeri pari riportato nell'esempio 6.10, se la lista di origine è rappresentata mediante array e informazione tappo, usando lo schema della generazione di una lista con lunghezza da calcolare si avrebbe:

```
n_pari = 0;
i = 0;
while ( lista[i] != TAPPO ) {
  if ( lista[i]%2 == 0 ) {
    pari[n_pari] = lista[i];
  n_pari++;
  i++;
}
```

dove `TAPPO` è un'espressione qualsiasi che denota il valore dell'informazione tappo.

Si noti come in questo caso la generazione di nuovi valori si traduca semplicemente nell'assegnare un opportuno valore (0 la prima volta, incremento le volte successive) all'indice `i` usato per accedere alle caselle dell'array `lista`. Infatti i valori da generare sono semplicemente quelli contenuti nelle caselle di tale array.

6.3.5 Scansione con accumulatore

Scopo Lo schema *scansione con accumulatore* ha lo scopo di comporre mediante un operatore associativo i valori contenuti in una lista. Esistono due varianti dello schema a seconda che la lista generata sia rappresentata mediante array e riempimento o mediante array e informazione tappo.

Pseudo-codice *versione per liste rappresentate con array e riempimento*

inizializza `accumulatore;`
for `( i=0; i<n; i++ ) {`
  componi `a[i]` *e* `accumulatore` *con l'operatore associativo*
  e assegna il risultato a `accumulatore;`
}

versione per liste rappresentate con array e informazione tappo

```
inizializza accumulatore;
i = 0;
genera un valore e assegnalo a x;
while ( a[i] != TAPPO ) {
  componi a[i] e accumulatore con l'operatore associativo
                        e assegna il risultato a accumulatore;
  i++;
}
```

Commenti Come è immediato verificare, le due varianti sono derivate dagli schemi di scansione di una lista con riempimento e di scansione di una lista con informazione tappo. Nel seguito pertanto ci limitiamo ai commenti sugli elementi aggiuntivi che caratterizzano la scansione *con accumulatore*.

L'elemento strutturale carratterizzante lo schema è proprio la presenza di una variabile (denominata `accumulatore` nello pseudo-codice) che permette la composizione dei valori contenuti nella lista. Lo schema compone ciascun elemento individualmente con il risultato parziale di volta in volta presente nella variabile `accumulatore` riassegnando a tale variabile il risultato della composizione. L'ordine con cui gli elementi vengono composti è quello secondo il quale essi sono memorizzati nelle caselle dell'array impiegato per rappresentare la lista. Poichè l'operatore è supposto associativo, tale scelta non influisce sul risultato.

Prima della scansione la variabile `accumulatore` deve essere opportunamente inizializzata con un valore "neutro", che produce il giusto risultato quando viene composto con il primo valore della lista. La scelta di tale valore dipende dall'operatore e può comportare a volte qualche complicazione. Non è possibile fornire a questo riguardo regole del tutto generali, ma nei successivi esempi di applicazione dello schema verranno presentati e discussi i casi più comuni che sono rappresentativi della maggior parte dei casi che si possono presentare in pratica.

Al termine del ciclo presente nello schema tutti gli elementi della lista sono stati composti e la variabile `accumulatore` contiene il valore del risultato finale.

Esempi Dagli esempi dei paragrafi precedenti possiamo estrarre i seguenti segmenti derivati dallo schema appena discusso:

1. Dagli esempi 6.4 e 6.7:

   ```
   somma = 0;
   for ( i=0; i<n; i++ ) {
     somma = somma + lista[i];
   }
   ```

   In questo caso, il valore neutro è lo zero che rappresenta l'identità per l'operatore di somma. Si noti che tale scelta ha il vantaggio di generalizzare la soluzione anche al caso in cui la lista sia vuota (`n`$= 0$).

2. Dagli esempi 6.5:

   ```
   n_zeri = 0;
   for ( i=0; i<n; i++ )
     if ( lista[i]==0 ) n_zeri++;
   ```

   In questo esempio, piuttosto che comporre i valori della lista tra loro, viene incrmentato un contatore nel caso in cui il valore esaminato sia nullo (naturalmente il discorso si può generalizzare a qualunque altra proprietà). L'operazione può essere vista come la composizione mediante un operatore associativo e il segmento di codice contiene tutti gli elementi di una scansione con accumulatore. Si noti che la varabile `n_zeri` svolge la funzione di accumulatore.

3. Il segmento:

```
min = a[0];
for ( i=1; i<n; i++ )
  if ( a[i]<min ) min = a[i];
```

è un'ulteriore esempio di scansione con accumulatore.

In questo caso l'operatore con cui vengono composti i valori è quello che dà come risultato il valore minimo tra gli operandi. Si noti che la varabile `min` svolge la funzione di accumulatore. In questo caso la paculiarità sta nel fatto che il calcolo del minimo non ha senso se la lista è vuota e partanto si deve assumere che $n > 0$. Da questa osservazione discende la scelta di inzializzare l'accumulatore (in questo caso la variabile `min`) con il primo valore della lista sempre presente, e di far partire la variabile di conteggio del ciclo **for** dal valore 1.

6.3.6 Ottimizzazione per sovrapposizione di schemi

Prima di concludere questo paragrafo introduttivo sugli schemi algoritmici, presentiamo una serie di esempi riepilogativi che illustrano come sia possibile costruire nuovi programmi anche complessi partendo da schemi e segmenti noti.

Esempio 6.11

Consideriamo nuovamente l'esempio 6.4. È facile rendersi conto che per effettuare la somma degli elementi della lista è sufficiente avere a disposizione un elemento per volta e che pertanto, dopo averli sommati, gli elementi potrebbero anche essere persi. D'altra parte, essendo entrambi casi particilari di scansioni, è chiara la forte somiglianza tra il segmento che acquisisce la lista e quello che calcola la somma. Viene allora naturale pensare di "sovrapporre" i due segmenti in modo da effettuare una sola volta la scansione e poter quindi utilizzare una sola variabile temporanea per acquisire gli elementi della lista al posto di un array.

La sovrapposizione dà luogo alla seguente soluzione alternativa:

```
# include <iostream.h>

int main ( ) {
  int n, i;
  double x, somma, media;
  // input dati di ingresso e calcolo della somma
  cin >> n;
  somma = 0;
  for ( i=0; i<n; i++ ) {
    cin >> x;
    somma = somma + x;
  }
  media = somma/n;
  // output risultato
  cout << media;
}
```

dove il segmento che acquisisce ed elabora la lista può essere visto come derivato da una sorta di "sovrapposizione" dello schema di generazione con lunghezza nota e della scansione con accumulatore.

Osservazione 6.2

È importante osservare che la sovrapposizione tra input ed elaborazione, quando possibile, è molto opportuna per due motivi:

- il programma richiede un'occupazione di memoria minore perché lo spazio necessario per memorizzare l'intera lista è sostituito con lo spazio necessario per memorizzare un solo elemento;
- il programma non ha più il vincolo di dover trattare liste di lunghezza limitata: la versione dell'algoritmo di somma appena presentata funziona correttamente con liste di qualsisasi lunghezza senza bisogno di modifiche.

Si noti che entrambi i vantaggi sono rilevanti. Tuttavia vale la pena osservare che il secondo lo è particolarmente perché migliora la generalità del programma.

Osservazione 6.3

È importante tuttavia mettere in risalto che la sovrapposizione tra input ed elaborazione non è sempre possibile. Ad esempio, se si vuole acquisire una lista e ristamparla con gli elementi in posizione invertita occorre necessariamente memorizzarla interamente in memoria perché la scansione di stampa procede dall'ultimo valore acquisito al primo.

Esempio 6.12

Specifica del problema Acquisita una lista di numeri naturali terminata da un numero negativo dallo `stdin`, produrre sullo `stdout` il numero di sottosequenze di zeri consecutivi presenti nella lista. Si noti che uno zero isolato va considerato come una sottosequenza di zeri di lunghezza 1.

Casi di test Per chiarire il significato della specifica riportiamo i seguenti casi di test.

`stdin`	`7 1 0 0 7 0 1 4`
`stdout`	`2`

`stdin`	`5 0 0 0 0 0`
`stdout`	`1`

`stdin`	`0`
`stdout`	`0`

Soluzione

```
# include <iostream.h>

int main ( ) {
  const int MAX = 100;
  int n, i;
  double a[MAX];
  int n_seq;
  // input dati di ingresso e calcolo del risultato
  n = 0;
  cin >> a[0];
  while ( a[n] >= 0 ) {
    n++;
    cin >> a[n];
  } // input dati di ingresso e calcolo del risultato
  n_seq = 0;
  i = 0;
  while ( a[i] >= 0 )
    if ( a[i] == 0 ) {
      n_seq++;
      // poiché la lista è terinata da un numero negativo
      // ogni sottosequenza di zeri è terminata da un valore diverso da zero
      while ( a[i] == 0 ) i++;
    }
    else
      i++;
  // output risultato
  cout << n_seq;
}
```

Commenti Il programma è composto dai soliti tre segmenti. Il primo è la acquisizione di una lista terminata da un'informazione tappo che, come abbiamo visto è un caso particolare di generazione di lista con lunghezza da calcolare. Il terzo segmento è costituito da un unica istruzione di output.

Il secondo segmento è a a sua volta composto da una scansione della lista acquisita (terminata dall'informazione tappo)) all'interno della quale l'elaborazione è governata da una selezione che distingue due casi:

- l'elemento di indice `i` è nullo, e in tal caso si incrementa un contatore e si effettua una scansione con informazione tappo per "saltare" la sottosequenza di zeri;
- l'elemento di indice `i` non è nullo, e in tal caso si avanza senza fare nulla.

La parte centrale del programma è pertanto una scansione con accumulatore (su una lista terminata da un'informazione tappo) che contiene al suo interno una scansione semplice con informazione tappo.

L'esame ulteriore del segmento centrale mostra che la lista viene comunque scandita una sola volta e che pertanto, anche in questo caso, si può sovrapporre il segmento di acquisizione dall'esterno con quello di elaborazione in modo da evitare di memorizzare l'intera lista. Si può pertanto introdurre una variabile temporanea per l'acquisizione dei valori della lista dall'esterno, eliminando sia l'array `a` che l'indice `n` (si noti tra l'altro che non è più necessario contare gli elementi letti). Si ottiene cosi il seguente codice:

```
# include <iostream.h>

int main ( ) {
  int x, n_seq;
  // input dati di ingresso e calcolo del risultato
  n_seq = 0;
  cin >> x;
  while ( x >= 0 )
    if ( x == 0 ) {
      n_seq++;
      // poiché la lista è terinata da un numero negativo
      // ogni sottosequenza di zeri è terminata da un valore diverso da zero
      while ( x == 0 ) cin >> x;
    }
    else
      cin >> x;
  // output risultato
  cout << n_seq;
}
```

6.4 Segmenti di codice e loro proprietà

Quanto detto fino ad ora mostra da un lato che i programmi sono logicamante composti da “pezzi”, denominati fino ad ora informalmente *segmenti* (cfr. esempio 6.1), e dall’altro che, facendo uso degli schemi algoritmici, è possibile derivare, per similitudine, per fusione, o per particolarizzazione da casi generali o astratti, una grande varietà di segmenti che, se opportunamente composti, permettono di risolvere problemi nuovi o comunque diversi da quelli già noti.

Questo procedere per decomposizione da problemi complessi a problemi più semplici e, viceversa, per fusione da problemi semplici a problemi più complessi, è molto tipico dell’attività progettuale nei diversi campi dell’ingegneria ed è universalmente considerata la chiave fondamentale per affrontare la complessità dei problemi del mondo reale.

Naturalmente, per individuare i componenti di un programma, per analizzarli, per sintetizzarli e per comporli in modo da ottenere componenti nuovi e più complessi, è necessario comprendere bene la natura di questi componenti e le regole che guidano il loro uso.

Più specificamente, si in fase di analisi dei programmi, che in fase di sintesi, occorre comprendere come:

- suddividere il programma in segmenti;
- individuare le relazioni tra i segmenti;
- individuare le proprietà dei singoli segmenti.

Naturalmente, sia nell’analisi che nella sintesi di un singolo segmento, se esso non è sufficientemete semplice, si può ripetere la decomposizione dando luogo a un procedimento *a scatole cinesi* tipico dei sistemi complessi.

Anche se, come abbiamo detto, in molti casi non vi sono regole precise e complete per affrontare i vari aspetti del problema, e non è possibile in ogni caso fare a meno dell’intuizione e dell’eseperienza, pur tuttavia è possibile identificare alcune regole base che possono guidare nell’analisi e nella sintesi dei segmenti. Tali regole, anche quando non riescono a fornire un procedimento risolutivo chiaro, permettono almeno di evitare errori banali e rappresentano in ogni caso uno strumento per valutare a posteriori il programma sintetizzato. Nei prossimi paragrafi cercheremo di illustrare queste regole base e di dimostrarne l’applicazione e l’utilità attraverso alcuni ulteriori esempi.

6.4.1 Definizione

Il primo punto da chiarire è la nozione di segmento di codice introdotta informalmente nella discussione fin qui sviluppata.

Definizione 6.2 *Un segmento di codice è una qualsiasi porzione di codice che contiene esclusivamente istruzioni semplici o istruzioni composte* integre, *complete cioè di tutte le loro parti componenti.*

Ad esempio (cfr. esempio 6.12), il gruppo di istruzioni:

```
if ( x == 0 ) {
  n_seq++;
  while ( x == 0 ) cin >> x;
}
else
  cin >> x;
```

è un segmento di codice, in quanto costituito da un’unica istruzione composta (istruzione **if**), interamente contenuta nel segmento. È anche un segmento di codice il gruppo di istruzioni (contenuto nel precedente):

```
n_seq++;
while ( x == 0 ) cin >> x;
```

in quanto composto da due istruzoni, la prima semplice e la seconda composta e interamente contenuta nel segmento (ciclo **while**).

Con riferimento al primo esempio, non è invece un segmento la porzione di codice:

```
if ( x == 0 ) {
  n_seq++;
  while ( x == 0 ) cin >> x;
}
```

perchè l'istruzione **if** è solo parzialmente presente (manca la parte *else*).

6.4.2 Composizione di segmenti di codice

Nell'analisi e nella sintesi dei programmi, quando si vogliono individuare o definire i componenti del programma, vanno presi in considerazione esclusvamente i segmenti di codice. Essi infatti, grazie al fatto di comprendere istruzioni complete, sono caratterizzati da un unico *punto di ingresso* e da un unico *punto di uscita*. In altre parole, per essi risultano ben determinate le istruzioni iniziale e finale di *ogni* sequenza dinamica che può essere generata dalla loro esecuzione.

Ad esempio, nel primo esempio del paragrafo precedente, trattandosi di un segmento costituito da un'unica istruzione composta (istruzione **if**), tutte le possibili sequenze dinamiche iniziano con la valutazione dell'espressione $(\texttt{x} == 0)$ e teminano dopo l'esecuzione della parte *then* o della parte *else* (entrambe contenute nel segmento). Nella figura 6.3-(a) è riportata la rappresentazione grafica della struttura del segmento da cui appare evidente l'unicità del punto di ingresso e del punto di uscita.

È facile verificare che questa stessa proprietà è posseduta dal segmento usato nel secondo esempio del paragrafo precedente (figura 6.3-(b)), mentre nel terzo esempio il punto di uscita non è univocamente determinato (figura 6.3-(c)).

I segmenti di codice, grazie all'unicità dei punti di ingresso e di uscita, hanno la proprietà di poter essere facilmente composti attraverso i tre meccanismi di composizione fondamentale (sequenza, selezione e ciclo) senza richiedere alcuna modifica.

Si consideri ad esempio la seconda soluzione dell'esempio 6.12. Una prima divisione in componenti è riportata in figura 6.4, dove il commento iniziale ha lo scopo di descrivere sinteticamente la funzione di ciascun segmento.

Come è facile verificare si tratta in entrambi i casi di segmenti di codice (il secondo consiste di una sola istruzione semplice), composti in sequenza. Questo evidenzia la logica del programma che può essere espressa scrivendo uno dopo l'altro i due commenti:

input dati di ingresso e calcolo del risultato
output risultato

Analizzando il primo componente, possiamo individuare al suo interno i seguenti tre sottocomponenti (tutti segmenti):

```
n_seq = 0;
cin >> x;

n_seq++;
while ( x != 0 ) cin >> x;

cin >> x;
```

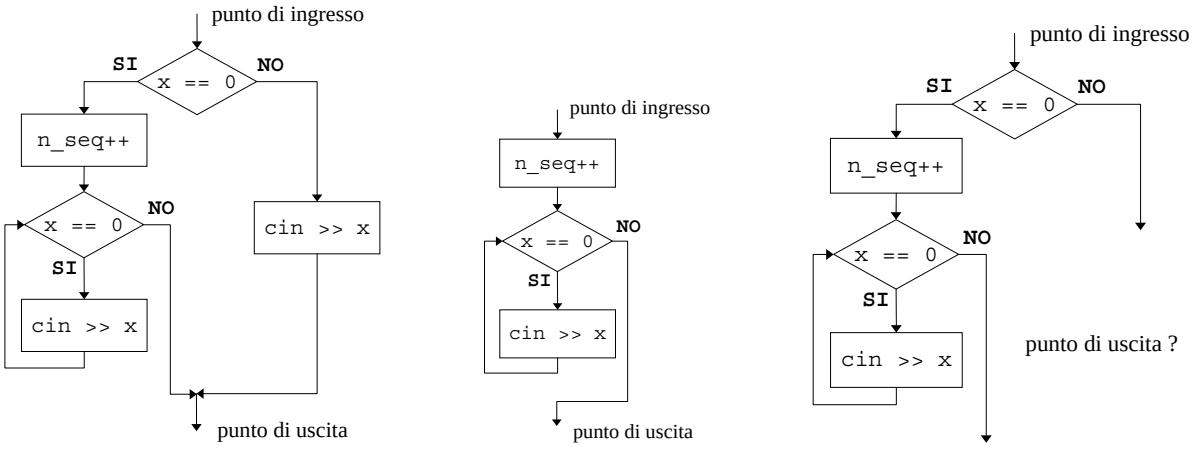


Figura 6.3: Flussi di informazioni in un segmento di codice.

Il primo segmento comprende l'inizializzazione di un contatore (`n_seq`) e l'acquisizione anticipata di un valore da `stdin`, il secondo l'incremento del contatore e l'avanzamento sullo `stdin` fino al termine di una sottosequenza di zeri consecutivi, il terzo effettua l'acquisizione di un valore da `stdin`.

I tre segmenti vengono composti secondo la logica di due schemi algoritmici (acquisizione di una lista terminata da un'informazione tappo e accumulazione del risultato nell'accumulatore `n_seq`) già evidenziati negli esempi precedenti. Tali schemi prevedono un'inizializzazione fuori ciclo (primo segmento) seguita da un ciclo **while** in cui si aggiorna l'accumulatore (prima istruzione del secondo segmento) e si avanza (ciclo nel secondo segmento o, in alternativa, terzo segmento). La presenza dell'istruzione di selezione all'interno del ciclo rappresenta la naturale generalizzazione dello schema dell'accumulatore al caso in cui l'aggiornamento debba essere fatto solo se si verifica una particolare condizione (cfr. l'esempio 2 del paragrafo 6.3.5).

Per quanto detto, l'analisi del segmento di figura 6.4-(a) porta alla seguente descrizione dove, ai tre segmenti sono stati sostituite delle frasi in linguaggio naturale che ne riassumono la funzione:

inizializzazione del contatore e lettura anticipata del primo valore dallo `stdin`
while `( x >= 0 )`
  if `( x == 0 ) {`
    incrementa il contatore e
      avanza fino al termine della sottosequenza di zeri consecutivi
  }
  else
    acquisisci un nuovo valore dallo `stdin`

Si noti che nella descrizione appenna presentata, la condizione `x >= 0` del ciclo significa *la lista non è terminata*, e la condizione `x == 0` dell'istruzione di selezione significa *siamo all'inizio di una sottosequenza di zeri consecutivi*.

Con queste osservazioni, l'algoritmo può essere descritto dal seguente pseudo-codice:

```
// input dati di ingresso e calcolo del risultato
n_seq = 0;
cin >> x;
while ( x >= 0 )
  if ( x == 0 ) {
    n_seq++;
    while ( x != 0 ) cin >> x;
  }
  else
    cin >> x;
```

(a)

```
// output risultato
cout << n_seq;
```

(b)

Figura 6.4: Componenti principali dell'algoritmo.

```
inizializzazione del contatore e lettura anticipata del primo valore dallo stdin
while ( la lista non è terminata )
  if ( siamo all'inizio di una sottosequenza di zeri consecutivi ) {
    incrementa il contatore e
        avanza fino al termine della sottosequenza di zeri consecutivi
  }
  else
    acquisisci un nuovo valore dallo stdin
```

che non contiene più alcun riferimento ai dettagli dell'algoritmo (variabili di appoggio usate, espressioni delle condizioni, ecc.).

L'ultima formulazione dell'algoritmo mette in evidenza la sua organizzazione logica e il significato che in questa organizzazione hanno le sue parti componenti. Pervenire a una formulazione dell'algoritmo con queste caratteristiche è particolarmente utile per comprendere il ragionamento fatto dal programmatore, per convincersi che non vi sono errori e per effettuare modifiche. Il procedimento seguito è basato sui seguenti passi fondamentali:

- la decomposizione in segmenti;
- l'analisi di tali segmenti attraverso l'impiego degli schemi algoritmici;
- l'identificazione del ruolo che tali segmenti svolgono nell'organizzazione complessiva dell'algoritmo ricorrendo ancora agli schemi algoritmici.

Sebbene tale procedimento non sia sempre lineare e dunque non sia riducibile a regole di automatica applicazione, l'esempio illustra il tipo di ragionamento da seguire e può essere usato come riferimento per effettuare l'analisi di altri semplici programmi (si veda a tal proposito gli esercizi proposti e i programmi mostrati nei paragrafi successivi).

6.4.3 Variabili di ingresso e di uscita

All'inizio del paragrafo 6.4 si è accennato alla necessità di individuare le relazioni tra i componenti di un programma. È infatti evidente che un programma non può ridursi ai suoi componenti presi isolatamente, ma che sono importanti anche le relazioni tra di essi, o, con altre parole, il modo con cui essi sono composti e cooperano al risultato finale.

Poiché i componenti sono segmenti di codice, occorre comprendere in che modo possono essere descritte le relazioni tra segmenti di codice. Tali relazioni sono di due tipi. Il primo tipo fa riferimento al meccanismo usato per

comporre tra loro i segmenti. In altre parole se essi sono in sequenza, in alternativa o se vengono inclusi in un ciclo. Le relazioni di questo tipo riguardano il *controllo di flusso* del segmento risultante e determinano le possibili sequenze dinamiche che esso può generare. Come abbiamo visto informalmente nei paragrafi precedenti, in molti casi il meccanismo di composizione si basa sull'impiego di uno schema algoritmico. I segmenti vengono cioè composti mediante la loro sostituzione a quelle parti dello schema che sono descritte da frasi in linguaggio naturale. Naturalmente, la sostituzione non è puramente meccanica, ma può richiedere delle modifiche ai segmenti da comporre.

Il secondo tipo di relazioni è legato alle variabili e al fatto che esse permettono il trasferimento di informazioni da un segmento all'altro. Prima di discutere in dettaglio la questione consideriamo il seguente segmento di codice:

```
// input dati di ingresso
cin >> n;
for ( i=0; i<n; i++ )
  cin >> lista[i];
// calcolo media
somma = 0;
for ( i=0; i<n; i++ )
  somma = somma + lista[i];
media = somma/n;
// output risultato
cout << media;
```

Il segmento è a sua volta composto da tre segmenti (individuati dai commenti). È abbastanza evidente che il primo segmento, prendendo i valori dall'esterno, definisce le variabili `n` e `lista` che inizialmente possono essere indefinite (non contengono cioè valori utili). Il secondo segmento utilizza i valori memorizzati dal primo segmento in `n` e `lista` per calcolare la somma e definire così la variabile `somma`. Infine il terzo segmento utilizza il valore memorizzato dal secondo segmento in `somma` per effettuare la stampa e non definisce alcuna variabile.

Considerazioni simili a quelle appena fatte possono essere ripetute su algoritmo decomponiile in più segmenti. Per chiarire meglio cosa vogliamo dire, è opportuno però prima introdurre le nozioni di *variabili di ingresso* e *variabili di uscita* di un segmento di codice.

A tal fine forniamo inizialmente una definizione semplificata che applicheremo ad alcuni esempi per chiarirne il significato e metterne in luce i limiti. Sulla base di questa discussione forniremo successivamente una definizione completa delle due nozioni.

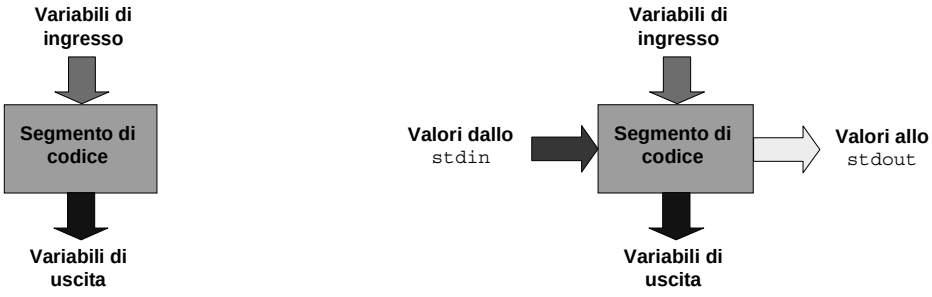
Consideriamo l'insieme $var(S)$ delle variabili il cui identificatore compare in un segmento di codice S. Le variabili appartenenti a $var(S)$ si dicono le variabili *riferite* da S. Ad esempio, nel segmento di figura 6.4-(a), l'insieme $var(S)$ comprende le variabili `n_seq`, `x`.

Ricordando che una variabile si dice *usata* quando viene usato il valore che essa contiene per il calcolo di un'espressione, e che si dice *definita* quando ad essa viene assegnato un valore, consideriamo l'insieme $in(S) \subseteq var(S)$ di variabili che vengono usate prima di essere definite indichiamo poi con il termine *variabili di ingresso* del segmento S le variabili appartenenti all'insieme $in(S)$. Ad esempio, nel segmento di figura 6.4-(a) tale insieme è vuoto perché sia `n_seq`, sia `x` vengono subito definite. Al contrario, nel segmento di figura 6.4-(b), l'unica variabile in $var(S)$, cioè `x`, è una variabile di ingresso.

Consideriamo poi l'insieme $out(S) \subseteq var(S)$ di variabili che sono definite (indipendentemente se sono anche usate) in un'istruzione del segmento e indichiamole con il termine *variabili potenzialmente*[4] *di uscita* di S. Ad esempio, nel segmento della figura 6.4-(a) le variabili potenzialmente di uscita sono, secondo questa definizione, `n_seq` e `x`, mentre nel segmento di figura 6.4-(b), non vi sono variabili di uscita.

Prima di procedere nel definire meglio sul piano formale i due concetti, conviene soffermarsi sul loro significato dal punto della logica che è alla base di un segmento di codice. Un segmento di codice descrive un algoritmo,

[4] La ragione della presenza dell'avverbio sarà chiara tra poco



(a) (b)

Figura 6.5: Flussi di informazioni in un segmento di codice.

eventualmente parziale, e pertanto descrive un procedimento che *produce un risultato a partire da certe informazioni di partenza*. Le variabili di ingresso rappresentano le informazioni di partenza dell'algoritmo descritto dal segmento e una parte di quelle di potenzialmente uscita rappresentano i risultati prodotti dall'esecuzione dell'algoritmo. In altri termini il segmento di codice, a partire dalle informazioni contenute nelle variabili di ingresso, produce dei risultati che vengono posti in un sottoinsieme delle variabili potenzialmente di uscita, che indicheremo con il termine variabili di uscita, senza più alcuna specificazione. La situazione può essere rappresentata graficamente con lo schema in figura 6.5-(a), dove il segmento è rappresentato da un rettangolo, le variabili di ingresso sono rappresentate mediante una freccia entrante nel lato superiore del rettangolo e le variabili di uscita sono rappresentate mediante una freccia uscente dal lato inferiore del rettangolo.

Osservazione 6.4

Si noti la somiglianza della rappresentazione grafica usata per i segmenti del paragrafo 6.4.2 con quella usata nella figura 6.5-(a). Nel primo caso le frecce rappresentano il *flusso di controllo*, cioè la sequenza con cui i segmenti vengono eseguiti o, in altre parole, i vincoli sulle possibili sequenze dinamiche. Nel secondo caso le frecce rappresentano il flusso dei dati, cioè le varabili che consentono un trasferimento di informazioni da un segmento di codice all'altro.

Si noti anche come, dal punto di vista grafico, quando si compongono in sequenza due segmenti di codice le variabili di uscita del primo segmento vanno a coincidere con le variabili di ingresso del secondo. Sebbene sul piano della logica dell'algoritmo complessivo non sempre si dia tale esatta coincidenza[5], pur tuttavia l'osservazione evidenzia anche graficamente il fatto che in un algoritmo ciascun segmento ha il compito di produrre le informazioni necessarie ai segmenti successivi e che tali informazioni vengono *trasferite* da un segmento all'altro attraverso le variabili definite (di uscita rispetto al segmento che precede) e usate (di ingresso rispetto al segmento che segue).

Osservazione 6.5

È importante osservare che non bisogna confondere le nozioni di *variabili di ingresso e di uscita* di un segmento di codice con i *dati di ingresso e di uscita*. Nel primo caso la nozione si applica a delle *variabili* che svolgono il ruolo di trasferire informazioni *già presenti all'interno dell'esecutore* da un segmento di codice ad un altro. Nel secondo caso la nozione si applica a *informazioni* (in concreto ai *valori* di tali informazioni) che devono essere trasferite dall'esterno dell'esecutore al suo interno (dati di ingresso) o viceversa dall'interno dell'esecutore all'esterno (dati di uscita). Nel primo caso si tratta quindi di flussi interni all'algoritmo, mentre nel secondo caso si tratta di flussi tra il mondo esterno e l'algoritmo.

[5] Non è sempre detto che *tutte* le variabili di uscita di un segmento siano di ingresso per il segmento immediatamente successivo. Una parte di esse, infatti, possono essere di ingresso per segmenti che seguono nella sequenza dinamica, ma che non sono contigui al segmento in esame.

La situazione può essere efficacemente rappresentata in modo grafico come nella figura 6.5-(b).

Oltre alle frecce che rappresentano le variabili di ingresso e di uscita che seguono una direzione verticale dall'alto verso il basso, sono riportate una freccia entrante e una uscente che seguono una direzione orizzontale da sinistra verso destra. Queste ultime frecce corrispondono, rispettivamete, a eventuali istruzioni di input e di output presenti nel segmento e rappresentano l'ingresso o l'uscita di *valori*. Naturalmente, se il segmento non contiene istruzioni di input o di output, la corrispondente freccia rappresenterebbe un insieme vuoto di valori e potrebbe essere omessa.

Vale infine la pena notare come, ai fini della comprensione della logica che governa un algoritmo, il ruolo svolto dalle variabili di ingresso e uscita è di gran lunga più importante di quello svolto dai dati di ingresso che, sotto il profilo concettuale, non comportano normalmente grandi difficoltà[6]. Per questo motivo nell'analisi di un algoritmo ci concentreremo nel seguito principalmente sulle variabili di ingresso e di uscita dei segmenti che lo compongono, piuttosto che sulle istruzioni di input e di output presenti nel segmento stesso.

Esempio 6.13

Indicando con S_1 il segmento di figura 6.4-(a), abbiamo che $in(S_1)$ è vuoto e che $out(S_1) = \{\texttt{n\_seq},\texttt{x}\}$. Questo corrisponde alla logica con cui è stato scritto il segmento che ha il compito di acquisire i dati dall'esterno (non vi sono cioè informazioni di partenza già presenti in memoria) e di produrre il numero di sottosequenze di zeri consecutivi contenute nella lista fornita sullo `stdio` nell'unica variabile di uscita `n_seq`.

Osservazione 6.6

Si noti come $out(S_1$ contenga anche la variabile `x` che non è realmente di uscita rispetto a S_1. Se infatti si esamina la logica con cui è stato scritto l'algoritmo, si può osservare che solo `n_seq` rappresenta un'informazione significativa al termine di S_1, mentre la variabile `x` ha il compito di memorizzare i valori della lista man mano che vengono acquisiti dallo standard input per consentire il conteggio delle sottosequenze di zeri consecutivi. Il suo valore finale però non ha alcun siginificato particolare. In altre parole la variabile `x` ha un ruolo all'interno di S_1, ma non ha alcun ruolo all'esterno di S_1.

Dal punto di vista della logica dell'algoritmo descritto da un segmento, questo comportamento è diverso sia da quello delle variabili di ingresso, che hanno il compito di legare un segmento con il precedente, sia da quello delle variabili di uscita come `n_seq`, che hanno il compito di legare un segmento con il successivo.

Questa osservazione porta a introdurre il termine di variabili *di algoritmo* o *ausiliarie*, termini usati per indicare le variabili solo potenzialmente di uscita, cioè quelle appartenenti ad $out(S)$ che però non sono poi effettivamente di uscita.

Purtroppo la distinzione tra variabili di algoritmo e variabili di uscita non può essere fatta impiegando esclusivamente regole meccaniche, ma richiede la conoscenza della logica con cui è organizzato l'algoritmo descritto dal segmento. In altre parole, applicando regole meccaniche è possibile individuare solo l'insieme delle variabili potenzialmente di uscita. In tale insieme, con considerazioni di altra natura, è poi possibile distinguere le variabili di algoritmo dalle variabili di uscita vere e proprie.

Esempio 6.14

Alla luce di queste osservazioni, proviamo ad applicare le definizioni date ad alcuni esempi. Indicando con Q il segmento di codice:

[6] Non a caso le nozioni di dati di ingresso e dati di uscita sono stati introdotti subito nel capitolo1, mentre le nozioni di variabili di ingresso e uscita hanno richiesto una trattazione molto più elaborata.

```
m = (p+u)/2;
if ( x < a[m] )
  u = m-1;
else if ( x > a[m] )
  p = m+1;
else
  trovato = 1;
```

abbiamo:

$$in(Q) = \{\mathtt{p}, \mathtt{u}, \mathtt{x}, \mathtt{a[m]}\},$$

$$out(Q) = \{\mathtt{m}, \mathtt{u}, \mathtt{p}, \mathtt{trovato}\}.$$

Naturalmente, non avendo informazioni sul ruolo che il segmento deve svolgere non è possibile stinguere in $out(Q)$ le variabili di algoritmo dalle variabili di uscita. Tuttavia, usando argomenti che non è il caso ora di illustrare, si può concludere che, nel caso specifico, tutte le variabili in $out(Q)$ sono di uscita.

Osservazione 6.7

Notiamo che le variabili `p` e `u` sono contemporaneamente di ingresso e di uscita. È naturale definire tali variabili di *ingresso/uscita*. Nel seguito tuttavia parleremo solo di variabili di ingresso e di uscita, distinguendo quelle che appartengono contemporaneamente ai due insiemi solo quando è necessario.

Osservazione 6.8

Notiamo anche che possono essere variabili di ingresso, di uscita o di ingresso/uscita anche gli elementi di un array. Nell'esempio è di ingresso un elemento generico dell'array, in quanto non è in generale noto il valore di `m`. In queste situazioni devono essere considerate variabili di ingresso, uscita o ingresso/uscita, rispettivamente, tutti gli elementi dell'array a cui il codice fa potenziale riferimento (nell'esempio specifico tutti gli elementi, non avendo informazioni sul valore dell'indice).

Esempio 6.15

Consideriamo un secondo esempio. Dato il segmento di codice R:

```
for ( i=0; i<n; i++ )
  if ( a[i] < min )
    min = a[i];
  else if ( a[i] > max )
    max = a[i];
```

abbiamo:

$$in(R) = \{\mathtt{n}, \mathtt{a[0]}, \mathtt{a[1]}, \ldots, \mathtt{a[n-1]}, \mathtt{min}, \mathtt{max}\},$$

$$out(R) = \{\mathtt{i}, \mathtt{min}, \mathtt{max}\}.$$

L'osservazione che `i` è la variabile di conteggio del ciclo **for** permette di concludere che `i` è una variabile di algoritmo e che pertanto le variabili di uscita sono solo `min` e `max`.

Si noti `min` e `max` sono sia di ingresso che di uscita perché entrambe sono usate nelle espressioni delle istruzioni di selezione prima di essere definite nella parte *then* delle stesse istruzioni.

Osservazione 6.9

L'esempio appena mostrato mette in luce alcune ambiguità delle definizioni date in precedenza. Infatti, se il valore di `n` fosse minore o uguale a zero, il ciclo **for** non verrebbe eseguito, e gli elementi dell'array `a` e le variabili `min` e `max` non verrebbero né usate, né definite. Sorge allora il dubbio se tali variabili debbano essere considerate di ingresso o no. Anche per quanto riguarda le variabili di potenzialmente uscita sorge un problema analogo. Le variabili `min` e `max` vengono definite solo se si verificano le seguenti condizioni: (`n`> 0) ed esistono elementi di `a` minori e maggiori, rispettivamente, dei valori iniziali di `min` e `max`), in caso contrario non vengono definite. Sorge allora il dubbio se le due variabili debbano essere ritenute di uscita o no.

Per rispondere al primo dubbio diamo una definizione più precisa di variabili di ingresso.

Definizione 6.3 *Una variabile riferita in un segmento di codice è di ingresso rispetto al segmento se essa viene usata prima di essere definita in almeno una sequenza dinamica possibile per le istruzioni del segmento.*

La novità rispetto alla nozione informale usata fino ad ora sta nelle parole *in almeno una sequenza dinamica possibile per le istruzioni del segmento*. Tale precisazione permette di scogliere l'ambiguità rilevata in precedenza e concludere che gli elementi dell'array `a` e le variabili `min` e `max` sono variabili di ingresso del segmento di codice in esame, perché esistono sequenze dinamiche possibili in cui tali variabili vengono usate prima di essere definite.

Il secondo dubbio è analogo a quello rilevato per le variabili di ingresso e si risolve precisando anche in questo caso la definizione di variabile di uscita.

Definizione 6.4 *Una variabile riferita in un segmento di codice è* potenzialmente *di uscita rispetto al segmento se si verifica una delle situazioni:*

1. *la variabile è definita in tutte le sequenze dinamiche possibili per le istruzioni del segmento;*
2. *la variabile è definita in almeno una sequenza dinamica possibile per le istruzioni del segmento, ed è anche di ingresso rispetto al segmento;*
3. *la variabile è definita in almeno una sequenza dinamica possibile per le istruzioni del segmento, non è di ingresso, ed è possibile stabilire se essa è stata effettivamente definita tramite altre variabili di ingresso o di uscita del segmento di codice.*

In quest'ultimo caso la variabile si dice condizionata *a tali variabili.*

Naturalmente, come già abbiamo osservato, tra le variabili potenzialmente di uscita così determinate andranno individuate quelle realmente di uscita con considerazioni che attengono alla logica dell'algoritmo descritto dal segmento.

Come si può notare la definizione appena data è molto più articolata di quella realtiva alle variabili di ingresso e richiede qualche spiegazione.

Per quanto riguarda le diverse condizioni che deve verificare una variabile per poter essere considerata effettivamente di uscita, ciascuna di esse è motivata dal fatto che le variabili di uscita di un segmento devono in ogni caso assumere un valore definito (prime due condizioni della definizione), o in alternativa, deve essere possibile distinguere quando esse hanno un valore definito e quando invece sono indefinite e pertanto non possono essere usate (terza condizione della definizione).

Con riferimento al segmento dell'esempio, le variabili `min` e `max` verificano la definizione anche se vengono definite solo in alcune delle sequenze dinamiche possibili, poiché sono anche di ingresso (caso 2). Esse pertanto possono essere variabli di uscita del segmento, e di fatto lo sono in quanto il loro valore al termine del segmento è evidentemente significativo (rappresentano il valore minimo e massimo contenuti nella lista rappresentata dall'array `a` e dal riempimento `n` nel caso in cui `n`> 0).

6.5 Regole per la costruzione, la composizione e la modifica di segmenti di codice

Nel paragrafo 6.4.2 abbiamo mostrato come i segmenti di codice possano essere composti mediante i meccanismi base della programmazione per ottenere segmenti più complessi. È tuttavia evidente che non basta comporre in un modo qualunque due segenti di codice, ciascuno dei quali svolge una precisa funzione, per ottenere un segmento di codice che svolga correttamente la funzione desiderata, ma che la composizione va effettuata scegliendo accuratamente i componenti, scegliendo la modalità di composizione più opportuna e aggiungendo, in qualche caso, ulteriore codice "di collegamento".

In generale, come abbiamo già precisato più volte, non è assolutamente possibile riassumere in regole precise come effettuare questa operazione che è per l'appunto uno dei compiti fondamentali nella sintesi dei programmi. È possibile però fornire alcune semplici regole che devono essere rispettate nella composizione di segmenti di codice perché il risultato della composizione possa essere potenzialmente corretto. Si tratta di condzioni *necessarie* per la correttezza, che sebbene non siano da sole sufficenti a garantirla, sono tuttavia utili per due ragioni. La prima è che esse offrono un procedimento per rilevare errori e incongruenze, sempre in agguato in ogni attività progettuale e, in particolare, nella sintesi dei programmi. La seconda ragione è che tali regole tendono a orientare l'attenzione del programmatore su aspetti essenziali, e pertanto rappresentano un solido fondamento su cui svilupare, con l'aiuto dell'intuizione e dell'esperienza, la soluzione cercata.

Le regole che vogliamo illustrare sono tutte basate sulla nozione di variabili di ingresso e variabili di uscita di un segmento di codice e derivano dalla regola fondamentale che le variabili non possono mai essere usate senza prima essere state definite. Sebbene tale regola sia stata già illustrata parlando delle espressioni e dell'istruzione di assegnazione, la ripetiamo per maggiore chiarezza dell'esposizione.

Regola 6.1 *Condizione necessaria per la correttezza di un programma è che, per qualunque sequenza dinamica possibile per le sue istruzioni, tutte le variabili usate siano state precedentemente definite.*

Applicando questa regola alla costruzione dei programmi mediante composizione e modifica di segmenti di codice, risultano come immediata conseguenza le seguenti condizioni necessarie per la correttezza della composizione.

Regola 6.2 *Se un segmento di codice S ha una variabile di ingresso x, esso deve essere composto con altri segmenti di codice in modo tale che, per tutte le sequenze dinamiche possibili, la sua esecuzione sia preceduta dall'esecuzione di un altro segmento Q che ha x come variabile di uscita non condizionata.*

La regola precedente può essere estesa al caso in cui x è di uscita ripetto a Q, ma condizionata a un'altra variabile y.

Regola 6.3 *Nel caso la variabile x di uscita rispetto al segmento Q che precede S, sia condizionata a una o pi ù variabili, allora la composizione è corretta solo se l'uso di x in S è condizionato al valore di tali variabili.*

Come corollario alla regola 6.1 vale la seguente.

Regola 6.4 *Il segmento di codice con cui inizia un programma non deve avere variabili di ingresso.*

Si noti infatti che, in caso contrario, esisterebbero sequenza dinamiche possibili nelle quali tali variabili di ingresso verrebero usate prima di essere state definite.

Un altra regola di costruzione dei programmi è la seguente.

Regola 6.5 *Un segmento di codice che non sia quello con cui termina il programma deve avere variabili di uscita, a meno che non contenga istruzioni di output.*

Il motivo di tale regola è che, in caso contrario (se cioè il segmento non ha né variabili di uscita, né contiene istruzioni di output), il segmento è inutile in quanto potrebbe essere tranquillamente tolto dal programma senza produrre alcun effetto. Sebbene una tale situazione non sia in sé necessariamente un errore, è quantomeno un indice che il programma contiene istruzioni non logicamente collegate alle altre.

Terminiamo con un esempio di applicazione delle nozioni introdotte nei paragrafi precedenti. Per snellire la discussione, nel seguito numereremo le linee di codice e useremo la convenzione di usare il simbolo S_{x-y} per indicare il segmento di codice che inizia alla linea x e termina alla linea y incluse. Nel caso un segmento sia costituito da una sola riga x useremo per indicarlo il simbolo S_x.

Esempio 6.16

Consideriamo la specifica:

> *Scrivere un programma che riceve in ingresso un numero naturale* N *seguito da una lista di* N *numeri e un ulteriore numero* M*, e che stampa in uscita la lista letta concatenata a se stessa M volte*

Il problema viene risolto dal seguente programma:

```
 1.  #include <iostream.h>
 2.  #include <stdlib.h>
 3.
 4.  int main()
 5.  {
 6.    int N, M;
 7.    const int MAX = 100;
 8.    int a[MAX];
 9.    int i, j;
10.
11.    cin >> N;
12.    for ( i=0; i<N; i++ )
13.      cin >> a[i];
14.
15.    cin >> M;
16.
17.    for ( j=0; j<M; j++ )
18.      for ( i=0; i<N; i++ ) {
19.        cout << a[i];
20.        cout << ' ';
21.      }
22.    cout << endl;
23.
24.    return 0;
25.  }
```

dove la numerazione delle linee di codice è stata riportata per semplificare la successiva discussione.

Cerchiamo ora di analizzare la soluzione proposta facndo uso delle nozioni di schema algoritmico e di segmento di codice e delle regole per la loro composizione.

A parte le dichiarazioni delle variabili, che possono essere trascurate perché non influenzano la logica dell'algoritmo, con riferimento agli schemi noti si può notare che il segmento:

```
11.    cin >> N;
12.    for ( i=0; i<N; i++ )
13.      cin >> a[i];
```

è un'acquisizione di lista di lunghezza nota.

Tale segmento non ha variabili di ingresso e ha `N`, `i`, `a[0]`, `a[1]`, ..., `a[N-1]` come variabili potenzialmente di uscita. Di queste, `i` è di algoritmo e le altre sono realmente di uscita. Il segmento è il primo del programma e verifica pertanto la regola 6.4.

Con riferimento alla logica con cui è organizzato il programma, il segmento appena considerato *acquisisce dallo standard input una lista preceduta dalla sua lunghezza*. Questo compito è coerente con quanto indicato nella traccia e con le variabili di uscita che coincidono con la rappresentazione della lista letta mediante array e riempimento.

Sempre con riferimento agli schemi introdotti, il segmento:

```
18.      for ( i=0; i<N; i++ ) {
19.        cout << a[i];
20.        cout << ' ';
21.      }
```

è una scansione semplice della lista (`a`,`N`) in cui l'elaborazione del generico elemento della lista coincide con un'istruzione di output che aggiunge il valore dell'elemento allo standard output, seguita da un'istruzione di output che aggiunge allo standard output un carattere bianco.

Le variabili di ingresso del segmento sono: `N`, `a[0]`, `a[1]`, ..., `a[N-1]`; c'è la variabile di algoritmo `i`; non vi sono variabili di uscita (ma la regola 6.5 è soddisfatta per la presenza di istruzioni di uscita).

Dal punto di vista logico il segmento *stampa la lista* (`a`,`N`), cioè aggiunge allo standard output la lista rappresentata dall'array `a` e dal riempimento `N`. Tale compito è coerente con le variabili di ingresso individuate e con il fatto che non vi sono variabili di uscita. In altre parole il "prodotto" dell'esecuzione del segmento è esclusivamente esterno all'esecutore.

Il segmento appena esaminato è il corpo di un ciclo **for**, che è a sua volta un'istanza dello schema della scansione. Tenendo presente le considerazioni appena fatte su S_{18-21}, il segmento S_{17-21} può essere descritto in termini astratti come segue:

for `( j=0; j<M; j++ )`
  stampa la lista (`a`,`N`)`;`

Che mette in evidenza come la stampa ripetuta della lista (`a`,`N`) viene ottenuta annidando la scansione che effattua una stampa della lista all'interno di una scansione delle `M` stampe. Il segmento S_{17-21}, oltre alle stesse variabili di ingresso del segmento S_{18-21}, ha `M` come ulteriore variabile di ingresso.

Vi sono poi due segmenti di codice non ancora considerati:

```
15.    cin >> M;
```

e:

```
22.    cout << endl;
```

Si tratta di segmenti costituiti da una sola istruzione per i quali l'analisi risulta banale.

Vale invece la pena osservare come i segmenti individuati siano composti rispettando la regola 6.1 e, più specificamente le sue conseguenze rappresentate dalle regole 6.2 e 6.3. Infatti, il segmento S_{17-21} (l'unico che ha variabili di ingresso) è correttamente preceduto da un segmento che ha come variabili di uscita le variabili: `N`, `a[0]`, `a[1]`, ..., `a[N-1]` (il segmento dalla S_{11-13}) e da un segmento che ha come variabili di uscita la variabile `M` (il segmento S_{15}). Tale ordine corrisponde infatti alla logica della soluzione che acquisisce dapprima la lista, poi il valore M e infine aggiunge allo standard output M copie concatenate della lista letta.

La presenza dell'ultimo segmento S_{22} ha il solo scopo di terminare con un'andata a capo la sequenza di valori aggiunta allo standard output.

6.6 Ricerca di una proprietà in una lista

Esaminiamo ora un ulteriore schema algoritmico di grandissima importanza per la possiilità di impiegarlo, con le opportune generalizzazioni, in un grandissimo numero di situazioni a prima vista completamente diverse.

Scopo Lo schema di *ricerca di una proprietà in una lista* ha lo scopo di scandire una lista alla ricerca di un elemento che verifica una determinata proprietà. Anche per questo schema esistono due varianti a seconda che la lista generata sia rappresentata mediante array e riempimento o mediante array e informazione tappo.

Pseudo-codice *versione per liste rappresentate con array e riempimento*

```
trovato = false;
i = 0;
while ( (i<n) && !trovato )
  if ( a[i] verifica la proprietà cercata )
    trovato = true;
  else
    i++;
```

versione per liste rappresentate con array e informazione tappo

```
trovato = false;
i = 0;
while ( (a[i]!=TAPPO) && !trovato )
  if ( a[i] verifica la proprietà cercata )
    trovato = true;
  else
    i++;
```

Commenti Nello schema oltre alle variabili che rappresentano la lista, sono impiegate altre due variabili essenziali: l'indice `i`, usato per scandire le celle dell'array, e una variabile `trovato` di tipo **bool**, che indica se la proprietà è stata trovata o no.

La prima osservazione da fare sullo schema riguarda la natura della condizione di uscita del ciclo **while**. Si tratta di una condizione composta, costituita da due sottocondizioni. La prima sottocondizione verifica che l'elemento corrente dell'array faccia ancora parte della lista (cioè: `(i<n)` nella prima variante e `(a[i]!=TAPPO)` nella seconda variante). La seconda sottocondizione verifica che la proprietà cercata non sia stata ancora trovata (cioè il valore della variabile `trovato` è **false**). Le due sottocondizioni sono composte mediante l'operatore AND e dunque il corpo del ciclo viene eseguito fin tanto che *l'elemento corrente dell'array fa parte della lista* e *la proprietà cercata non è stata trovata*.

Quando il ciclo termina si possono dare due casi:

1. `i`$\geq$`n` e `trovato` uguale a **false**: in questo caso sono state esaminate una alla volta tutte le caselle dell'array che contengono elementi della lista senza che sia stata trovata la proprietà cercata.

2. `i<n` e `trovato` uguale a **true**: la proprietà è stata trovata e il valore di `i` indica la casella dell'array che ha verificato la proprietà cercata; si noti che possono esserci altre caselle successive che verificano la proprietà, ma il ciclo si ferma alla prima; ulteriori considerazioni in proposito verranno svolte negli esempi riportati di seguito.

Si noti che il ciclo è scritto in modo tale che non può accadere che le due sottocondizioni della condizione di uscita del ciclo siano false entrambe contemporaneamente. Questa osservazione permette di concludere che, al termine dell'esecuzione dello schema, il valore della variabile `trovato` permette di sapere quale dei due casi si è verificato.

Un'altra osservazione importante riguarda il valore dell'indice `i` al termine dell'esecuzione dello schema. Nel caso 1 il valore assunto da `i` non ha alcun significato particolare e tale variabile è servita solo per scandire l'intera lista senza che sia stata trovata la proprietà cercata. Al contrario, nel caso 2 il valore assunto da `i` corrisponde all'indice del primo elemento della lista che verifica la proprietà cercata. In questo caso pertanto `i` contiene un'informazione potenzialmente significativa.

Questa considerazione ci permette di valutare meglio il ruolo delle variabili impiegate nello schema. Le variabili di ingresso sono quelle usate per rappresentare la lista (array e riempimento nella prima variante, solo l'array nella seconda variante). Le variabili potenzialmente di uscita sono `trovato` e `i`. La prima è di uscita e contiene il principale risultato dell'algoritmo (se la proprietà è stata trovata o no). La seconda è pure di uscita, ma condizionata al valore di `trovato`, nel senso che il suo valore è significativo solo nel caso 2[7].

Vale la pena osservare come la classificazione in variabili di ingresso e variabili di uscita fatta sulla base delle proprietà strutturali del codice corrisponda ancora una volta al loro ruolo dal punto di vista concettuale. Poiché lo schema ha lo scopo di cercare se esiste un elemento della lista che verifica una una priprietà data, è ovvio che la lista rappresenti l'informazione di partenza (variabili di ingresso) e che l'informazione se la proprietà è stata trovata rappresenti il risultato (variabile di uscita). Inoltre, nel caso la proprietà sia stata verificata da un elmento della lista, anche l'informazione su quale elemento ha verificato la proprietà è un risultato dell'algoritmo.

Un ultimo commento riguarda la variante dello schema per liste rappresentate con array e informazione tappo. Se si confronta tale variante con lo schema di scansione di una lista con informazione tappo, si può osservare facilmente che la seconda variante dello schema di ricerca è una *fusione* della prima variante dello schema di ricerca con lo schema di scansione di una lista con informazione tappo. In altre parole, la seconda variante dello schema di ricerca non è propriamente uno schema base, ma un derivato da schemi già noti. L'osservazione mette in luce ancora una volta la capacità degli schemi algoritmici di combinarsi e generare un gran numero di varianti in grado di trattare le situazioni più diverse.

Esempi Forniamo di seguito alcuni esempi di applicazione dello schema alla ricerca di proprietà specifiche. In tutti i casi forniamo il segmento corrispondente alla prima variante dello schema, lasciando al lettore il compito di derivare l'equivalente segmento nel caso la lista sia rappresentata con array e informazione tappo.

1. Il seguente segmento verifica se la lista contiene un elemento uguale a un valore x dato.

```
trovato = false;
i = 0;
while ( (i<n) && !trovato )
  if ( a[i] == x )
    trovato = true;
  else
    i++;
```

   Come è ovvio sulla base di quanto osservato in precedenza, sono variabili di ingresso del segmento: `n`, le caselle di `a` dall'indice 0 all'indice `n`-1 inclusi, e `x`. In altre parole la lista e il valore da cercare. Le variabili `trovato` e `i` sono di uscita, la seconda condizionata alla prima.

   È interessante vedere anche come un algoritmo di ricerca possa essere seguito da un segmento che usi i risultati della ricerca stessa. Volendo, ad esempio, semplicemente stampare un messaggio che informi sull'esito della ricerca, il codice riportato sopra potrebbe essere immediatamente seguito dal seguente segmento:

[7] Si noti che al termine dello schema, da un punto di vista concettuale, nel caso 1 il valore di `i` va considerato *indefinito* e pertanto tale variabile non può essere usata dai segmenti successivi.

```
if ( trovato ) {
  cout << " Il valore " << x;
  cout << " e' presente nella posizione ";
  cout << i+1 << endl;
}
else {
  cout << " Il valore " << x;
  cout << " non e' presente" << endl;
}
```

Si noti come, rispetto a questo segmento, le variabili `trovato` e `i` siano di ingresso, ma l'uso di quest'ultima sia condizionato al valore della prima (nella parte *else* la variabile `i` non compare). Questo è coerente con il ruolo che tali variabili hanno rispetto al segmento che effettua le ricerca (cfr. regola 6.3). Considerazioni analoghe possono ripetersi nel caso dei segmenti presentati negli esempi successivi.

2. Il seguente segmento verifica se la lista contiene un elemento maggiore di un valore x dato.

```
trovato = false;
i = 0;
while ( (i<n) && !trovato )
  if ( a[i] > x )
    trovato = true;
  else
    i++;
```

Si noti come la ricerca di una proprietà differente comporti solo la modifica della condizione che determina se cambiare il valore di `trovato` o avanzare nella scansione della lista.

3. Se invece di voler trovare il primo elemento della lista che verifica una determinata proprietà si vuole trovare l'ultimo elemento cella lista che la verifica, è sufficiente modificare l'algoritmo di ricerca in modo da effettuare la scansione della lista partendo dal fondo. Ad esempio, volendo trovare l'ultima occorrenza di un valore dato x, si può usare il seguente segmento di codice:

```
trovato = false;
i = n-1;
while ( (i>=0) && !trovato )
  if ( a[i] == x )
    trovato = true;
  else
    i--;
```

dove la variabile `i` viene inizializzata con l'indice dell'ultimo elemento della lista, e viene decrementata fino al valore 0, indice del primo elemento della lista. Se il valore di `i` diviene negativo il ciclo termina perché non è stato trovato alcun elemento uguale a `x`.

Si noti che il segmento appena presentato non può essere esteso al caso in cui la lista è rappresentata mediante array e informazione tappo. In tal caso, infatti, la scansione deve necessariamente essere fatta "in avanti". Se non si dispone di una variabile che contenga la lunghezza della lista occorre prima eseguire una scansione in avanti che determina tale lunghezza e poi effettuare la ricerca all'indietro.

4. Il seguente segmento verifica se due liste di uguale lunghezza sono uguali. Le due liste sono rappresentate dagli array `lista1` e `lista2` e dal riempimento (comune) `n`. Il valore della variabile `uguali` di tipo **bool** indica se le due liste sono uguali (il valore di `uguali` è **true**) o diverse (il valore di `uguali` è **false**).

```
trovato = false;
i = 0;
while ( (i<n) && !trovato )
  if ( lista1[i] != lista2[i] )
    trovato = true;
  else
    i++;
uguali = !trovato;
```

Il corretto funzionamento del segmento si basa sull'osservazione che, avendo supposto uguale la lunghezza, le due liste sono uguali se e solo se nessun elemento della prima è diverso dal corrispondente elemento della seconda. Il problema si trasforma pertanto nella ricerca della seguente proprietà:

esiste un elemento della prima lista diverso dal corrispondente elemento della seconda lista

e nell'assegnazione alla variabile `uguali` del corrispondente valore di verità *negato*. In altre parole, si usa lo schema di ricerca per risolvere il problema opposto a quello dato e si ottiene il risultato voluto negando il risultato della ricerca.

Osservazione 6.10

L'esempio precedente suggerisce di derivare dallo schema di ricerca un'ulteriore variante, che usa valori opposti per la variabile di tipo **bool**. La variante è la seguente (come al solito diamo solo la versione in cui la lista è rappresentata con array e riempimento, essendo l'altra di facile derivazione):

```
non_trovato = true;
i = 0;
while ( (i<n) && non_trovato )
  if ( a[i] non verifica la proprietà voluta )
    non_trovato = false;
  else
    i++;
```

dove abbiamo cambiato nome alla variabile di tipo **bool** per metterne in risalto il differente ruolo logico. Anche la frase che descrive la condizione cercata è stata modificata per mettere in risalto che questa versione può essere usata in quei casi in cui si vuole stabilire se una proprietà è verificata da *tutti* gli elementi della lista e pertanto il problema si traduce nella ricerca di un elemento che non la verifica.

Se la condizione "`a[i]` *non verifica la proprietà voluta*" risulta sempre falsa, allora il ciclo termina dopo aver esaminato l'ultimo elemento della lista e la variabile `non_trovato` conserva il valore **true** assegnato inizialmente. Questo indica che tutti gli elementi della lista verificano la proprietà voluta. Se invece la condizione "`a[i]` *non verifica la proprietà voluta*" risulta vera in almeno un caso, allora la variabile `non_trovato` assume il valore **false** e questo indica che almeno un elemento della lista non verifica la proprietà voluta.

Il segmento di codice che verifica se due liste sono uguali si può allora scrivere:

```
uguali = true;
i = 0;
while ( (i<n) && uguali )
  if ( lista1[i] != lista2[i] )
    uguali = false;
  else
    i++;
```

dove la variabile *uguali*, che ha lo stesso significato di prima, viene usata direttamente per controllare l'eventuale termine prematuro della ricerca.

5. Lo schema della ricerca di una proprietà si può anche ulteriormente generalizzare al caso in cui la proprietà coinvolge più elementi della lista. È sufficiente solo che sia possibile verificare la proprietà esaminando un elemento per volta, usando variabili ausiliarie per tener conto dello stato della ricerca sugli elementi precedentemente esaminati.

   Per illustrare la questione consideriamo il problema di stabilire se una lista di numeri contiene almeno k elementi uguali a zero. Si tratta evidentemente della ricerca di una proprietà, ma che non riguarda un singolo elemento della lista. D'altra parte noi sappiamo che per contare gli elementi di una lista uguali a zero possiamo usare una scansione con accumulatore che esamina gli elementi della lista per l'appunto uno alla volta.

   La soluzione più naturale al problema è pertanto la fusione dello schema di ricerca con lo schema della scansione con accumulatore. Il segmento che ne risulta è il seguente:

```
n_zeri = 0;
almeno_k_zeri = false;
i = 0;
while ( (i<n) && !almeno_k_zeri ) {
  if ( a[i] == 0 )
    n_zeri++;
  if ( n_zeri >= k )
    almeno_k_zeri = true;
  else
    i++; }
```

   Il lettore è invitato per esercizio a identificare, nel codice dato, gli elementi appartenenti ai due schemi sopra citati. Quello che invece vale la pena di osservare è come, in questo caso, la condizione cercata sia:

```
n_zeri >= k,
```

   e come, al fine di valutarla correttamente, sia necessario aggiungere istruzioni addizionali per esaminare l'elemento corrente della lista `a[i]` prima di aggiornare la variabile `n_zeri`.

6. Un ultimo esempio, che si ispira al caso appena discusso, è la ricerca dell'i-esimo elemento dell'array che verifica una determinata proprietà. Usando ancora una volta l'uguaglianza con un valore x come proprietà da cercare, il segmento che trova l'i-esimo elemento della lista uguale a x è il seguente:

```
n_occorrenze = 0;
trovato = false;
i = 0;
while ( (i<n) && !trovato ) {
  if ( a[i] == x )
    n_occorrenze++;
  if ( n_occorrenze == k )
    trovato = true;
  else
    i++; }
```

   Un confronto con l'esempio precedente evidenzia immediatamente come la logica seguita sia ancora quella di una fusione tra lo schema della ricerca con quello della scansione con accumulatore.

Concludiamo il paragrafo svolgendo un esempio completo che ci consenta di applicare gran parte dei concetti fin qui illustrati.

Esempio 6.17

Specifica del problema Scrivere un programma che riceve in ingresso un numero naturale N1 seguito da una prima lista di N1 numeri, seguita da un numero naturale N2 seguito da una seconda lista di N2 numeri, e che stampa in uscita la lista dei numeri presenti solo in una delle due liste lette.

Casi di test Per chiarire il significato della specifica riportiamo i seguenti casi di test.

stdin	4 1 0 3 7 5 1 8 2 3 6
stdout	0 7 8 2 6

stdin	2 1 0 3 7 5 4
stdout	1 0 7 5 4

stdin	0 5 0 3 7 5 4
stdout	0 3 7 5 4

Soluzione Sulla base dell'esperienza accumulata nei numerosi esempi visti fino ad ora possiamo abbozzare la seguente soluzione:

```
#include <iostream.h>
#include <stdlib.h>
int main() {
  const int MAX = 100;
  int N1, N2, i;
  double l1[MAX], l2[MAX];

  // lettura della prima lista (l1,N1)

  // lettura della seconda lista (l2,N2)

  // stampa degli elementi di (l1,N1) non presenti in (l2,N2)

  // stampa degli elementi di (l2,N2) non presenti in (l1,N1)

  return 0;
}
```

dove i commenti rappresentano dei segmenti di codice da sviluppare che svolgono il compito descritto brevemente nel commento stesso.

I segmenti di lettura delle due liste sono banali e vengono riportati nella versione finale della soluzione. Per quanto riguarda la *stampa degli elementi di (*`l1,N1`*) non presenti in (*`l2,N2`*)* osserviamo che si tratta di scandire la lista (`l1,N1`), stampando ogni elemento che non sia presente in (`l2,N2`). Il segmento può essere pertanto sviluppato nella seguente soluzione non definitiva che usa lo schema di scansione:

```
for ( i=0; i<N1; i++ ) {
  // cerca l1[i] in (l2,N2) e assegna il risultato della ricerca a trovato
  if ( !trovato ) {
    cout << l1[i];
    cout << ' ';
  }
}
```

Come prima il commento rappresenta un segmento da sviluppare. In questo caso si tratta evidentemente della ricerca del valore di `l1[i]` nella lista rappresentata dall'array `l2` e dal riempimento `N2`. Si ha pertanto la seguente soluzione definitiva:

```
for ( i=0; i<N1; i++ ) {
  trovato = false;
  j = 0;
  while ( (j<N2) && !trovato )
    if ( l2[j] == l1[i] )
      trovato = true;
    else
      j++;
  if ( !trovato ) {
    cout << l1[i];
    cout << ' ';
  }
}
```

dove compaiono le nuove variabili `trovato` e `j` che vanno opportunamente dichiarate.

Osservando che il segmento corrispondente al commento *stampa degli elementi di (*`l2`,`N2`*) non presenti in (*`l1`,`N1`*)* è identico a quello appena scritto purché si scambi `l1` con `l2`, e `N1` con `N2`, si ottiene la seguente soluzione finale:

```
#include <iostream.h>
#include <stdlib.h>
int main() {
  const int MAX = 100;
  int N1, N2, i, j;
  double l1[MAX], l2[MAX];
  bool trovato;

  // lettura della prima lista (l1,N1)
  cin >> N1;
  for ( i=0; i<N1; i++ )
    cin >> l1[i];

  // lettura della seconda lista (l2,N2)
  cin >> N2;
  for ( i=0; i<N2; i++ )
    cin >> l2[i];

  // stampa degli elementi di (l1,N1) non presenti in (l2,N2)
  for ( i=0; i<N1; i++ ) {
```

```
      trovato = false;
      j = 0;
      while ( (j<N2) && !trovato )
        if ( l2[j] == l1[i] )
          trovato = true;
        else
          j++;
      if ( !trovato ) {
        cout << l1[i];
        cout << ' ';
      }
    }

    // stampa degli elementi di (l2,N2) non presenti in (l1,N1)
    for ( i=0; i<N2; i++ ) {
      trovato = false;
      j = 0;
      while ( (j<N1) && !trovato )
        if ( l1[j] == l2[i] )
          trovato = true;
        else
          j++;
      if ( !trovato ) {
        cout << l2[i];
        cout << ' ';
      }
    }

    return 0;
  }
```

dove sono stati lasciati i commenti per evidenziare i segmenti che compongono l'algoritmo complessivo.

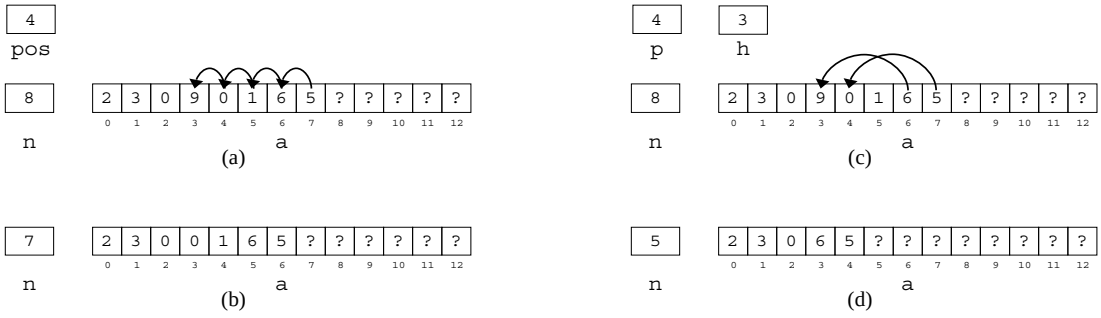


Figura 6.6: Compattamento di liste rappresentate mediante array e riempimento; eliminazione di un elemento: prima (a) e dopo (b); eliminazione di h elementi: prima (c) e dopo (d).

6.7 Altri schemi algoritmici su array

6.7.1 Compattamento di un array

Scopo Lo schema di *compattamento di un array* ha lo scopo di eliminare da una lista rappresentata con array e riempimento un elemento di posizione assegnata.

Pseudo-codice

```
for ( i=pos; i<n; i++ )
  a[i-1] = a[i];
n--;
```

Commenti Lo schema assomiglia a una scansione perché l'eliminazione dell'elemento di posizione assegata (il valore della variabile `pos`) viene ottenuta traslando indietro di una posizione tutti gli elementi successivi. Si tratta quindi di una scansione parziale della lista in cui l'elaborazione consiste nell'assegnare il valore dell'elemento corrente alla cella dell'array che contiene l'elemento precedente. Tale scansione parziale è mostrata in figura 6.6-(a), assumendo di voler eliminare il quarto elemento della lista $< 2, 3, 0, 9, 0, 1, 6, 5 >$. Essa deve procedere "in avanti", cioè partendo dall'indice `pos`, che corrisponde alla posizione `pos`$+1$, all'indice `n`-1, che corrisponde all'ultimo elemento della lista.

Si noti che, poiché la scansione viene effettuata *in avanti*, cioè per indici crescenti, ogni assegnazione sovrascrive l'elemento spostato al ciclo precedente, tranne la prima volta in cui viene sovrascritto l'elemento da eliminare.

Dopo la scansione, lo schema prevede il decremento del riempimento per tener conto del fatto che la lista si è accorciata di un elemento. Il risultato finale dello scompattamento è mostrato in figura 6.6-(b). Si noti che nell'esempio l'elemento di indice 7 che era l'ultimo prima del compattamento non viene modificato, ma viene indicato nella figura come indefinito perché il decremento di `n` lo toglie logicamente dalla lista.

Nel caso si debba eliminare l'ultimo elemento (cioè `pos`$=$`n`) la condizione di uscita del ciclo è subito falsa e non viene spostato nessun valore. Viene però decrementato `n`, ottenendo così l'eliminazione dalla lista dell'elemento voluto. Dopo il decremento, infatti, la casella dell'array che conteneva l'ultimo elemento non fa più parte della lista).

Quest'ultima osservazione mette in evidenza che lo schema può essere usato solo se l'elemento che si vuole eliminare è effettivamente presente nella lista (cioè il valore di `pos` è un numero intero positivo minore o uguale al valore di `n`). In caso contrario, infatti vengono effettuate assegnazioni non corrette e in particolare viene decrementato `n`, quando in effetti la lunghezza della lista non dovrebbe variare perché la posizione indicata non corrisponde ad

alcun elemento della lista. In particolare occorre fare attenzione che lo schema sia usato solo se la lista non è vuota (n>0) perché altrimenti, anche se il ciclo non viene eseguito, viene comunque decrementato n che assumerebbe così un valore negativo senza senso.

Esempi Lo schema di compattamento di un array è già un segmento di codice completo che non richiede una vera e propria istanziazione. Sono tuttavia possibili numerose varianti che possono essere facilmente derivate dallo schema appena illustrato.

1. Nello schema si è supposto che sia data la *posizione* dell'elemento da eliminare. Se invece della posizione fosse dato il valore dell'indice in una variabile index, lo schema verrebbe leggermente modificato cme segue:

   ```
   for ( i=index; i<(n-1); i++ )
     a[i] = a[i+1];
   n--;
   ```

   dove invece di assegnare l'elemento corrente alla casella dell'array precedente viene assegnato l'elemento successivo alla casella corrente dell'array. Naturalmente, poiché l'effetto finale deve essere lo stesso la variabile i usata come indica sull'array assume i valori da index a n-2, invece che da index$+1$ a n-1.

   Variazioni di questo tipo sono molto frequenti nei cicli che operano su array e corrispondono semplicemente a traslare i valori iniziali o finali degli indici coinvolti.

2. Volendo eliminare dalla lista h elementi contigui a partire da quello di posizione p, si può usare il seguente segmento di codice:

   ```
   for ( i=(p+h-1); i<n; i++ )
     a[i-h] = a[i];
   n = n - h;
   ```

   Il segmento si basa sull'osservazione che gli elementi dalla posizione $p+h$, e cioè dall'indice (p+h-1), alla posizione n, cioè all'indice n, devono essere spostati indietro di h caselle. L'azione del ciclo di scansione parziale è mostrata graficamente in figura 6.6-(c), mentre in figura 6.6-(d) viene mostrata la rappresentazione della lista dopo il compattamento.

   Si noti anche a n viene sottratto il valore di h dal momento che alla lista sono stati tolti h elementi. Non è difficile comprendere come il segmento possa essere correttamente applicato solo se inizialmente la lista contiene almeno $p+h-1$ elementi.

3. Se il problema precedente viene posto fornendo la posizione u dell'ultimo elemento da eliminare invece del numero di elementi da eliminare, il segmento si modifica come segue:

   ```
   for ( i=u; i<n; i++ )
     a[i-h] = a[i];
   n = n - h;
   ```

4. Volendo effettuare l'eliminazione di un elemento di posizione assegnata da una lista rappresentata con array e informazione tappo, si tratta di fondere lo schema del compattamento con quello della scansione con informazione tappo, ottenendo il seguente segmento:

   ```
   i = pos;
   while ( a[i] != TAPPO ) {
     a[i-1] = a[i];
     i++;
   }
   a[i-1] = TAPPO;
   ```

Figura 6.7: Scompattamento di liste rappresentate mediante array e riempimento: prima (a) e dopo (b) l'operazione.

dove l'ultima assegnazione è necessaria perché all'uscita del ciclo `a[i]` contiene l'informazione tappo che indicava la fine della lista prima dell'eliminazione, e tale informazione va assegnata alla cella di indice `i`-1 per indicare che la lunghezza della lista si è ridotta di uno.

6.7.2 Scompattamento di un array

Scopo Lo schema di *scompattamento di un array* ha lo scopo di inserire in una lista rappresentata con array e riempimento un elemento di valore e posizione assegnati.

Pseudo-codice

```
for ( i=(n-1); i>=(pos-1); i-- )
  a[i+1] = a[i];
a[pos-1] = x;
n++;
```

Commenti Lo schema svolge il compito opposto al compattamento. L'inserimento dell'elemento di valore assegnato `x` nella posizione assegnata `pos` (cioè nella cella di indice `pos`-1) viene ottenuta in due passi. Dapprima vengono traslati in avanti di una posizione tutti gli elementi dall'indice `pos`-1, che corrisponde alla posizione `pos`, all'indice `n`-1, che corrisponde all'ultimo elemento della lista. Viene quindi assegnato alla cella "liberata" (di indice `pos`-1) il valore `x`. In figura 6.7 viene mostrato graficamente il caso di inserimento nella lista $< 2, 3, 0, 9, 0, 1, 6, 5 >$ del valore 7 in posizione 4.

Si noti che questa volta la scansione deve essere effettuata *all'indietro*, cioè per indici decrescenti. In caso contrario, ogni assegnazione sovrascriverebbe l'elemento successivo non ancora spostato, ottenendo come risultato finale di assegnare a tutte le celle dopo la posizione `pos` lo stesso valore.

Dopo la scansione, lo schema prevede l'incremento del riempimento per tener conto del fatto che la lista si è allungata di un elemento.

Si noti che nel caso dello scompattamento non vi sono vincoli sul valore di `n` ed è possibile ovviamente inserire un nuovo valore anche in una lista vuota. Naturalmente il valore di `pos` deve però essere positivo e non maggiore di `n`$+1$.

Esempi Anche lo schema di scompattamento di un array è già un segmento di codice completo per il quale sono possibili numerose varianti che possono essere facilmente derivate dallo schema appena illustrato. Facciamo di seguito solo pochi esempi poiché molte delle considerazioni fatte per lo scompattamento si applicano, con le dovute modifiche, anche allo scompattamento.

1. Volendo effettuare l'inserimento di un elemento di valore e posizione assegnati in una lista rappresentata con array e informazione tappo, non si può fondere lo schema dello scompattamento con quello della scansione con informazione tappo perchè nei due schemi l'array viene necessariamente scorso in direzione opposta.

Occorre pertanto prima determinare la lunghezza della lista usando il segmento dell'esempio 4 del paragrafo 6.3.3. Poichè la lunghezza della lista è anche l'indice della cella che contiene l'informazione tappo, l'inserimento del valore x in posizione p si ottiene con il seguente segmento:

```
1.  i = 0;
2.  while ( a[i] != TAPPO ) {
3.    i++;
4.  }
5.  // in questo punto i continene la lunghezza della lista;
6.  for ( j=i; j>=(pos-1); i-- )
7.    a[j+1] = a[j];
8.  a[pos-1] = x;
```

dove si è usata la variabile `j` come indice del ciclo **for** perché `i` è usata per memorizzare la lunghezza della lista.

Inoltre, confrontando il segmento dalla riga 6 alla riga 8 con lo schema dello scompattamento si osserva che il ciclo **for** esegue inizialmente un ciclo in più copiando la cella `a[i]` che contiene l'informazione tappo nella cella `a[i+1]`. Questo fa in modo che l'informazione tappo, necessaria per rappresentare correttamente la lista, sia spostata più avanti di una posizione. Il ciclo in più sostituisce quindi l'aggiornamento finale del riempimento che ovviamente manca in questo caso.

2. Assumendo che la lista di numeri rappresentata dall'array `lista` e dal riempimento `len` sia già ordinata in senso crescente, per inserire un nuovo elemento il cui valore è memorizzato nella variabile `x` si può usare il seguente segmento di codice:

```
 1. trovato = false;
 2. i = 0;
 3. while ( (i<len) && !trovato )
 4.   if ( lista[i] >= x )
 5.     trovato = true;
 6.   else
 7.     i++;
 8. for ( j=(len-1); j>=i; i-- )
 9.   lista[j+1] = lista[j];
10. lista[i] = x;
11. len++;
```

Il segmento è composto da su sottosegmenti: quello dalla riga 1 alla riga 7, che indicheremo con S_{1-7}, e quello dalla riga 8 alla riga 11, che indicheremo con S_{8-11}.

S_{1-7} è la ricerca del primo elemento della lista maggiore o uguale a `x`. Al termine di questo segmento il valore di `i` rappresenta l'indice della casella in cui va inserito il valore `x`[8].

S_{8-11} è lo scompattamento dell'array con inserimento in posizione `i`$+1$, cioè proprio nella cella di indice `i`.

3. La soluzione al problema posto nell'esempio precedente può essere migliorata. Osservando che la ricerca del punto di inserimento si può effettuare sia scorrendo la lista "in avanti" che "all'indietro" (se la lista è ordinata la cosa è indifferente), si può evitare di effettuare un doppio ciclo, fondendo la scansione all'indietro dello schema di ricerca con la scansione, sempre all'indietro dello scompattamento. Si ottiene il seguente segmento che risolve il problema dell'inserimento in ordine con un solo ciclo:

[8] Infatti, se la ricerca ha avuto successo il valore `x` va inserito nella cella di indice `i`; se viceversa la ricerca non ha avuto successo (tutti gli elementi sono minori di `x`), allora `i` è uguale a `len` che è proprio l'indice della cella in cui va memorizzato il nuovo valore (cioè in fondo alla lista).

```
trovato = false;
i = n - 1;
while ( (i>=0) && !trovato )
  if ( lista[i] <= x )
    trovato = true;
  else {
    lista[i+1] = lista[i];
    i--;
  }
lista[i] = x;
len++;
```

Capitolo 7

Sottoprogrammi (bozze, v. 2.0)

Finché il compito che un programma deve svolgere si riduce ad una semplice trasformazione di una quantità di dati limitata, il programma può essere formulato come un unico blocco di istruzioni.

Normalmente però, il compito da svolgere è complesso da un punto di vista algoritmico e con riferimento alle capacità dell'esecutore. Tale complessità si traduce nel fatto che la strategia risolutiva deve prevedere lo svolgimento di più sottocompiti. In altre parole il problema è decomponibile in più sottoproblemi, in relazione l'uno con l'altro secondo una certa struttura.

Per facilitare il compito del programmatore è opportuno che anche il programma (cioè la formulazione dell'algoritmo nel linguaggio di programmazione utilizzato) rispecchi la struttura del problema e sia suddiviso in *sottoprogrammi*, cioè in unità ben definite, caratterizzate da una certa autonomia, che possono essere composte secondo la strategia risolutiva prescelta.

In questo capitolo vengono pertanto discussi la definizione e l'impiego dei sottoprogrammi, che rappresentano in tutti i linguaggi di programmazione un eccezionale meccanismo concettuale di astrazione e di strutturazione degli algoritmi.

7.1 Libreria standard

Cominciamo con il definire informalmente un *sottoprogramma* come un insieme di istruzioni eseguibili dal nostro esecutore che portano alla soluzione di un compito ben definito. In altre parole, un sottoprogramma *contiene la descrizione di un algoritmo per quel compito*.

I sottoprogrammi sono entità dotate di una certa autonomia e, oltre ad essere impiegati nella risoluzione di compiti complessi, possono in essere predisposti a priori per svolgere compiti di utilità generale (comuni cioè a problemi anche molto diversi).

In particolare, sono già disponibili un certo numero di sottoprogrammi che formano la cosiddetta *libreria standard* (dall'inglese *standard library*). I sottoprogrammi della libreria standard sono divisi in categorie a seconda della loro funzione e per ciascuna categoria è disponibile, oltre al codice del sottoprogramma già tradotto in linguaggi macchina, anche un *header* file (vedi capitolo 10). Per utilizzare i sottoprogrammi appartenenti a una certa categoria, il programmatore deve aggiungere all'inizio del proprio programma una direttiva del tipo:

```
# include <nome header file>
```

Le principali categorie della libreria standard i nomi dei corrispondenti header file sono tiportati nella tabella 7.1.

Nome dell'header file	Categoria
`ctype.h`	sottoprogrammi di conversione e di test su caratteri
`math.h`	funzioni matematiche in doppia precisione
`stdio.h`	sottoprogrammi di I/O (linguaggio C)
`iostream.h`	sottoprogrammi di I/O (linguaggio C++)
`fstream.h`	sottoprogrammi di I/O su file (linguaggio C++)
`stdlib.h`	sottoprogrammi di utilità generale
`string.h`	sottoprogrammi per la manipolazione di stringhe di caratteri
`time.h`	sottoprogrammi per la manipolazione di informazioni temporali

Tabella 7.1: Principali categorie di sottoprogrammi della libreria standard con i corrispondenti header file.

7.2 Chiamata a sottoprogramma

Quando, ad un certo passo di un algoritmo occorre svolgere un sottocompito per cui è già disponibile un sottoprogramma, possiamo usare una *chiamata al sottoprogramma* per indicare all'esecutore di risolvere il sottocompito usando quel sottoprogramma.

Esistono due tipologie di sottoprogrammi: le *funzioni* e le *procedure* che differiscono per il modo con cui la corrispondente chiamata può essere inclusa in un programma.

La chiamata a un sottoprogramma, indipendentemente se esso sia una funzione o una procedura, ha la seguente forma generale:

nome (*lista dei parametri effettivi*)

dove *nome* è un identificatore che individua il sottoprogramma da usare e *lista dei parametri* è una lista di *parametri effettivi* separati da virgole. Si noti che, come chiariremo meglio nel paragrafo 7.3, *un parametro effettivo pu ò essere un'espressione o una variabile.*

7.2.1 Chiamata a funzione

Le funzioni sono sottoprogrammi che *restituiscono un valore*, detto *risultato* della funzione. La chiamata a una funzione è pertanto *un'espressione* che denota il valore restituito e la sua esecuzione *causa la valutazione di tale espressione*.

Per questo motivo la chiamata a una funzione non è un'istruzione autonoma, ma *deve* comparire all'interno di una qualunque istruzione che preveda la presenza di un'espressione.

Ad esempio, poiché nella categoria delle funzioni matematiche è disponibile una funzione della libreria standard che calcola il logaritmo naturale, assumendo che `x` e `y` sono variabili di tipo **double**, se fosse richiesto di assegnare a `y` il logaritmo naturale dell'espressione `x+3.1`, si può scrivere l'assegnazione:

```
y = log(x+3.1);
```

dove `log(x+3.1)` è la chiamata al sottoprogramma di tipo funzione di nome `log` e l'espressione `x+3.1` tra parentesi è l'unico parametro effettivo richiesto dalla funzione. Si noti come, nell'assegnazione, la chiamata coincida con l'espressione prevista a destra dell'operatore di assegnazione.

Esempio 7.1

Una chiamata alla funzione `log` potrebbe anche comparire nella condizione di uscita di un ciclo **while** o come argomento di un'istruzione di output, come nel seguente codice (`a` è una variabile di tipo **double**):

```
while ( log(a) > 0 ) {
  a = a/2;
  cout << a;
  cout << log(a);
}
```

Si noti che in tutti i casi la chiamata svolge il ruolo di un'espressione il cui valore è il risultato della funzione.

Osservazione 7.1

Durante l'esecuzione del programma, quando l'interprete valuta l'espressione che contiene la chiamata, il sottoprogramma di tipo funzione viene eseguito e il valore restituito viene usato per determinare il valore dell'espressione. Nell'esempio precedente quindi la funzione `log` viene chiamata ed eseguita ogni volta che viene valutata la condizione di uscita del ciclo **while** e ogni volta che viene eseguita l'ultima istruzione di output nel corpo del ciclo. In totale quindi le chiamate a `log` sono pari al doppio del numero di volte che viene eseguito il corpo del ciclo **while** più uno.

7.2.2 Chiamata a procedura

Contrariamente alle funzioni, le procedure sono sottoprogrammi che non restituiscono un valore. La chiamata a una procedura è pertanto un istruzione autonoma che ha l'effetto di produrre l'esecuzione della procedura da parte dell'esecutore.

Ad esempio, nella categoria dei sottoprogrammi di utilità generale, è inclusa la procedura `exit` che provoca la terminazione immediata del programma prima che si raggiunga l'ultima istruzione, e che prevede un unico parametro intero che serve per comunicare all'esecutore il motivo della terminazione [1].

La chiamata al sottoprogramma di tipo procedura `exit` si esprime con l'istruzione:

```
exit(0);
```

Si noti il punto e virgola finale, richiesto dal fatto che la chiamata è un'istruzione autonoma.

Osservazione 7.2

Un procedura non può essere chiamata come una funzione perchè la sua chiamata *non* è un'espressione. Pertanto l'istruzione:

```
x = exit(0);
```

è errata.

Viceversa una funzione può essere chiamata come una procedura, usando un'istruzione autonoma. Ad esempio, tra le funzioni già pronte previste dal linguaggio ce ne è una che permette di far eseguire all'esecutore un comando non previsto dal linguaggio, ma previsto dall'ambiente operativo che gestisce la machhina reale (cfr. capitolo 9). La funzione è denominata `system`, ha come unico parametro una stringa di caratteri che corrisponde al comando da eseguire, e ha come risultato un valore intero che indica se il comando è stato eseguito con successo o meno. La chiamata a *system* dovrebbe pertanto essere inserita in un'espressione, come ad esempio nell'istruzione:

```
stato = system("dir a:");
```

[1] In genere il ruolo di tale parametro non è di interesse nello sviluppo di semplici programmi e pertanto si usa come paramtero effettivo una qualsiasi costante intera (per es. 0).

Tale istruzione causa l'esecuzione del comando `dir a:` da parte dell'ambiente operativo esterno (con conseguente visualizzazione del catalogo corrispondente al floppy disk inserito nel drive `a:`) e l'assegnazione alla variabile `stato` di un valore intero che indica se vi sono stati problemi nell'esecuzione del comando (in genere un risultato nullo indica che non vi sono stati problemi, mentre un risultato diverso da zero indica che si è verificata una anomalia durante l'esecuzione del comando).

Se, tuttavia, non interessasse in alcun modo il risultato della funzione, essa potrebbe essere chiamata con l'istruzione autonoma:

```
system("dir a:");
```

che causa ancora l'esecuzione del comando `dir a:`. Questa volta, tuttavia, il valore restituito dalla funzione viene perduto.

Si noti che la possibilità di chiamare una funzione come una procedura è utile solo in alcuni casi particolari e che in generale non ha senso farlo. Ad esempio, nel caso della funzione `log`, l'istruzione:

```
log(2.03);
```

pur corretta dal punto di vista linguistico, è del tutto inutile perché l'unico effetto della chiamata è il calcolo del logaritmo naturale di `2.03`, ma tale valore, non essendo assegnato a variabili o usato nella valutazione di espressioni più articolate, viene perduto.

7.3 Interfaccia di un sottoprogramma

Nel paragrafo precedente abbiamo introdotto la chiamata a un sottoprogramma, specificando che essa è formata dal nome del sottoprogramma seguito dalla lista dei paramteri effettivi. Nel caso il sottoprogramma sia di tipo funzione, la sua chiamata è un espressione che denota il valore che viene prodotto dall'esecuzione del sottoprogramma, e che va inclusa in un'istruzione che utilizzi tale valore.

È quindi evidente che per chiamare correttamente un sottoprogramma occorre conoscere oltre al suo nome:

- se è una funzione o una procedura
- il tipo del suo risultato nel caso si tratti di un sottoprogramma di tipo funzione;
- il numero e la natura dei suoi parametri;
- il compito che esso svolge.

L'insieme di queste informazioni rappresenta tutto ciò che occorre sapere per usare il sottoprogramma allo scopo di risolvere un problema più ampio. Si noti peraltro che tali informazioni definiscono completamente ciò che il sottoprogramma può fare e pertanto costituiscono anche tutte le informazioni necessarie per scrivere l'algoritmo racchiuso dal sottoprogramma.

Per questo motivo l'insieme di informazioni sopra elencate formano l'*iterfaccia* del sottoprogramma, cioè la descrizione completa delle informazioni condivisa da chi deve utilizzare il sottoprogramma e da chi lo deve implementare. Si noti che l'interfaccia costituisce una sorta di *contratto* tra le due parti.

Da un lato infatti, chi utilizza il sottoprogramma deve rispettare quanto specificato dall'interfaccia perché altrimenti la chimata può dar luogo a comportamenti imprevisti. Ad esempio, se l'interfaccia indica che un sottoprogramma è una funzione che calcola il logaritmo di un numero, la chiamata non può essere usata per ottenere la radice quadrata del numero. Analogamente se il sottoprogramma prevede due paramteri di tipo **double**, esso non può essere chiamato con una stringa di caratteri come unico paramtero.

Dall'altro lato, chi implementa il sottoprogramma deve usare un algoritmo che compia tutto quanto è previsto dall'interfaccia. In caso contrario, infatti, potrebbe accadere che una chiamata corretta (che cioè rispetta l'interfaccia) possa produrre un risultato errato.

7.3.1 Prototipi

Da un punto di vista pratico l'interfaccia di un sottoprogramma viene descritta attraverso due componenti:

- un *prototipo*, che è una dichiarazione prevista dal linguaggio e che include, nell'ordine, le seguenti informazioni:
  - se il sottoprogramma è una funzione o una procedura e, nel primo caso, quale è il tipo del valore restituito;
  - il nome del sottoprogramma;
  - il numero, il tipo dei parametri e se essi possono essere modificati dal sottoprogramma;
  - per ciascun parametro la *modalità di passaggio*, e cioè se il corrispondente parametro effettivo deve essere un'espressione o il nome di una variabile;
- una *descrizione* completa del compito che il sottoprogramma svolge, incluso il significato dei parametri e dell'eventuale valore restituito; tale descrizione, per poter descrivere con accuratezza *cosa* fa il sottoprograma, è normalmente fornita attraverso un commento contenente frasi in linguaggio naturale o altra notazione non prevista dal linguaggio.

Esempio 7.2

Di seguito sono riportate le interfacce dei sottoprogrammi usati nel paragrafo 7.2:

```
double log ( double x );
/* restituisce il logaritmo naturale di x */

void exit ( int exitcode );
/* termina il programma e comunica all'ambiente operativo
   il valore di exitcode */
```

Si noti che:

- prima del commento contenente la descrizione di cosa fa il sottoprograma è riportato il prototipo che comprende:
  - il tipo del risultato se si tratta di una funzione (come nel caso del sottoprogramma `log`), o la parola riservata **void** per indicare che si tratta di una procedura;
  - l'identificatore corrispondente al nome del sottoprogramma;
  - una lista di dichiarazioni di parametri *formali* separate da virgole (negli esempi c'è un solo parametro in entrambi i casi);
- il prototipo è una dichiarazione del linguaggio ed è pertanto normalmente terminata dal un punto e virgola;
- le dichiarazioni relative ai parametri formali assomigliano alle dichiarazioni di variabili;
- la descrizione nei commenti è sintetica e *fa esplicito riferimento* ai parametri, chiarendo in tal modo il loro ruolo nel compito svolto dal sottoprogramma.

Osservazione 7.3

Un confronto tra i prototipi dei sottoprogrammi `log` e `exit` e le corrispondenti chiamate (cfr. paragrafo 7.2) indica che il prototipo di un sottoprogramma descrive *come* deve essere formulata la chiamata. In particolare, i parametri effettivi presenti nella chiamata *devono*:

- corrispondere in numero e tipo ai parametri formali presenti nel prototipo;

- essere espressioni o, alternativamente, nomi di variabili, a seconda della modalità di passaggio dei corrispondenti parametri formali (vedi sottoparagrafi successivi).

Osservazione 7.4

Per chiamare un sottoprogramma, è necessario che in precedenza sia stato aggiunto al codice il relativo prototipo.

Nel caso dei sottoprogrammi della libreria standard, il prototipo viene aggiunto mediante la direttiva `include`, applicata all'header file corrispondente.

Nel caso di sottoprogrammi definiti dal programmatore, per evitare errori nella traduzione del programma è bene aggiungere in testa al codice i prototipi di tutti i sottoprogrammi chiamati (cfr. esempio 7.4).

7.3.2 Passaggio per valore

Si consideri la seguente interfaccia:

```
double pow ( double x, double y );
/* restituisce il risultato di x elevato alla y */
```

di una funzione della libreria standard.

Come nei prototipi dell'esempio 7.2, nella lista dei parametri formali, per ciascun parametro, è indicato il nome preceduto dal tipo. Questo modo di dichiarare i parametri formali indica che al momento della chiamata i parametri formali vengono associati ai *valori* dei corrispondenti paramtri effettivi che devono essere pertanto *espressioni*.

Al termine dell'esecuzione del sottoprogramma l'associazione tra i parametri formali e il valore dei paramtri effettivi viene persa, e in un'eventuale successiva chiamata allo stesso sottoprogramma è possibile stabilire una nuova associazione tra i parametri formali e i valori dei corrispondeti parametri effettivi.

Esempio 7.3

Il programma:

```
# include <iostream.h>

int main () {
  cout << pow(2.5,3.6);
  cout << pow(-4.0,1.2);
  return 0;
}
```

chiama due volte la funzione `pow` stampandone il risultato. Nella prima chiamata il primo parametro di `pow` viene associato al valore reale `2.5` mentre il secondo parametro viene associato al valore reale `3.6` e viene stampato il risultato di $2.5^{3.6}$. Nella seconda chiamata il primo parametro di `pow` viene associato al valore reale `-4.0` mentre il secondo parametro viene associato al valore reale `1.2` e viene stampato il risultato di $-4.0^{1.2}$.

Osservazione 7.5

Questo modo di procedere, che prende il nome di *passaggio per valore dei parametri di formali*, è adatto per quei parametri che rappresentano *i valori che il sottoprograma deve inizialmente ricevere per svolgere il suo compito*.

Ad esempio, con riferimento ai prototipi di `log`, `exit` e `pow`, i valori associati ai parametri formali sono necessari ai tre sottoprogrammi per calcolare, rispettivamente, il logaritmo naturale di un numero reale, comunicare all'ambiente operativo il valore di `exitcode`, calcolare la potenza di due numeri reali.

7.3.3 Passaggio per riferimento

Consideriamo la seguente interfaccia:

```
void leggi_data ( int &giorno, int &mese, int &anno );
/* legge dallo standard input tre numeri e li mette rispettivamente
   in giorno, mese ed anno */
```

Questa volta, nella lista dei parametri formali, tra il tipo e il nome di ciascun parametro compare il simbolo `&`. Esso indica che al momento della chiamata i parametri formali vengono associati alle *celle di memoria* dei corrispondenti paramtri effettivi che devono essere pertanto *nomi di variabili*.

In effetti, dalla descrizione riportata nel commento, si comprende facilmente come i tre parametri della procedura rappresentino dei *parametri di uscita*, e cioè parametri che *ricevono un valore* a seguito dell'esecuzione del sottoprogramma. In altre parole, i parametri effettivi associati ai tre parametri formali (che sono per quanto appena detto delle variabili) vengono *definiti* dal sottoprogramma in modo del tutto analogo a quanto avverrebbe se i valori letti dallo standard input venissero direttamente assegnati ad essi.

Come nel caso del passaggio per valore, al termine dell'esecuzione del sottoprogramma l'associazione tra i parametri formali e le celle di memoria dei paramtri effettivi viene persa, e in un'eventuale successiva chiamata allo stesso sottoprogramma è possibile stabilire una nuova associazione tra i parametri formali e le celle di memoria dei corrispondeti parametri effettivi.

Si noti però che, a differenza da quanto accade nel passaggio per valore, la perdita dell'associazione tra parametri formali e parametri effettivi al termine dell'esecuzione del sottoprogramma non siginifica che vengono persi i valori assegnati dal sottoprogramma. perché essi vengono assegnati direttamente alle celle di memoria corrispondenti ai parametri effettivi, la cui esistenza è indipendente dal sottoprogramma

Esempio 7.4

Il programma:

```
# include <iostream.h>

void leggi_data ( int &giorno, int &mese, int &anno );

int main () {
  int giorno_nascita;
  int mese_nascita;
  int anno_nascita;
  int giorno_oggi;
  int mese_oggi;
  int anno_oggi;
  leggi_data(giorno_nascita,mese_nascita,anno_nascita);
  leggi_data(giorno_oggi,mese_oggi,anno_oggi);
  if ( giorno_nascita==giorno_oggi && mese_nascita==mese_oggi )
    cout << "Oggi e' il tuo compleanno!";
  else
    cout << "Oggi non e' il tuo compleanno!";
  cout << endl;
  return 0;
}
```

chiama due volte la procedura `leggi_data`. Nella prima chiamata vengono letti dallo standard input tre numeri interi che vengono memorizzati, rispettivamente, nelle variabili `giorno_nascita`, `mese_nascita` e `anno_nascita`.

Nella seconda chiamata vengono letti dallo standard input altre tre numeri interi che vengono memorizzati, rispettivamente, nelle variabili `giorno_oggi`, `mese_oggi` e `anno_oggi`.

Osservazione 7.6

Questo comportamento è dovuto al modo di associazione specificato per i tre parametri formali mediante la presenza del simbolo `&`. Tale simbolo indica che l'associazione tra i parametri formali e i parametri effettivi è *per riferimento* che siginifica che il sottoprogramma riceve un riferimento ai parametri effettivi e che, durante l'esecuzione del sottoprogramma, l'uso e la definizione dei parametri formali corrispondono in realtà a a uso e definizione dei parametri effettivi. In altre parole, quando un parametro formale è associato (o passato) per riferimento, esso *diviene un sostituto del parametro effettivo*.

Osservazione 7.7

Si noti che un parametro effettivo associato per riferimento *non* può essere un'espressione perché un'espressione non può essere né usata, né definita. Tali azioni possono essere effettuate solo sulle variabili e quindi un parametro effettivo associato per riferimento *deve* essere necessariamente una variabile.

Osservazione 7.8

Il parametro effettivo associato a un parametro formale passato per riferimento può essere un singolo elemento di un array, perché tale elemento è a tutti gli effetti una variabile del tipo base dell'array.

Ad esempio, volendo laggere una lista di date dallo standard input, si potrebbero usare tre array di interi "paralleli": `giorno` che deve contenere le informazioni relative al giorno di ciascuna data, `mese` che deve contenere le informazioni relative al mese di ciascuna data, e `anno` che deve contenere le informazioni relative all'anno di ciascuna data.

Assumando che la lista di date sia terminata dal fine file, e usando il sottoprogramma `leggi_data`, il segmento di codice che legge la lista è il seguente:

```
n_date = 0;
leggi_data(giorno[0],mese[0],anno[0]);
while ( !cin.eof() ) {
  n+;
  leggi_data(giorno[n_date],mese[n_date],anno[n_date]);
}
```

dove `n_date` è una variabile intera che svolge il ruolo di riempimento dei tre array paralleli.

7.3.4 Parametri formali array

Consideriamo la seguente interfaccia:

```
int conta_zeri ( int l[], int n );
/* restituisce il numero di zeri contenuto negli elementi
   dell'array l di indici da 0 a n-1 inclusi */
```

La funzione `conta_zeri` ha due parametri, il primo dei quali corrisponde a un array di interi. L'esempio mostra che un parametro formale array viene dichiarato aggiungendo dopo il nome una coppia di parentesi quadre "vuote", cioè senza l'estensione.

Osservazione 7.9

Il motivo per cui nel caso di parametri formali array non viene specificata alcuna estensione è che, come abbiamo avuto modo di evidenziare illustrando le modalità di passaggio dei parametri, *un parametro formale non è una variabile autonoma*, ma un nome che assume significato solo quando viene associato al corrispondente parametro effettivo in una chiamata. Un parametro formale array ha perciò di volta in volta l'estensione del corispondente parametro effettivo.

Osservazione 7.10

Con riferimento al prototipo della funzione `conta_zeri` riportato sopra, un esempio di chiamata potrebbe essere il seguente (la chiamata è nell'ultima assegnazione):

```
  .  .  .
const int MAX = 100;
int lista[MAX];
int len;
int tot_zeri;
  .  .  .
/* istruzioni per definire len e gli elementi di lista da 0 a len-1 */
  .  .  .
tot_zeri = conta_zeri(lista,len);
```

dove nella chiamata, come primo parametro effettivo, corrispondente al paramtero formale array, viene indicato semplicemente il nome `lista`, dichiarato in precedenza come array di 100 interi. Durante l'esecuzione della funzione il parametro formale `l` viene pertanto associato all'array `lista` e ha un estensione di 100.

Osservazione 7.11

Poiché nella dichiarazione di un parametro array non è presente il simbolo `&`, il parametro effettivo viene associato al parametro formale *per valore*. Questo però non significa che al parametro formale vengono associati i valori di tutte le caselle dell'array, perché il parametro effettivo, che coincide con nome dell'array senza alcuna estensione e senza alcun indice, non indica *tutto l'array*, ma solo un *riferimento* all'array[2]. Analogamente, anche l'aggiunta della coppia di parentesi vuote nella dichiarazione del parametro formale indica che si tratta di un *riferimento* a un array. Pertanto, al momento della chiamata, il *valore* del parametro effettivo (che, ripetiamo, è un riferimento a un array) viene associato al corrispondente parametro fiormale. Durante la sua esecuzione, il sottoprogramma può poi manipolare i singoli elementi dell'array attraverso tale riferimento.

Esempio 7.5

Consideriamo la seguente interfaccia:

```
int leggi_lista ( int l[] );
/* legge dallo standard input un intero N seguito da N interi;
   restituisce il valore di N e mette gli N interi letti negli
   elementi di l di indice da 0 a N-1 inclusi */
```

La funzione `leggi_lista` ha un parametro formale `l` che, al momento della chiamata, deve essere associato al riferimento a un array di interi.

La chiamata alla funzione `leggi_lista` ha la forma:

```
n = leggi_lista(lista);
```

[2] Si ricordi che nel paragrafo 5.10 abbiamo accennato al fatto che non è consentito manipolare l'intero array all'interno del codice.

Ruolo del parametro	Array	Forma della dichiarazione
ingresso	no	**const** *tipo nome*
uscita o ingresso/uscita	no	*tipo nome*
ingresso	si	**const** *tipo_base nome* `[]`
uscita o ingresso/uscita	si	*tipo_base nome* `[]`

Tabella 7.2: Convenzioni per la dicharazione dei parametri formali in base al loro ruolo.

dove `n` è una variabile intera e `lista` è il riferimento a un array di interi. Si noti che, il meccanismo di passaggio dell'array è lo stesso usato nel caso discusso nell'osservazione 7.10, ma questa volta il sottoprogramma ha il compito di modificare gli elementi del parametro effettivo array. La cosa è possibile proprio perché, nel caso degli array, ciò che viene passato è unicamente un riferimento all'array e pertanto il sottoprogramma, tramite tale riferimento ha accesso direttamente al parametro effettivo i cui elementi possono pertanto essere sia *usati* che *definiti*.

7.3.5 Parametri di ingresso e parametri di uscita

Negli esempi visti fino ad ora abbiamo incontrato tre categorie di parametri formali:

- i parametri passati per valore che non sono array;
- i parametri passati per riferimento che non sono array;
- i parametri array, sempre passati per valore.

Nel primo caso si tratta di *parametri di ingresso*, cioè di parametri che rappresentano *valori* da cui il sottoprogramma deve partire per svolgere il proprio compito.

Nel secondo caso si tratta di *parametri di uscita*, cioè di parametri che rappresentano *variabili* che il sottoprogramma deve modificare durante lo svolgimento del proprio compito. Si noti peraltro che, nel caso i parametri siano *contemporaneamente di ingresso e di uscita*, è in ogni caso necessario usare il passaggio per riferimento perché altrimenti il sottoprogramma non potrebbe modificarli.

Nel terzo caso si tratta di parametri che possono essere sia di ingresso, di uscita o di ingresso/uscita. Nel caso di array infatti è solo possibile passare un riferimento al parametro effettivo e pertanto il sottoprogramma ha sempre diretto accesso agli elementi dell'array.

Occorre pertanto tenere concettualmente distinti il ruolo del parametro (di ingresso, di uscita, di ingresso/uscita), che è legato alla natura del compito che il sottoprogramma deve svolgere, dalla modalità di passaggio che non dipende esclusivamente dal ruolo del parametro, ma è influenzata anche da altri aspetti come ad esempio se si tratta o meno di un array.

Per migliorare la corrispondenza tra il ruolo dei parametri formali e la forma della loro dichiarazione è possibile ricorrere alla parola riservata **const**. La presenza di tale parola nella dichiarazione di un parametro formale non modifica il modo di passaggio (che dipende unicamente dalla presenza del simbolo `&`), ma indica che il sottoprogramma *non può modificare* il valore del parametro, *qualunque sia la modalità di passaggio usata*.

È quindi possibili usare la parola riservata **const** per distinguere i parametri di ingresso da quelli uscita o di ingresso/uscita secondo le regole riportate nella tabella 7.2.

Esempio 7.6

Il sottoprogramma `conta_zeri` ha bisogno di ricevere il valore associato a `n` e di poter usare gli elementi di `l` di indice da `0` a `n-1` per contare il nunero di zeri presenti. Pertanto, volendo usare la parola riservata **const** per evidenziare che entrambi i parametri sono di ingresso, il prototipo andrebbe riscritto come segue:

```
int conta_zeri ( const int l[], const int n );
```

Si noti ancora una volta che entrambi i parametri del sottoprogramma tt conta_zeri sono passati per valore e che:

- nel caso del primo parametro, il nome formale `l` viene associato al valore del riferimento di un array, e tramite tale riferimento il sottoprogramma può usare gli elementi dall'array (cioè del parametro effettivo);
- nel caso del secondo parametro, il nome formale `n` è associato al valore del parametro effettivo (un'espressione di tipo intero), e il sottoprogramma può usare tale valore per svolgere il proprio compito.

Esempio 7.7

Si consideri la procedura `leggi_lista2` con la seguente interfaccia:

```
void leggi_lista2 ( int l[], int &n );
/* legge dallo standard input un intero N seguito da N interi;
   assegna il valore di N a n e mette gli N interi letti negli
   elementi di l di indice da 0 a N-1 inclusi */
```

La procedura è una variante della funzione `leggi_lista` che restituisce la lunghezza della lista in un parametro `n`. Pertanto, entrambi i parametri sono di uscita come evidenziato dall'assenza della parola riservata **const** prima del tipo base di `l`, e dalla presenza del simbolo `&` (passaggio per riferimento) prima del nome `n`.

Si noti che questa volta i parametri del sottoprogramma `conta_zeri` sono passati uno per valore e uno per riferimento, e in particolare:

- nel caso del primo parametro, il nome formale `l` viene associato al valore del riferimento di un array, e tramite tale riferimento il sottoprogramma può definire gli elementi dall'array (cioè del parametro effettivo);
- nel caso del secondo parametro, il nome formale `n` è un sostituto del parametro effettivo, che deve necessariamente essere una variabile di tipo intero; il sottoprogramma può definire tale variabile durante lo svolgimento del proprio compito.

7.4 Implementazione di un sottoprogramma

Oltre a usare sottoprogrammi già pronti mediante l'istruzione di chiamata, il programmatore può *definire* nuovi sottoprogrammi.

Le modalità di tale definizione sono molto semplici in quanto un sottoprogramma non è altro che la descrizione di un algoritmo in una forma autonoma, distinta e parzialmente indipendente dal resto del programma che lo chiama. Un sottoprogramma si definisce pertanto descrivendo l'algoritmo che esso deve utilizzare mediante i normali costrutti del linguaggio di programmazione, e racchiudendo le istruzioni in un blocco associato al sottoprogramma.

Formalmente, la definizione di un sottoprogramma è costituita dal suo prototipo, detto *intestazione* (in inglese *header*, seguito da un blocco di istruzioni, detto *corpo* (in inglese *body*). A differenza della dichiarazione isolata di un prototipo introdotta nei paragrafi precedenti, nella definizione di un sottoprogramma l'intestazione non è terminata da un punto e virgola, ma è immediatamente seguita dalla parentesi graffa che inizia il corpo.

Esempio 7.8

La definizione del sottoprogramma `leggi_data` è la seguente:

```
int leggi_data ( int &giorno, int &mese, int &mese ) {
  cin >> giorno;
  cin >> mese;
  cin >> anno;
}
```

Osservazione 7.12

Il corpo si un sottoprogramma è un normale segmento di programma. La particolarità è che, nel caso del corpo di un sottoprogramma, le variabili di ingresso coincidono con i parametri di ingresso, quelle di uscite coincidono con i parametri di uscita, e quelle di ingresso/uscita coincidono con i parametri di ingresso/uscita.

Nel corpo del sottoprogramma i parametri non devono essere dichiarati in quanto già compaiono nell'intestazione. Eventuali altre variabili che devono essere definite e usate nel corpo del sottoprogramma devono essere normalmente dichiarate all'inizio del blocco.

7.5 Esecuzione dei sottoprogrammi

7.5.1 Chiamata e record di attivazione

Quando l'esecutore incontra una chiamata a sottoprogramma sospende l'esecuzione del flusso di istruzioni corrente e passa ad eseguire la prima istruzione del sottoprogramma.

La situazione è mostrata graficamente nella figura 7.1-(b), dove la freccia evidenzia il trasferimento dell'esecuzione dal flusso di istruzioni corrente alle istruzioni del sottoprogramma.

Quando l'esecutore termina l'esecuzione dell'ultima istruzione del sottoprogramma, riprende l'esecuzione del flusso di istruzioni sospeso in precedenza.

La situazione è mostrata graficamente nella figura 7.1-(c), dove la freccia evidenzia il trasferimento dell'esecuzione dall'ultima istruzione del sottoprogramma al flusso di istruzioni sospeso.

Per quanto riguarda la memoria, durante l'esecuzione di un sottoprogramma è necessario disporre delle celle di memoria corrispondenti alle variabili locali del corpo e ai parametri formali. Tale insieme di celle forma il *record di attivazione* del sottoprogramma.

La situazione è mostrata in figura 7.1-(a), dove si suppone che l'esecutore stia eseguendo istruzioni del programma principale e sia pertanto accessibile un record di attivazione contenente le variabili intere `x`, `y` e `z`.

Il record di attivazione di un sottoprogramma viene *creato* subito prima che il sottoprogramma inizi la sua esecuzione e viene distrutto subito dopo la terminazione del sottoprogramma.

La situazione è mostrata nelle figure 7.1-(b), dove è rappresentato il record di attivazione del sottoprogramma `bin` contenente i tre parametri formali `n`, `k` e `result` e la variabile locale `temp`, e 7.1-(c), dove si suppone che l'esecutore abbia ripreso l'esecuzione del programma principale.

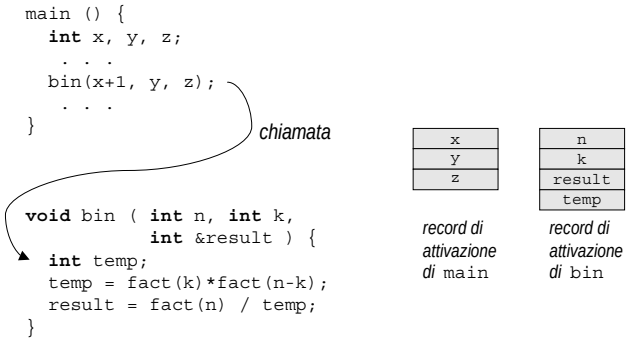
Si noti che il record di attivazione del programma principale viene "congelato" durante l'esecuzione del sottoprogramma, in modo che, alla ripresa del flusso di istruzioni interrotto, le variabili in esso contenute siano di nuovo disponibili.

7.5.2 Passaggio dei parametri per valore e per riferimento

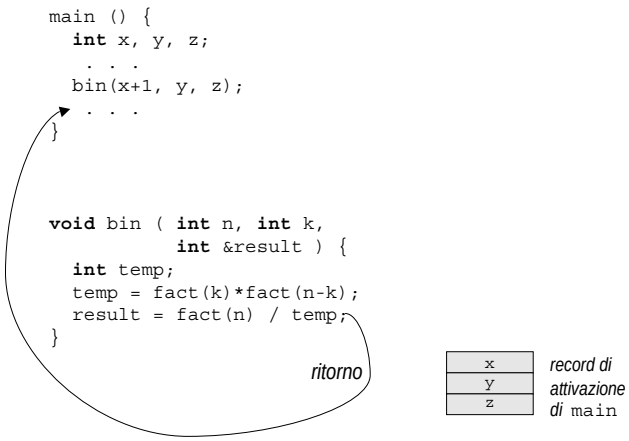
7.5.3 Passaggio dei parametri array

7.5.4 Passaggio del valore di ritorno di una funzione

(a) durante l'esecuzione del programma principale, prima della chiamata al sottoprogramma



(b) durante l'esecuzione del sottoprogramma



(c) durante l'esecuzione del programma principale, dopo la chiamata al sottoprogramma

Figura 7.1: Esecuzione di un sottoprogramma.

Capitolo 8

Complementi del linguaggio di programmazione (bozze, v. 2.0)

8.1 Stringhe di caratteri

I testi che può essere necessario trattare all'interno di un programma (nomi di file, didascalie, intestazioni, ecc.) sono in tutti i casi costituite da liste di caratteri. Aggiungendo caratteri speciali addizionali a quelli usuali è possibile rappresentare come caratteri anche aspetti grafici tipici dei testi come la suddivisione in linee (il corrispondente carattere speciale è denominato *newline* e indicato in C con il simbolo `\n`), l'allineamento a sinistra di colonne di testo (il corrispondente carattere speciale è denominato tab e indicato in C con il simbolo `\t`), ecc.

Quando si parla di liste di caratteri è usuale in informatica usare il termine stringa, sinonimo di lista o sequenza. Useremo pertanto il termine *stringa di caratteri*, o anche semplicemente *stringa*, per indicare il tipo di dato usato per rappresentare le informazioni testuali. Tali informazioni possono consistere in lunghi testi, in singole parole, anche di un solo carattere, o addirittura in un testo vuoto, cioè in una stringa di lunghezza nulla che denominiamo stringa *vuota*.

8.1.1 Rappresentazione di stringhe di caratteri in C

In C le stringhe di caratteri sono rappresentate mediante *array e informazione tappo*. Una variabile di tipo stringa di caratteri è pertanto un normale array che ha per tipo base il tipo **char**. Questo significa che una variabile di tipo stringa di caratteri può contenere solo stringhe di lunghezza pari al numero di caselle dell'array meno uno (una casella dell'array deve essere sempre riservata per contenere l'informazione tappo).

Poiché i caratteri sono codificati attraverso il codice ASCII, le combinazioni di bit effettivamente utilizzate sono minori di quelle potenzialmente disponibili. Questo permette di individuare facilmente una combinazione di bit che possa essere usata come informazione tappo. La scelta è quella di usare la combinazione costituita da 8 zeri come informazione tappo. Per uniformità di terminologia, tale combinazione viene associata a un particolare carattere speciale denominato carattere nullo e indicato in C con il simbolo `\0`.

I meccanismi messi a disposizione dal linguaggio e dai sottoprogrammi di libreria per manipolare le stringhe di caratteri all'interno dell'esecutore presuppongono:

- che tutte le stringhe siano costituite da array di caratteri di lunghezza sufficiente a contenere la stringa da rappresentare;
- che subito dopo la casella contenente l'ultimo carattere della stringa esista almeno un'ultriore casella contenente il carattere nullo.

La verifica di tali condizioni sono a carico del programmatore (o dell'utente che immette i dati nel caso delle operazioni di input) e se non sono soddisfatte la manipolazione delle stringhe da parte dell'esecutore può dare risultati non prevedibili e i corrispondenti programmi risultano non corretti.

Esempio 8.1

I seguenti sono esempi di dichiarazione di variabili di tipo stringa di caratteri:

```
const int MAX_LEN1 = 10;
char parola_breve[MAX_LEN1+1];

const int MAX_LEN2 = 100;
char linea[MAX_LEN2+1]

const int MAX_LEN3 = 10000;
char testo[MAX_LEN3+1];
```

Si noti che nel dichiarare l'estensione degli array, il termine 1 sommato alla costante intera assicura che la lunghezza massima ammessa per una stringa sia proprio pari al valore della costante simbolica. Si tratta di una convenzione di dichiarazione che riduce il rischio di dimenticare che la lunghezza massima consentita per la stringa è pari all'estensione dell'array meno uno.

8.1.2 Costanti

Le costanti di tipo stringa di caratteri si indicano scrivendo la stringa tra doppi apici (il carattere "). Le costanti di tipo stringa sono rappresentate nella memoria dell'esecutore allo stesso modo dei valori memorizzati nelle variabili, e cioè come array di caratteri contenente i caratteri della stringa terminati dal carattere nullo. L'array di caratteri usato per rappresentare le costanti non ha un nome (non deve neppure essere dichiarato), ha lunghezza pari alla lunghezza della stringa costante che esso contiene più uno, per memorizzare il carattere nullo finale.

Osservazione 8.1

Si noti la differenza tra la notazione `'a'` e `"a"`. Nel primo caso la notazione indica il singolo carattere "a minuscola" che occupa un byte in memoria. Nel secondo caso la notazione indica una stringa, costituita dal solo carattere "a minuscolo" seguito dal carattere nullo, che occupa due byte. Come vedremo la seconda notazione può essere usata come parametro effettivo di un sottoprogramma che prevede un parametro di tipo stringa, mentre la prima notazione no.

8.1.3 Lettura di stringhe

La lettura di una parola limitata da spazi o da newline si effettua tramite l'istruzione `cin` in modo analogo agli altri tipi di dato. Assumendo di aver già scritto le dichiarazioni dell'esempio 8.1, l'istruzione:

```
cin >> parola_breve;
```

consuma e trasferisce i caratteri diversi da uno spazio nell'array `parola_breve` fino a che non si incontra uno spazio. Eventuali spazi iniziali presenti sullo standard input vengono consumati senza essere trasferiti nell'array. Lo spazio che fa terminare l'operazione di input non viene consumato. Si noti che, nell'esempio considerato, la lunghezza della parola letta (cioé della sequenza di caratteri consecutivi diversi da uno spazio che viene trasferita nell'array) deve essere minore o uguale a 10 (cioè al valore di `MAX_LEN1`). Come abbiamo già osservato, la verifica corrispondente è a carico dell'utente che immette i dati e non dell'esecutore.

Esempio 8.2

Come anche in altri esempi, per descrivere lo standard input, elencheremo nel seguito la stringa di caratteri da cui è costituito, indicando esplicitamente la presenza di spazi, newline e tab con i simboli `\b`, `\n` e `\t`, rispettivamente, e separando graficamente i caratteri con spazi (che non fanno pertanto parte dell'input).

Ciò premesso, se lo standard input contiene i caratteri:

```
Q u e s t o \b è \b u n \b t e s t o
```

l'istruzione:

```
cin >> parola_breve;
```

consuma i primi sei caratteri dello standard input e li memorizza nelle prime sei celle (dall'indice 0 all'indice 5 inclusi) dell'array di caratteri `parola_breve`. L'istruzione memorizza inoltre il carattere nullo nella settima cella dell'array. Dopo tale istruzione la varabile `parola_breve` contiene pertanto la stringa Questo.

Se la precedente istruzione viene eseguita una seconda volta, la seconda esecuzione consuma senza trasferirlo in memoria il carattere spazio che segue la parola Questo, consuma quindi il carattere `è` memorizzandolo nella prima cella dell'array `parola_breve` (sovrascrivendo il carattere precedentemente memorizzato), e memorizza il carattere nullo nella seconda cella dell'array. Dopo tale seconda istruzione di input la variabile `parola_breve` contiene pertanto la stringa è.

8.1.4 Sottoprogrammi di libreria per la manipolazione di stringhe

La maggior parte delle elaborazioni sulle stringhe di caratteri possono essere effettuate usando un insieme di sottoprogrammi inclusi nella libreria standard che operano su array di caratteri contenenti stringhe terminate dal carattere nullo. Vengono riportati di seguito i sottoprogrammi di uso più comune utilizzati in tutti gli esempi successivi. Alcuni dei seguenti prototipi sono leggermente modificati rispetto alla loro formulazione standard reperibile sui manuali per semplificarne l'uso da parte degl allievi. In tutti gli esempi presentati di seguito si può fare riferimento al prototipo qui presentato.

```
void strcat ( char s1[], char s2[] );
/* concatena la stringa s2 dopo la stringa s1 e pone la stringa
   risultante in s1 */

void strcpy ( char s1[], char s2[] );
/* assegna la stringa s2 alla stringa s1 */

int strlen ( char s[]  );
/* restituisce la lunghezza della stringa s (escluso il carattere nullo
   finale) */

bool strcmp ( char s1[], char s2[] );
/* restituisce true se s1 è diversa da s2, restituisce false se s1 è
   uguale a s2 */
```

Esempio 8.3

Date le dichiarazioni:

```
const int MAX_TESTO = 10000;
const int MAX_PAROLA = 100;
char parola[MAX_PAROLA+1];
char testo[MAX_TESTO+1];
```

Supponendo di voler leggere nella variabile `testo` un testo costituito da parole separate da spazi il cui numero venga fornito subito prima del testo stesso, si può usare l'algoritmo:

```
int n, i;
cin >> n;
strcpy(testo,"");
for ( i=0; i<n; i++ ) {
  cin >> parola;
  strcat(testo,parola);
  strcat(testo," ");
}
```

Si noti che dopo aver concatenato una nuova parola alla parte di testo già letta, memorizzata nella variabile `testo`, viene concatenata una stringa costante costituita da uno spazio per evitare che parole del testo risultino tutte attaccate.

Si noti anche come il segmento di programma riportato sopra segua lo schema di scansione con accumulatore di una lista di lunghezza nota. La cosa risulta evidente confrontando il precedente segmento con il segmento:

```
int n, i, somma;
cin >> n;
somma = 0;
for ( i=0; i<n; i++ ) {
  cin >> x;
  somma = somma + x;
}
```

Nel segmento che legge il testo, l'istruzione:

```
strcpy(testo,"");
```

ha l'effetto di assegnare la stringa vuota alla variabile `testo`. Tale istruzione equivale pertanto a un assegnazione che non può essere espressa col la notazione normale:

```
testo = "";
```

perché tale notazione non ha senso in C in quanto, secondo le regole del linguaggio, l'identificatore `testo` rappresenta un riferimento all'array e non l'array vero e proprio.

Analogamente, nel segmento che legge il testo, le istruzioni:

```
strcat(testo,parola);
strcat(testo," ");
```

hanno l'effetto di concatenare il testo già letto con la nuova parola letta e con uno spazio, e assegnare il risultato della concatenazione alla variabile `testo`. Tale istruzione ha pertanto un ruolo analogo all'assegnazione:

```
somma = somma + x;
```

Anche in questo caso si ricorre all'impiego di un sottoprogramma non potendo usare una normale istruzione di assegnazione.

Esempio 8.4

Volendo verificare prima di effettuare la concatenazione:

```
strcat(testo,parola);
```

che vi sia sufficiente spazio libero nella variabile testo, si può utilizzare il sottoprogramma di tipo funzione stdlen nel modo seguente:

```
if ( strlen(testo)+strlen(parola)+1 <= MAX_TESTO ) {
  strcat(testo,parola);
  strcat(testo," ");
}
```

Esempio 8.5

Scrivere un programma che legge una parola, seguita da un intero N, seguito da un testo di N parole e che stampa il numero di occorrenze della prima parola letta nel testo.

Per chiarire il significato della specifica individuaiamo i seguenti casi di test:

```
input:   prova 8 questa è una prova che riguarda le stringhe
output:  1
input:   testo 11 questo testo è un esempio di testo che contiene due occorrenze
output:  2
input:   xxx 6 questo è un testo di prova
output:  0
```

Il programma risolutivo richiede, dopo la lettura della prima parola e della lunghezza del testo, la scansione del testo e il conteggio di parole uguali alla prima parola letta. Si ottiene pertanto il programma:

```
# include <iostream.h>
# include <string.h>

main () {
  int n, i;
  int const MAX_PAROLA = 100;
  char prima_parola[MAX_PAROLA+1], parola_generica[MAX_PAROLA+1];
  int occorrenze;

  cin >> prima_parola,
  cin >> n;
  occorrenze = 0;
  for ( i=0; i<n; i++ ) {
    cin >> parola_generica;
    if ( !strcmp(prima_parla, parola_generica) ) // se sono uguali
      occorrenze++;
  }

  cout << "Il numero di occorrenze e': " << occorrenze << endl;
}
```

Per riconoscere gl schemi usati si confronti la soluzione con un algoritmo che conta il numero di occorrenze di un valore k in una lista di numeri:

```
cin >> k,
cin >> n;
occorrenze = 0;
for ( i=0; i<n; i++ ) {
  cin >> x;
  if ( x == k ) // se sono uguali
    occorrenze++;
}
```

In entrambi i casi si è usata la scansione di una lista con accumulazione su un contatore e si è proceduto a una fusione della fase di input e della fase di elaborazione della lista.

8.1.5 Output di stringhe

Per aggiungere allo standard output una stringa si può usare una normale istruzione di output. Ad esempio l'istruzione:

```
cout << testo;
```

aggiunge allo standard output la stringa contenuta nella variabile `testo`. Si noti che, come in tutti gli altri casi, afinché la stampa venga effettuata correttamente, la stringa memorizzata nella variable `testo` deve essere terminata dal carattere nullo.

8.1.6 Ordinamento di stringhe

Un'operazione che è spesso richiesta quando si manipolano stringhe è quella di confrontare due stringhe per stabilire quale delle due sia minore rispetto all'ordinamento lessicografico (cioè quello che con termine comune è detto ordinamento alfabetico).

A tal fine non può essere usata la funzione di libreria `strcmp` nella forma descritta in precedenza che consente solo di stabilire se due stringhe sono uguali o diverse. Tuttavia, il prototipo mostrato in precedenza è una forma semplificata del vero prototipo della funzione `strcmp`. Tale forma semplificata è efficace se la funzione deve essere usata per effetuare un test di uguaglianza, ma è necessario fare riferimento al prototipo effettivo di `strcmp` per poter usare tale funzione al fine di stabilire l'ordinamento lessicografico di due stringhe.

Prima di presentare il prototipo completo di `strcmp`, occorre ricordare che nel linguaggio C i valori numerici possono essere usati in sostituzione dei valori logici, con la convenzione che il valore numerico 0 si comporta come il valore logico **falso** e qualunque valore numerico diverso da 0 si comporta come il valore logico **vero**. In altre parole, l'espressione costante:

```
false || true
```

può essere scritta anche usando valori numerici:

```
0 || 1
```

oppure:

```
0 || -1
```

oppure usando qualsiasi valore diverso da 0 al posto del valore true.

Con questa premessa, il prototipo completo della funzione `strcmp` è il seguente:

```
int strcmp ( char s1[], char s2[] );
/* restituisce un valore negativo se s1 precede lessicograficamente s2,
   restituisce 0 se s1 è uguale a s2, restituisce un valore positivo se
   s1 segue lessicograficamente s2 */
```

Si noti come tale prototipo comprenda come caso particolare quello dato precedentemente. Infatti se le stringhe sono diverse (s1 precede s2 o s1 segue s2 nell'ordinamento lessicografico) viene restituito un valore numerico diverso da 0 che equivale al valore logico **vero**, se invece le stringhe sono uguali viene restituito il valore numerico 0 che equivale al valore logico **falso**.

Esempio 8.6

Facendo riferimento al prototipo completo di `strcmp`, la funzione può essere usata per stabilire l'ordinamento lessicografico tra stringhe. Ad esempio, il seguente segmento di codice legge sullo standard input una lista di parole preceduta dalla sua lunghezza e stampa la parola che risulta minore rispetto all'ordinamento lessicografico. Si assuma che siano state dichiarate le variabili `parola`, `parola_minore`, `n`, `i`.

```
cin >> n;
if ( n > 0 ) {
  cin >> parola_minore;
  for ( i=1; i<n; i++ ) {
    cin >> parola;
    if ( strcmp(parola,parola_minore) < 0 )
      strcpy(parola_minore,parola);
  }
  cout << "La parola minore è :" << parola_minore << "\n";
}
else
  cout << "La lista e' vuota, non esiste una parola minore!\n";
```

8.2 Input/output con formato (a caratteri)

Quando sono state introdotte le istruzioni di input e di output, gli effetti della loro esecuzione sono stati spiegati descrivendo lo standard input e lo standard output come sequenze di valori. Tale spiegazione è sufficiente fin tanto che il formato dei dati di ingresso e di uscita non risulta troppo complesso. Vi sono tuttavia alcuni casi in cui l'effetto delle istruzioni di input e di output non si può spiegare descrivendo lo standard input e lo standard output come sequenze di valori, ma bisogna ricorrere a una descrizione che corrisponda meglio alla realtà.

8.2.1 Output

Cominciamo con il caso più semplice: lo standard output.

Se l'esecuzione di un programma produce il seguente output:

```
x y
0 0
1 2
2 5
3 10
4 17
```

è certamente possibile dire che esso aggiunge allo standard output la sequenza di 12 valori (tra parentesi il tipo del valore):

```
x  (carattere)
y  (carattere)
```

```
0  (numero)
0  (numero)
1  (numero)
2  (numero)
2  (numero)
5  (numero)
3  (numero)
10 (numero)
4  (numero)
17 (numero)
```

È tuttavia ovvio che descrivere lo standard output in questo modo non dice tutto, e di fatto la sequenza di istruzioni:

```
cout << 'x';
cout << 'y';
cout << 0;
cout << 0;
cout << 1;
cout << 2;
cout << 2;
cout << 5;
cout << 3;
cout << 10;
cout << 4;
cout << 17;
```

genera il seguente output:

```
xy001225310417
```

Per produrre infatti l'output su due colonne sopra riportato bisogna aggiungere, nei punti opportuni, dei caratteri newline, dei caratteri tab o dei caratteri bianchi.

Un ulteriore esempio, che mostra come sia insufficiente la descrizione dello standard output in termini di sequenza di valori, è il caso di stampa di un valore reale. Se lo stantdard output fosse una sequenza di valori, l'istruzione:

```
cout << 3.000018563;
```

dovrebbe generare l'output:

```
3.000018563
```

Invece, eseguende la precedente istruzione di uscita, viene aggiunto allo standard output il numero:

```
3.00002
```

I due esempi illustrati mostrano come, in casi non banali, sia necessario descrivere le istruzioni di output come istruzioni che aggiungono allo standard output caratteri e non valori. Di conseguenza lo standard output risulta essere in realtà una sequenza di caratteri e non di valori.

La distinzione può sembrare capziosa perchè noi siamo abituati a rappresentare i valori attraverso sequenze di caratteri (la parole sono sequenze di lettere, i numeri sono sequenza di cifre con l'aggiunta di caratteri come il segno e il punto decimale, ecc.) e dunque tendiamo a non dare importanza alle regole di conversione da valori a caratteri e viceversa che invece vanno prese in esplicita considerazione quando si tratta di programmare un esecutore automatico.

Tipo dell'argomento di cout	Caratteri aggiunti allo standard output
carattere	lo stesso carattere
stringa	la stessa stringa
numeri interi	rappresentazione posizionale in base 10
numeri reali	rappresentazione posizionale in base 10 con punto decimale e numero prefissato di cifre frazionarie (tipicamente 5)

Tabella 8.1: Regole di conversione valori-caratteri impiegate nell'istruzione di output.

In effetti, la distinzione è sostanzialmente inutile in tutti i casi in cui le regole di conversione tra valori e caratteri sono semplici e intuitive, ed è per questo motivo che, limitandoci a tali casi semplici, abbiamo potuto introdurre le istruzioni di output in termini di sequernze di valori.

Volendo però dare ragione degli effetti delle istruzioni di output in casi più complessi come quelli riportati sopra, occorre fare riferimento in modo esplicito alle regole di conversione tra valori e caratteri utilizzate dall'istruzione cout.

Tali regole sono riportate nella tabella 8.1 per i tipi di dato che possono essere usati come argomento dell'istruzione di output.

8.2.2 Spiegazione degli esempi

È possibile ora dare ragione del perchè nel primo esempio riportato in precedenza l'output sia:

```
xy001225310417
```

Applicando infatti alla sequenza di istruzioni di output le regole riportate nella tabella 8.1, si verifica che vengono aggiunti allo standard output i caratteri `'x'` e `'y'` seguiti dalle cifre che rappresentano i numeri 0, 0, `1`, `2`, `2`, `5`, `3`, `10`, `4`, `17`. Poiché le istruzioni non lo prevedono non vengono inseriti spazi tra tali valori, e tantomeno vengono inseriti dei caratteri newline o dei tab per incolonnare i valori stessi.

Analogamente, nel secondo esempio il valore `3.000018563` viene convertito in rappresentazione posizionale con sole 5 cifre frazionarie e nella conversione viene operato un arrotondamento per eccesso sulla quinta cifra frazionaria.

8.2.3 Input

Anche per quanto riguarda l'input la descrizione in termini di sequenza di valori è adatta a descrivere solo situazioni relativamente semplici come ad esempio la lettura di una sequenza di numeri interi, o la lettura di una sequenza di parole.

Si consideri tuttavia il seguente segmento di codice:

```
const int MAX = 10;
char stringa[MAX];
for ( i=0; i<MAX; i++ )
  cin >> stringa[i];
for ( i=0; i<MAX; i++ )
  cout << stringa[i];
```

Se lo standard input contiene i caratteri:

```
a b c d \b \n e f g h \b \n i j k l \b \n
```

Tipo dell'argomento di cin	Caratteri usati per acquisire il valore
carattere	primo carattere diverso da uno spazio
stringa	prima stringa di caratteri consecutivi diversi dallo spazio
numeri interi	prima stringa di cifre consecutive, eventualmente precedute dal segno; se il primo carattere diverso dallo spazio non è una cifra il comportamento è indefinito
numeri reali	prima stringa di cifre consecutive eventualmente comprendenti il punto decimale, ed eventualmente precedute dal segno o dal punto decimale (viene in ogni caso presa in condsiderazione solo la prima occorrenza del punto decimale); se il primo carattere diverso dallo spazio non è una cifra il comportamento è indefinito

Tabella 8.2: Regole di conversione caratteri-valori impiegate nell'istruzione di input.

sullo standard output viene generata la sequenza di caratteri:

```
abcdefghij
```

contrariamente a quanto ci aspetteremmo, dal momento che sullo standard input erano presenti anche caratteri bianchi e *newline*.

Il motivo di questo comportamento è che anche gli effetti delle istruzioni di input devono essere descritti in termini di acquisizione di sequenze di caratteri (e non di valori) secondo determinate regole di conversione. Nel caso dell'input tali regole cercano di rendere il comportamento delle istruzioni quanto più simile possibile all'acquisizione di valori. Tuttavia vi sono casi in cui tale spiegazione è insufficiente e bisogna applicare esplicitamente le regole di conversione riportate nella tabella 8.2 per i tipi di dato che possono essere usati come argomento dell'istruzione di input.

È opportuno fare alcune osservazioni sul contenuto della tabella.

Prima di di tutto occorre ricordare che le istruzioni di input consumano i caratteri seguendo l'ordine con cui essi sono presenti sullo standard input. Questo significa che quando nella tabella viene indicato che la conversione riguarda il primo carattere con certe caratteristiche, i caratteri eventualmente precedenti vengono saltati. Essi cioè vengono consumati senza produrre effetto.

La seconda osservazione riguarda la caratteristica comune a tutti i casi riportati in tabella: gli spazi presenti sullo standard input vengono sempre saltati. Occorre a questo proposito chiarire che per spazio non si intende solo il carattere spazio bianco, ma anche tutti quei caratteri che producono una separazione quando vengono aggiunti allo standard output. Oltre allo spazio bianco sono pertanto considerati spazi anche il carattere di newline e il carattere tab.

Il fatto che gli spazzi vengano sempre saltati sembra rendere impossibile effettuare operazioni come: contare gli spazi bianchi presenti sullo standard input all'inizio di una riga, contare le righe di un testo fornito in input, ecc. In effetti, tali operazioni non possono essere fatti usando l'istruzione di input, a meno di non cambiare le regole di conversione che essa applica. Tale modifica è possibile e s rimanda a un manulae di C++ per ulteriori dettagli. Introdurremo comunque nel seguito un'istruzione che permette di leggere un intera linea sullo standard input, inclusi eventuali spazi presenti. Usando tale istruzione è possibile effettuare qualunque operazione riguardi gli spazi presenti sullo standard input.

Venendo alle ultime due righe della tabella, occorre innazitutto sottolineare come, quando si leggono dati numerici, la presenza sullo standard input di caratteri che non siano cifre, o segno, o, solo nel caso di numeri reali, punto decimale, vada assolutamente evitata perché può produrre il mal funzionamento del programma.

Infine, per chiarire ulteriormente le regole applicate nella lettura di dati numerici, è opportuno discutere alcuni esempi.

Consideriamo il segmento di programma:

```
int a;
double x;
char ch;
cin >> a;
cin >> x;
cin >> ch;
```

Se sullo standard input sono presenti i caratteri:

```
\b 1 0 . 0 3 . 0 3 \n
```

la prima istruzione di input salta lo spazio bianco iniziale e assegna alla variabile `a` il valore `10` consumando i primi tre caratteri sullo standard input; la seconda istruzione assegna alla variabile `x` il valore `0.03` consumand altri tre caratteri sullo standard input; la terza istruzione assegna alla variabile `ch` il valore `'.'` (carattere punto) e consuma un ulteriore carattere; successive istruzioni di input partono dal carattere `'0'` successivo al secondo punto.

Se sullo standard input sono presenti i caratteri:

```
5 0 8 \n - 9 8 \n . 0 3 \n
```

la prima istruzione di input assegna alla variabile a il valore 508 consumando i primi tre caratteri sullo standard input; la seconda istruzione salta il carattere newline e assegna alla variabile x il valore -98, consumando complessivamente altri quattro caratteri sullo standard input; la terza istruzione salta il secondo carattere newline e assegna alla variabile ch il valore '.' (carattere punto), consumando altri due ulteriori caratteri; successive istruzioni di input partono dal carattere '0' successivo al punto.

8.2.4 Altre istruzioni di input

Sono disponibili altre istruzioni di input che hanno una forma simile alla chiamata di sottoprogramma, con la differenza che il nome del sottoprogramma deve essere preceduto dal nome cin separato da un punto. Tali istruzioni, analgamente ai sottoprogrammi possono essere descritte fornendo un prototipo che indica se si tratta di funzioni o di procedure e se vi sono parametri.

Getline

L'istruzione getline viene descritta attraverso il seguente prototipo:

```
void getline ( char line[], int maxlen );
/* consuma tutti i caratteri presenti sullo standard input fino al primo
   carattere newline, memorizza i caratteri consumati (escluso il newline)
   nell'array line terminandoli con un carattere nullo; se la lunghezza
   della linea supera maxlen-1 caratteri, memorizza nell'array solo i primi
   maxlen-1 caratteri seguiti dal carattere nullo */
```

e viene indicata nel programma con la notazione:

```
cin.getline(nome_array,valore_intero);
```

dove `nome_array` e `valore_intero` sono i parametri effettivi della chiamata, rispettivamente di tipo array di caratteri e intero.

L'istruzione getline consente di leggere dallo standard input anche gli spazi bianchi e i tab eventualmente presenti su una linea. Essa non consente di leggere esplicitamente i newline, ma poiché ogni sua esecuzione consuma esattamente un newline dopo l'ultimo carattere letto, essa di fatto consente di tenere conto anche dei newline presenti sullo standard input.

Esempio 8.7

Il segmento di codice:

```
int n, i, j, n_bianchi;
const int MAX_LINEA = 250;
char linea[MAX_LINEA+1];
cin >> n;
n_bianchi = 0;
for ( i=0; i<n; i++ ) {
  cin.getline(linea,MAX_LINEA+1);
  j = 0;
  while ( linea[j] != '\0' ) {
    if ( linea[j]==' ' ) n_bianchi++;
    j++;
  }
}
cout << n_bianchi;
```

conta i caratteri bianchi presenti in un testo di n linee (il valore di n viene fornito sullo standard input prima del testo).

Se quindi lo stantard input contiene i caratteri:

```
5 \b a a \n b b \b \n c \b c \n \b \b \b \n e e e \n
```

il programma, dopo aver letto il valore 5 nella variabile n, legge nell'array linea: la prima volta la stringa `" aa"`, la seconda volta la stringa `"bb "`, la terza volta la stringa `"c c"`, la quarta volta la stringa `"  "` (3 spazi bianchi), la quinta volta la stringa `"eee"`, e stampa il valore `6`.

Se invece lo stantard input contiene i caratteri:

```
3 \n \n c c c \n
```

il programma, dopo aver letto il valore `3` nella variabile `n`, legge nell'array linea: la prima volta la stringa `""` (stringa vuota), la seconda volta ancora la stringa `""`, la terza volta la stringa `"ccc"`, e stampa il valore `0`.

Eof

La lettura di dati terminati da un'informazione tappo ha il problema che non sempre è facile determinare quale valore usare come tappo perchè tale valore va escluso a priori tra quelli di interesse per il programma. Il problema è particolarmente grave nel caso di lettura di testi perchè da un lato in questi casi non è ragionevole assumere di conoscere la lunghezza del testo prima di elaborarlo (si pensi ad esempio di dover fornire il numero di parole presenti in una relazione di 10 pagine!), e dall'altro non è facile trovare un carattere che non venga mai usato in nessun testo per assegnargli il ruolo di informazione tappo.

Per questo motivo tutti i linguaggi introducono la possibilità di inserire sullo standard input un carattere speciale che ha il compito di segnalare la fine dell'input stesso. Per vari motivi il programma non può controllare la presenza di tale carattere con un test esplicito, ma deve ricorrere a un istruzione descritta dal seguente prototipo:

```
bool eof ( );
/* restituisce true se nell'ultima operazione di lettura è stato consumato
   il carattere di fine file, restituisce false altrimenti */
```

Come nel caso di getline, tale istruzione viene invocata mediante una chiamata preceduta da cin separato da un punto. Questa volta, tuttavia, il prototipo indica che si tratta di una funzione e la chamata non è pertanto un'istruzione autonoma, ma va inserita all'interno di un'espressione.

Esempio 8.8

Si supponga di voler contare il numero di spazi in un testo (vedi esempio precedente), ma che non si disponga del numero di righe da cui il testo e costituito. Se dopo aver digitato il testo si aggiunge dopo l'ultimo newline il carattere speciale di fine file (che si ottiene si quasi tutti i sistemi premendo insieme i tasti `Ctrl` e `z`), si può usare l'istruzione eof per far terminare il ciclo che legge il file, ottenendo il segmento di codice:

```
int i, j, n_bianchi;
const int MAX_LINEA = 250;
char linea[MAX_LINEA+1];
n_bianchi = 0;
cin.getline(linea,MAX_LINEA+1);
while ( !cin.eof() ) {
  j = 0;
  while ( linea[j] != '\0' ) {
    if ( linea[j]==' ' ) n_bianchi++;
    j++;
  }
  cin.getline(linea,MAX_LINEA+1);
}
cout << n_bianchi;
```

Si osservi che il segmento segue lo schema della lettura di una lista terminata da un'informazione tappo che riportiamo di seguito per comodità di lettura:

```
acquisisci in a[0] il primo valore;
i = 0;
while ( a[i] != TAPPO ) {
  /* a[i] contiene l'elemento di posto i+1 */
  i++;
  acquisisci in a[i] il valore successivo ;
}
/* i contiene la lunghezza della lista */
```

Si noti che nel caso in esame la lista da leggere è la lista delle righe del testo e che manca l'indice i perché le righe vengono immediatamente elaborate, e l'array linea usato per leggerle viene riutilizzato ad ogni lettura. Si tratta in effetti di un esempio di fusione tra input ed elaborazione secondo uno scema del tipo:

```
acquisisci in il primo valore;
while ( input non terminato ) {
  elabora valore acquisito;
  acquisisci il valore successivo;
}
```

Si noti infine come, in accordo alle proprietà di lettura con informazione tappo, è necessario effettuare una lettura prima di cominciare il ciclo **while** per verificare se il testo non sia vuoto (lo standard input potrebbe contenere subito il carattere di fine file). Inoltre, la lettura interna al ciclo è posta in fondo al suo corpo, in modo da controllare subito dopo di non aver incontrato il fine file (condizione di uscita del ciclo).

8.2.5 Input/ouput da file

Fino ad ora abbiamo visto come acquisire dati dallo standard input e aggiungere dati sullo standard output. Vi sono tuttavia casi in cui è più conveniente acquisire dati da un file o aggiungere dati a un file. Si pensi ad esempio al caso in cui, in input o in output, occorre gestire grandi quantità di dati che non possono essere gestiti manualmente, o quando i dati vanno conservati per un uso successivo.

I moderni linguaggi di programmazione risolvono il problema mediante meccanismi del tutto simili a quelli usati per gestire lo standard input e lo standard output in modo che i programmi risultino largamente indipendenti dall'origine e dalla destinazione effettiva dei dati.

Questo risultato si è ottenuto osservando che lo standard input e lo standard output sono sequenze di caratteri e che pertanto risultano equivalenti a un file che contenga esclusivamente caratteri. Un tale file, denominato *text file*, può anche essere generato o letto dagli utenti mediante programmi già pronti come ad esempio il programma *notepad* nel caso di windows o il programma *vi* nel caso di UNIX o LINUX.

Sulla base di questa equivalenza, sono stati predefiniti due tipi di dato, denominati `ifstream` e `ofstream`, che permettono di dichiarare variabili che possono essere associate a file, usate allo stesso modo di `cin` e `cout`, rispettivamente. In altre parole sulle variabili `ifstream` si possono usare le operazioni:

- `>>` e `getline` per acquisire dati da un file, con le stesse regole viste in precedenza, caratteri, stringhe e numeri;
- l'operazione `eof` per individuare la fine del file.

Viceversa, sulle variabili `ofstream` si può usare l'operazione `<<` per aggingere caratteri ad un file, sempre con le stesse regole già viste.

Per usare i tipi `ifstream` e `ofstream` occorre includere nel programma il file header `fstream.h`. Inoltre prima di effettuare qualunque operazione sulle variabili di entrambi i tipi occorre associarle ad un file mediante l'operazione `open` il cui prototipo è:

```
void open ( char nome_file[] );
/* associa nome_file (stringa di caratteri contenente il nome
   secondo le convenzioni del file system) alla variabile ifstream
   o ofstream a cui e' applicata l'operazione; l'associazione puo'
   terminare con successo o con fallimento (vedi operazione
   is_open);
   nome_file può comprendere anche il path relativo o assoluto del
   file; in caso di path relativo la directory di default viene
   decisa secondo regole che dipendono dall'implementazione del
   linguaggio usata */
```

È anche disponibile un'istruzione complementare ad `open` che annulla l'associazione di una variabile di tipo `ifstream` o `ofstream` con un file. L'istruzione ha il prototipo:

```
void close ( );
/* annulla l'associazione con un file della variabile di tipo ifstream
   o ofstream su cui viene applicata */
```

Anche se la terminazione di un programma annulla ogni associazione a file di variabili di tipo `ifstream` o `ofstream`, è buona norma di programmazione annullare esplicitamente l'associazione tra una variabile e un file con l'istruzione close quando tale associazione non è più utile.

Esempio 8.9

Il segmento di codice:

```
int n;
int x;
int somma;
cin >> n;
somma = 0;
for ( i=0; i<n; i++ ) {
  cin >> x;
  somma = somma + x;
}
cout << somma;
```

legge dallo standard input una lista di numeri precedeuta dalla sua lunghezza, ne calcola la somma e aggiunge tale somma allo standard output. Volendo effettuare la stessa operazione su una lista di memorizzata sul file `c:\my_data\lista.txt`, il segmento di codice varrebbe così modificato:

```
ifstream in_file;
int n;
int x;
int somma;
in_file.open("c:\my_data\lista.txt");
in_file >> n;
somma = 0;
for ( i=0; i<n; i++ ) {
  in_file >> x;
  somma = somma + x;
}
cout << somma;
in_file.close();
```

Si noti che, ovviamente, all'inizio del file che contiene il codice va inserita la direttiva:

```
# include <fstream.h>
```

Inoltre si assume che nella directory `c:\my_data` esista un text file denominato lista.txt e che tale file contiene una lista di numeri preceduta dalla sua lunghezza. In altre parole, il contenuto del file `lista.txt` deve coincidere con quello che l'utente avrebbe dovuto digitare sul dispositivo associato allo standard input nel caso del primo segmento di codice.

Si noti infine che le istruzioni `open` e `close` hanno la forma di chiamate a procedura, con la differenza che devono essere precedute dal nome della variabile di tipo `ifstream` su cui vengono applicate, separata da un punto. Si tratta di una situazione del tutto analoga a quelle viste in prcedenza per le operazioni `getline` e `eof` applicate su `cin`.

Esempio 8.10

Nel caso di variabili `ofstream`, l'istruzione di associazione al file ha una forma del tutto simile al caso `ifstream`. Il seguente segmento di codice genera il text file `c:\my_data\tabella.txt` contenente la tabella della moltiplicazione tra cifre decimali.

```
ofstream out_file;
int i, j;
```

```
out_file.open("c:\my_data\tabella.txt");
for ( i=0; i<10; i++) {
  for ( j=0; j<10; j++ ) {
    out_file << "\t";
    out_file << i*j;
  }
  out_file << "\n";
}
out_file.close();
```

8.2.6 Gestione degli errori

L'esecuzione dell'istruzione `open` su una variabile di tipo `ifstream` richiede che, nella directory indicata (esplicitamente se nell'istruzione è incluso il path assoluto, implicitamente se è indicato un path relativo o non è indicato alcun path), il file da associare esista e che l'utente abbia il diritto di leggerne il contenuto. È ovvio infatti che il file deve già esistere per poter essere letto.

Se tale condizione non si verifica, l'istruzione `open` non produce alcuna associazione della variabile `ifstream` e le successive operazioni su di essa non sono lecite. Poiché l'eventualità che l'associazione non vada a buon fine non è rara (basta ad esempio che il file o una directory del path sia stata rinominata dopo che il programma è stato compilato) è opportuno che il programma controlli la buona riuscita dell'istruzione `open` prima di proseguire.

A tal fine è disponibile l'istruzione `is_open` caratterizzata dal seguente prototipo:

```
bool is_open ( );
/* restituisce true se la variabile di tipo ifstream o ofstream su cui
   e' è validamente associata ad un file, restituisce false altrimenti */
```

Esempio 8.11

Quando si effettua l'associazione di una variabile di tipo `ifstream` o è opportuno farlo usando un segmento di codice simile al seguente:

```
ifstream in_file;
in_file.open("c:\my_data\lista.txt");
if ( !in_file.is_open() ) {
  cout << "errore --- apertura del file fallita\n";
  exit(1);
}
```

Si noti che la verifica di corretta associazione delle variabili di tipo `ofstream` si effettua in maniera del tutto analoga. In questo caso, tuttavia, l'eventualità di errore è molto minore perché l'associazione di un file a una variabile `ofstream` produce la creazione del file se il file non esiste e la sua cancellazione e riscrittura se esiste. La possibilità di errore è quindi legata solo all'impossibilità di creare il file che è un evento raro (accade quando il disco è pieno o quando l'utente non ha i diritti necessari a scrivere sul disco).

Esempio 8.12

Forniamo ancora un esempio completo di utilizzo delle variabili `ifstream` e `ofstream`.

Il seguente programma legge un testo e lo ristampa su righe con lunghezza massima assegnata. Il testo viene letto da un file e viene ristampato su un altro file. I nomi dei due file vengono forniti dall'utente sullo standard input. Inoltre eventuali errori vengono segnalati da messaggi stampati sullo standard output.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>

int main()
{

// -------------------------- DICHIARAZIONI -------------------------

  const int MAX_PAROLA = 100;
  char parola[MAX_PAROLA];

  const int MAX_NOME_FILE = 35;
  char nome_in_file[MAX_NOME_FILE], nome_out_file[MAX_NOME_FILE];

  int linea_len, max_len;

  ifstream in_file;
  ofstream out_file;

// ----------------------- INIZIALIZZAZIONI -----------------------

  cout << "Dammi i nomi dei file di input e di output: ";
  cin >> nome_in_file;
  cin >> nome_out_file;

  in_file.open(nome_in_file);
  if ( !in_file.is_open() ) {
    cout << "errore --- non posso aprire il file di input\n";
    system("PAUSE");
    exit(1);
  }
  out_file.open(nome_out_file);
  if ( !out_file.is_open() ) {
    cout << "errore --- non posso aprire il file di output\n";
    system("PAUSE");
    exit(1);
  }

  cout << "Dammi la lunghezza massima delle linee del file di output: ";
  cin >> max_len;

// --------------------------- ALGORITMO --------------------------

  linea_len = 0;
  in_file >> parola;
  while ( !in_file.eof() ) {
```

```
      in_file >> parola;
      if ( linea_len + strlen(parola) > max_len ) {
        out_file << "\n";
        linea_len = 0;
      }
      out_file << parola << " ";
      linea_len = linea_len + strlen(parola);
    }
    out_file << "\n";

  // ---------------------------- CHIUSURA --------------------------

    in_file.close();
    out_file.close();

    return 0;
  }
```

Vale la pena fare le seguenti osservazioni:

- l'algoritmo è una fusione di input, elaborazione e output e segue lo schema dell'elaborazione di una lista (di parole) con informazione tappo (il carattere di fine file); l'elaborazione consiste essenzialmente nel ricopiare le parole lette in uscita separandole con uno spazio;
- la variabile `linea_len` è usata come accumulatore, con la variante che viene riazzerata ogni volta che viene suparata la lunghezza massima ammessa per una linea del testo in uscita;
- i nomi dei file di ingresso e di uscita sono forniti dall'utente e letti in due variabili stringa; le due variabili stringa sono usate nelle istruzioni di `open` per associare le variabili `in_file` e `out_file` ai corrispondenti file;
- grazie all'impiego di variabili `ifstream` e `ofstream` il programma acquisisce dati dallo standard input (i nomi dei file, la lunghezza massima) e da un file (il testo di ingresso) e genera dati sullo standard output (i messaggi di errore) e su un file (il testo di uscita).

8.3 Array multidimensionali

Nel linguaggio C è possibile dichiarare e manipolare variabili array con più di una dimensione. Si parla in tal caso di array multidimensionali.

La dichiarazione di un array multidimensionale è del tutto simile a quella degli array ad una dimensione (detti pertanto monodimensionali), con l'estensione relativa a ciascuna dimensione racchiusa tra parentesi quadre.

Ad esempio le dicharazioni:

```
const int MAX_DIM1 = 20;
const int MAX_DIM2 = 10;
const int MAX_DIM3 = 10;
const int MAX_DIM4 =  4;
char      matrice[MAX_DIM2][MAX_DIM1];
int       cubo[MAX_DIM3][MAX_DIM2][MAX_DIM1];
double    ipercubo[MAX_DIM4][MAX_DIM3][MAX_DIM2][MAX_DIM1];
```

dichiarano, rispettivamente, un array bidimensionale (normalmente indicato con il termine matrice) di $10 \times 20 = 200$ caratteri, un array tridimensionale di $10 \times 10 \times 20 = 2000$ interi, un array a quattro dimensioni di 4x10x10x20=8000 reali.

Nelle istruzioni le celle di un array multidimensionale si indicano elencando dopo il nome dell'array tutti gli indici, ciascuno racchiuso tra parentesi quadre. Come per gli array monodimensionali, gli indici sono espressioni semplici o composte, costanti o non costanti.

Ad esempio, assumendo che `i`, `j`, `k` sono variabili intere opportunamente definite, le istruzioni:

```
cout << matrice[i+1][j];
cubo[0][0][k] = cubo[1][0][k] + cubo[2][0][k] ;
ipercubo[i+j][i-j][k][k] = 0.001;
```

rispettivamente:

- aggiungono allo standard output il valore memorizzato nella cella dell'array `mat` appartenente alla riga di indice `i+1` e alla colonna `j`;
- assegnano alla cella dell'array `cubo` di indici `0`, `0`, `k`, la somma dei valori contenuti nelle celle dello stesso array di indici `1`, `0`, `k`, e `2`, `0`, `k`;
- assegnano alla cella dell'array `ipercubo` di indici `i+j`, `i-j`, `k`, `k` il valore constante `0.001`.

8.4 Parametri di scambio array multidimensionali

Come abbiamo visto un array monodimensionale è caratterizzato da un tipo base (il tipo dei valori contenuti nelle sigole celle) e da un'estensione (il numero di celle da cui è costituito l'array).

Secondo le regole del linguaggio, anche gli array a più dimensioni sono considerati array monodimensionali, ma con la proprietà che il loro tipo base è a sua volta un array, eventualmente anche esso a più dimensioni. In altri termini, una matrice viene trattata come un array il cui tipo base è un array monodimensionale, un array tridimensionale viene trattatyo come un array il cui tipo base è una matrice, e cosi' via.

Questa caratteristica è ricca di conseguenze per quanto riguarda il passaggio di array multidimensionali ai sotto-programmi.

Consideriamo innanzi tutto la dichiarazione (nel seguito useremo sempre array le cui celle elementari sono interi, ma il discorso è indipendente da questo):

```
int a1[MAX_DIM1];
```

Il tipo base di `a1` è **int** e la sua estensione è il valore di `MAX_DIM1`. Consideriamo quindi la dichiarazione:

```
int a2[MAX_DIM2][MAX_DIM1];
```

Questa volta il tipo base è `int[MAX_DIM1]` (volendo significare con questa notazione che il tipo base è un array di `MAX_DIM1` interi) e l'estensione è il valore di `MAX_DIM2`. Consideriamo infine la dichiarazione:

```
int a3[MAX_DIM3][MAX_DIM2][MAX_DIM1];
```

In questo caso il tipo base è `int[MAX_DIM2][MAX_DIM1]` (volendo significare con questa notazione che il tipo base è una matrice di `MAX_DIM2` $\times$ `MAX_DIM1` interi) e l'estensione è il valore di `MAX_DIM3`.

Da questi esempi risulta che il tipo base di un array è caratterizzato dal tipo delle celle elementari specificato all'inizio della dichiarazione e dalle estensioni, specificate in fondo alla dichiarazione, esclusa la prima che indica il numero di elementi dell'array (inteso come array monodimensionale di qualcosa).

Supponiamo ora, per fissare le idee, di avere una procedura di nome `proc1` con un paramtro array monodimensionale di nome `par1`. Il prototipo di tale procedura è (assumiamo sempre senza perdita di generalità che le celle elementari siano degli interi):

```
void proc1 ( int par[] );
```

Tenendo presente le osservazioni fatte sul tipo base di un array multidimensionale, si può comprendere come il protitipo di una procedura `proc2` con un parametro array bidimensionale `par2` va dichiarato nel modo seguente:

```
void proc2 ( int par2[][MAX_DIM1] );
```

dove `MAX_DIM1` è il numero di elementi del tipo base dell'array (che è per l'appunto un array monodimensionale di interi).

Analogamente, il protitipo di una procedura `proc3` con un parametro array tridimensionale `par3` va dichiarato nel modo seguente:

```
void proc3 ( int par3[][MAX_DIM2][MAX_DIM1] );
```

dove `MAX_DIM2` $\times$ `MAX_DIM1` è il numero di elementi del tipo base dell'array (che è per l'appunto un array bidimensionale di interi).

Nella chiamata a sottoprogramma, il parametro effettivo corrispondente a un array multidimensionale deve essere un array con pari numero di dimensioni. Le estensioni devono essere coincidenti con quelle dichiarate nel prototipo del sottoprogramma, tranne la prima estensione che può essere qualsiasi.

Si vede facilmente come il caso di array monodimensionali sia un caso particolare di quello descritto, dove il tipo base dell'array non è a sua volta un array e l'estensione può essere qualsiasi.

Si noti infine che le regole descritte hanno un interessante conseguenza. Si consiteri l'array bidimensionale `matrice` dichiarato in precedenza. Indicando con `i1`, `i2` due espressioni qualsiasi di tipo intero di valore compatibile con le estensioni dichiarate per le due dimensioni di matrice, la notazione:

```
matrice[i2][i1]
```

denota, come già sappiamo, la casella dell'array bidimensionale associata agli indici `i1`, `i2`. D'altra parte, per quanto detto il precedenza, l'array matrice può essere visto come un array monodimensionale di array monodimensionali, a dunque la notazione:

```
matrice[i2]
```

denota l'elemento di indice i2 di tale array monodimensionale che è appunto un array monodimensionale di `MAX DIM1` elementi. Più esattamente, in coerenza con le regole già viste sui riferimenti agli array, tale notazione denota il riferimento a un array monodimensionale di `MAX_DIM1` elementi. Si noti che questo è perfettamente coerente con il significato della notazione a due indici, se si immagina di applicare gli indici nell'ordine da sinistra verso destra. Se infatti la notazione `matrice[i2]` equivale a riferire il nome di un array monodimensionale, la notazione `matrice[i2][i1]` rappresenta la cella di indice `i1` di tale array, che è appunto la cella di indice `i1` appartenente alla riga della matrice di indice `i2`.

Si noti anche che, per quanto detto, assumendo che la procedura `leggi_array_mono` abbia il prototipo:

```
void leggi_array_mono ( char stringa[] );
/* legge dallo standard input una stringa di caratteri e la memorizza
   nell'array sringa terminandola con un carattere nullo */
```

il segmento di codice (la variabile `matrice` è quella dichiarata precedentemente):

```
for ( i=0; i<MAX_DIM2; i++ )
  leggi_array_mono(matrice[i]);
```

è perfettamente legittimo e siginifica che il sottoprogramma `leggi_array_mono` riceve come parametro effettivo di volta in volta le singole righe della matrice, trattate come array monodimensionali di caratteri.

Capitolo 9

Ambiente operativo (bozze, v. 2.0)

Capitolo 10

Struttra dei programmi e strumenti di sviluppo (bozze, v. 2.0)

10.1 Prerequisiti

La seguente trattazione ha carattere riepilogativo rispetto a molti argomenti oggetto del corso di Elementi di Informatica. In particolare è richiesta la conoscenza approfondita dei seguenti argomenti:

- Algoritmo ed esecutore;
- Linguaggio di programmazione e corrispondente macchina astratta;
- Segmento di programma, variabili di ingresso e uscita di un segmento;
- Sottoprogrammi e meccanismi di passaggio dei parametri di scambio;
- Procedimento di traduzione dal programma sorgente al programma eseguibile: proeprocessore, compilatore, collegatore;
- Architettura di Von Neumann e struttura del linguaggio macchina.

10.2 Struttura dei programmi

Allo scopo di risolvere un problema attraverso l'esecutore associato ad un linguaggio di programmazione, l'algoritmo risolutivo deve essere formulato in ogni dettaglio, fino a giungere ad un programma completo secondo le regole del linguaggio scelto.

Nel caso il problema da risolvere richieda l'impiego di un algoritmo complesso o di un numero elevato di algoritmi cooperanti, il programma risultante è formato da un numero molto elevato di istruzioni. Per mantenere la complessità del programma entro limiti accettabili, la sua stesura richiede pertanto il ricorso a forme di strutturazione che consentano di dividere il codice in parti che possano essere manipolate con una certa indipendenza.

Si possono distinguere due aspetti della strutturazione di un programma. Il primo può essere indicato col termine *strutturazione logica* perché riguarda la divisione in parti dell'*algoritmo* che il programma descrive. La seconda può essere indicato col termine *strutturazione fisica* perché riguarda la divisione in parti del *testo* del programma. Si noti che la strutturazione fisica coincide con la divisione in file del programma.

In generale, la struttura fisica di un programma non coincide con la la sua struttura logica, anche se in parte dipende da essa. Nel seguito riassumeremo pertanto prima i meccanismi fondamentali di strutturazione logica e

successivamente illustreremo quelli di strutturazione fisica, mostrando come questi ultimi devono tenere conto della struttura logica del programma.

10.2.1 I sottoprogrammi

Il principale meccanismo di strutturazione logica disponibile in tutti i linguaggi di programmazione è la possibilità di isolare il segmento di codice che descrive un algoritmo in un sottoprogramma.

Nel riassumere come tale meccanismo viene usato per strutturare logicamente un programma complesso, è opportuno soffermarsi sulle relazioni che intercorrono tra il segmento di codice incapsulato in un sottoprogramma e il resto del codice. Tali relazioni sono completamente identificate dai seguenti elementi:

- le variabili di ingresso e di uscita del corpo del sottoprogramma;
- i valori consumati dallo standard input e prodotti sullo standard output dalle istruzioni del corpo del sottoprogramma;
- la descrizione precisa di come vengono assegnati i valori alle variabili di uscita e di come vengono prodotti i valori sullo standard output, a partire dai valori contenuti inizialmente nelle variabili di ingresso e dai valori letti dallo standard input.

Normalmente le variabili di ingresso e di uscita del corpo di un sottoprogramma coincidono con i parametri di scambio (rispettivamente di ingresso e di uscita) del sottoprogramma. Altre variabili di appoggio, definite e usate nel corpo, sono *locali* al sottoprogramma. Tali variabili vengono dichiarate all'interno corpo del sottoprogramma e vengono create e distrutte all'inizio e al termine dell'esecuzione del sottoprogramma stesso. Esse risultano pertanto disponibili solo durante l'esecuzione del sottoprogramma, e corrispondono a variabili diverse in esecuzioni diverse del sottoprogramma.

10.2.2 Le variabili globali

Esistono tuttavia altre variabili che, pur non essendo locali (non sono cioè dichiarate all'interno del sottoprogramma) sono peraltro usate o definite nel corpo. Tali variabili sono ovviamente di ingresso per il corpo se usate prima di essere definite, e possono (e normalmente sono) di uscita se definite.

Queste variabili vengono dette *globali* o *statiche* perché esse possono essere usate o definite da più sottoprogrammi, e perché esse, a differenza di quelle locali, sono disponibili all'esecutore indipendentemente dal fatto che il sottoprogramma che le usa o le definisce sia attivo.

Ad esempio, nel seguente programma, la variabile `valori_letti` è globale, e viene usata dal `main` e usata e definita da `leggi_lista_int`.

```
# include <iostream.h>

void leggi_lista_int ( int lista[], int &n );

int valori_letti = 0;

int main () {
  const int MAX_LEN = 100;
  int L1[MAX_LEN];
  int n_int;

  while ( valori_letti < MAX_LEN )
    leggi_lista_int(L1,n_int);
}

void leggi_lista_int ( int lista[], int &n ) {
  int i;
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  valori_letti = valori_letti + n;
}
```

Come mostrato dall'esempio, le variabili globali devono essere dichiarate all'esterno dei sottoprogrammi e per questo motivo esse vengono anche indicate come *esterne* dallo standard del C. Una volta dichiarate esse possono essere usate o definite da tutti i sottoprogrammi che seguono testualmente la dichiarazione.

Naturalmente anche l'impiego delle variabili globali deve rispettare la regola per cui nessuna variabile può essere usata prima di essere stata definita. Questo è il motivo per cui normalmente le variabili globali vengono inizializzate all'atto della loro creazione (usando cioè un inizializzatore nella dichiarazione). Infatti, poiché le variabili globali possono essere usate e definite in sottoprogrammi diversi, il codice che le manipola non è sempre facilmente identificabile, né è facile ricostruire la sequenza dinamica con cui vengono usate o definite. L'inizializzazione, ove possibile, rappresenta pertanto un modo per assicurare il rispetto della regola sopra menzionata. Nell'esempio, l'inizializzazione al valore 0 rende corretto il riferimento a `valori_letti` sia nel `main` che in `leggi_lista_int`.

Le ultime considerazioni evidenziano peraltro la generale difficoltà a controllare la sequenza di valori assunti da una variabile globale e la conseguente difficoltà ad analizzare la correttezza dei programmi che ne fanno uso. E' consigliabile pertanto evitare un uso generalizzato delle variabili globali, limitandolo a quei casi (piuttosto rari) in cui esso risulta assolutamente necessario o in cui rappresenta un chiaro vantaggio rispetto a soluzioni alternative.

10.2.3 Struttura logica di un programma

Per quanto detto, la struttura logica (dell'algoritmo) di un programma consiste pertanto in un insieme di sottoprogrammi e in un insieme di variabili globali.

Ogni sottoprogramma comprende una lista di parametri di ingresso, una lista di parametri di uscita, un insieme di variabili locali e un insieme di variabili globali usate o definite. Come caso particolare, ciascuna lista e ciascun insieme può essere vuoto. Il corpo del programma usa e definisce i parametri e le variabili globali nel rispetto delle regole discusse in precedenza.

In particolare, ogni sottoprogramma interagisce con il resto del programma ricevendo valori nei parametri di ingresso e nelle variabili globali usate, e restituendo valori nei parametri di uscita e nelle variabili globali definite. Ogni

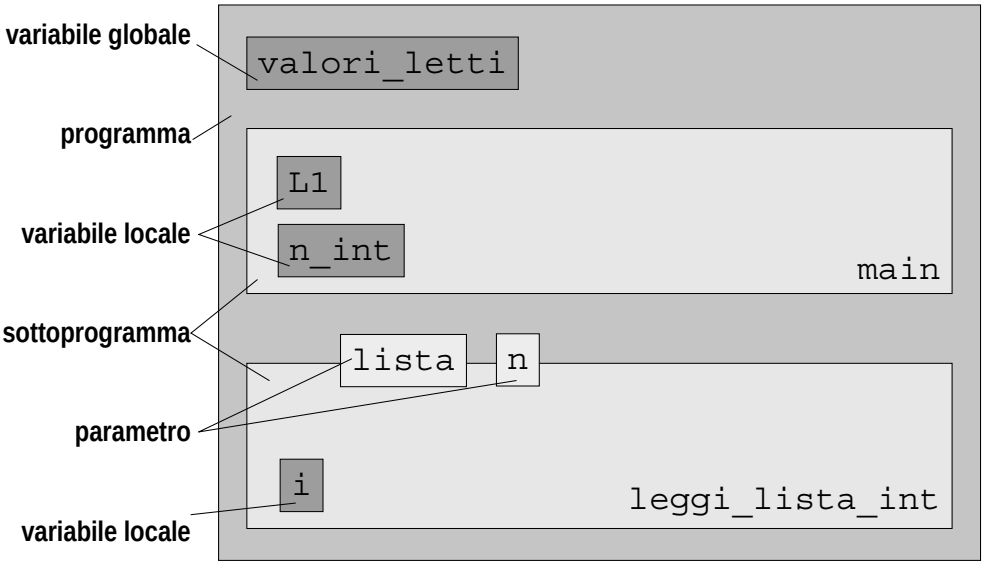


Figura 10.1: Struttura di un programma.

sottoprogramma può inoltre interagire con l'ambiente esterno consumando valori dallo standard input o producendo valori sullo standard output.

In figura 10.1 è mostrata in termini grafici la struttura logica del programma riportato inp precedenza.

10.2.4 Organizzazione in file di un programma

La struttura logica illustrata nel paragrafo precedente rappresenta una decomposizione dell'algoritmo in parti e non implica necessariamente anche una suddivisione fisica del codice in più file. In altre parole, il codice di un programma strutturato in sottoprogrammi potrebbe essere interamente contenuto in un unico file. In tal caso la suddivisione logica, pur risultando utile sul piano dell'organizzazione concettuale, non comporterebbe vantaggi sotto il profilo della gestione del codice.

Per consentire invece una reale suddivisione del codice in più file sono disponibili due meccanismi, di natura molto diversa, entrambi molto utili nella gestione del codice di programmi complessi.

Prima di descrivere il funzionamento dei due meccanismi e di discutere come essi contribuiscano alla gestione del codice, è opportuno ricordare il procedimento attraverso il quale un programma formulato in un linguaggio ad alto livello (programma *sorgente*), che fa riferimento ad un esecutore astratto, viene trasformato in un equivalente programma (programma *eseguibile*) adatto ad essere eseguito sull'esecutore reale.

La trasformazione principale da programma sorgente a programma eseguibile è effettuata dal compilatore che traduce le istruzioni del linguaggio ad alto livello in linguaggio macchina. Tuttavia la compilazione è normalmente preceduta da una fase in cui il testo del programma scritto dal programmatore viene modificato da un processore di testo detto preprocessore. In questa fase preliminare non viene fatta alcuna analisi del programma sorgente, né viene effettuata alcuna traduzione, ma, sulla base di direttive, si procede alla sostituzione, all'aggiunta o alla eliminazione di testo. Il testo così prodotto dal preprocessore è quello effettivamente analizzato e tradotto dal compilatore.

La compilazione, inoltre, è seguita da un ulteriore fase in cui il prodotto della compilazione (detto file oggetto) che è codice macchina incompleto viene integrato con altro codice macchina in modo da generare il programma eseguibile. Il programma che effettua tale integrazione è detto collegatore (linker).

La compilazione, per sua natura, prevede la traduzione di *un* testo scritto nel linguaggio sorgente in codice macchina contenuto in *un* file oggetto. Essa pertanto non prevede l'integrazione di file distinti. L'integrazione è invece possibile sia attraverso le funzionalità messa a disposizione dal preprocessore, sia attraverso quelle messe a disposizione dal linker.

Divisione in file mediante la direttiva `#include`

La direttiva del preprocessore `#include` causa la sostituzione della riga di testo che la contiene con l'intero contenuto del file ad argomento della direttiva. Ad esempio, se il file sorgente `prova.cpp` contenesse il seguente testo:

```
# include "prototipi.h"

int valori_letti = 0;

int main () {
  const int MAX_LEN = 100;
  int L1[MAX_LEN];
  int n_int;

  while ( valori_letti < MAX_LEN )
    leggi_lista_int(L1,n_int);
}
```

e il file `prototipi.h` contenesse la linea:

```
void leggi_lista_int ( int lista, int &n );
```

la compilazione del file `prova.cpp` verrebbe effettuata sul seguente testo prodotto dal preprocessore (invocato automaticamente prima del compilatore):

```
void leggi_lista_int ( int lista, int &n );

int valori_letti = 0;

int main () {
  const int MAX_LEN = 100;
  int L1[MAX_LEN];
  int n_int;

  while ( valori_letti < MAX_LEN )
    leggi_lista_int(L1,n_int);
}
```

Nella sua semplicità l'esempio mostra l'uso della direttiva `#include` per separare il testo del programma (l'ultimo riportato, che è poi quello effettivamente tradotto dal compilatore) in file distinti. Si tratta di una separazione puramente testuale che può essere usata per spezzare in più file il codice da compilare. Solitamente il meccanismo dell'inclusione viene usato per mettere in un file separato le dichiarazioni di tipi, le variabili globali e i prototipi che vanno ripetute in più di un'unità di compilazione (vedi paragrafo successivo), in modo da non doverle scrivere più volte. Oltre al risparmio legato ad un unica scrittura, l'uso dell'inclusione per inserire delle dichiarazioni in un unità di compilazione ha il vantaggio di garantire che il compilatore confronti il codice compilato separatamente con le medesime dichiarazioni in modo da rilevare eventuali incongruenze.

Unità di compilazione

Il secondo meccanismo disponibile per separare in più file il codice di un programma, detto *compilazione separata* fa riferimento alla capacità del compilatore di compilare *separatamente* solo una parte del codice, purché tale parte di codice verifichi alcune regole di completezza. Una porzione di codice che non sia un programma completo, ma che possa essere compilato separatamente dal resto del programma viene detta *unit à di compilazione*.

Le regole che deve rispettare un'unità di compilazione per essere tale dipendono dal linguaggio di programmazione e si rimanda pertanto al manuale di riferimento del linguaggio per una loro descrizione dettagliata. Tuttavia, in prima approssimazione tali regole si possono riassumere nelle seguenti due proprietà:

- le istruzioni che definiscono singole componenti logiche del programma (come per esempio sottoprogrammi e variabili globali) devono essere interamente contenute in un'unità di compilazione;
- un'unità di compilazione deve contenere le dichiarazioni corrispondenti a tutti gli identificatori che in essa compaiono.

Un'unità di compilazione può essere suddivisa in più file usando il meccanismo dell'inclusione. Per comodità si identifica tuttavia l'unità di compilazione con il file che viene usato come argomento per invocare il compilatore, anche se di fatto il testo dell'unità è in realtà quello generato dal preprocessore prima della compilazione vera e propria.

Un'unità di compilazione viene esaminata e tradotta dal compilatore in modo completamente separato dal resto del codice e senza conoscere nulla di esso. Ad esempio, volendo separare in più unità di compilazione il codice corrispondente alla struttura logica mostrata in figura 10.1, si può decidere di mettere in una prima unità (file `prova.cpp`) il programma principale e la variabile globale `valori_letti`, e in una seconda unità (file `leggi_lista.cpp`) il sottoprogramma `leggi_lista_int`. Il contenuto del file `prova.cpp` è quello riportato all'inizio del paragrafo 10.2.4, mentre quello del file `leggi_lista.cpp` è riportato di seguito.

```
# include <iostream.h>
# include "prototipi.h"

extern int valori_letti;

void leggi_lista_int ( int lista[], int &n ) {
  int i;
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  valori_letti = valori_letti + n;
}
```

È opportuno sottolineare quanto segue:

- ciascuna unità di compilazione è in realtà composta da più file (`prova.cpp` e `prototipi.h` la prima; `prova.cpp`, `iostream.h` e `prototipi.h` la seconda) che vengono integrati dal preprocessore prima di procedere alla compilazione di ciascuna unità;
- l'identificatore `valori_letti`, che identifica una variabile globale, è dichiarato in entrambe le unità di compilazione per consentire ad entrambe la possibilità di riferirlo; le due dichiarazioni non sono tuttavia equivalenti: la definizione vera e propria (comprendente anche l'inizializzazione della variabile) è quella contenuta nella prima unità, mentre la dichiarazione contenuta nella seconda unità, detta *allusione* in alcuni manuali e caratterizzata dall'uso della parola chiave **extern**, indica solo il tipo della variabile e che essa è definita in un'altra unità.

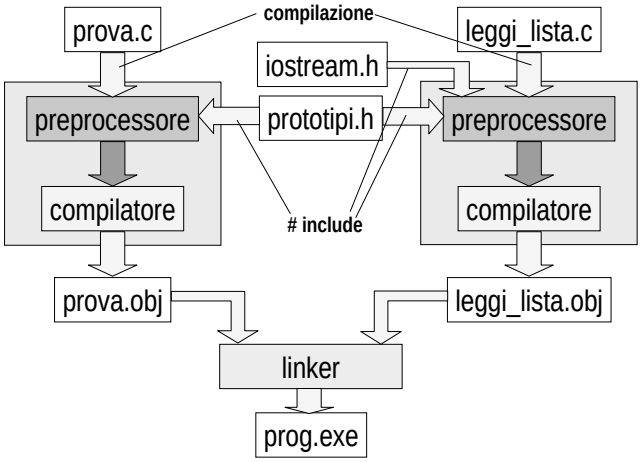


Figura 10.2: Procedimento di traduzione di un programma dal codice sorgente (suddiviso in file) al programma eseguibile.

Come già visto in precedenza, e come illustrato in maggiore dettaglio di seguito, una volta compilate le due unità e generati i rispettivi file oggetto, il collegatore provvede a integrare le due unità e a generare un unico file eseguibile contenente tutte le istruzione e le variabili globali necessarie per l'esecuzione sull'esecutore reale.

Per chiarire ulteriormente come le diverse componenti fisiche del codice vengono integrate fino a giungere al programma eseguibile, la figura 10.2 mostra graficamente il procedimento di generazione del file eseguibile `prog.exe` a partire dai 4 file in cui è suddiviso il codice.

Mentre l'uso tipico del meccanismo di inclusione mira a non duplicare le dichiarazioni di carattere globale, la compilazione separata viene utilizzata per raggruppare in file distinti le componenti logiche del programma in modo da sfruttare la sua struttura logica nella gestione del codice. Oltre a evidenti vantaggi pratici nel caso di programmi di grandi dimensioni, e a consentire lo sviluppo in parallelo di parti diverse da parte di più programmatori, la compilazione separata ha un'ulteriore vantaggio. Consentendo l'integrazione delle parti compilate separatamente a livello di file oggetto non richiede di avere accesso necessariamente al codice in linguaggio sorgente per generare un programma (cosa invece richiesta dal meccanismo dell'inclusione che opera sul testo prima della compilazione). Tale possibilità consente tra l'altro di ricompilare solo una piccola parte del programma quando bisogna apportare modifiche limitate a poche unità di compilazione, e di utilizzare sottoprogrammi sviluppati da terze parti che non possono o non desiderano mettere a disposizione il corrispondente codice sorgente.

10.3 Esecuzione sulla macchina reale

Quando si scrive un programma in un linguaggio ad alto livello, si formula un algoritmo con riferimento all'esecutore astratto definito dalle regole semantiche del linguaggio usato (detto anche *macchina astratta*). Tale esecutore non corrisponde normalmente ad una macchina reale, ma esso viene realizzato mediante l'uso combinato di una macchina reale basata sul modello di Von Neumann, e di programmi speciali che "traducono" il programma formulato nel linguaggio ad alto livello in un equivalente programma formulato nel linguaggio direttamente eseguibile dalla macchina reale.

Le regole che descrivono le capacità operative dell'esecutore astratto definito dal linguaggio sono state ampiamente descritte illustrando i meccanismi fondamentali dei linguaggi di programmazione. Essi sono essenzialmente riassumibili nei seguenti punti:

- Capacità di creare e distruggere variabili in memoria; le variabili possono essere *globali*, nel qual caso vengono create all'inizio dell'esecuzione del programma e vengono distrutte quando il programma termina, o *locali*, nel qual caso vengono create e distrutte quando viene, rispettivamente, iniziata e terminata l'esecuzione del blocco all'interno del quale sono dichiarate.

- Capacità di eseguire sequenze di istruzioni che usano espressioni per specificare i calcoli da effettuare, e che usano l'assegnamento per memorizzare i risultati di tali calcoli; tra le istruzioni ne esistono alcune che hanno il compito di determinare la sequenza con cui le istruzioni di calcolo devono essere eseguite.

- Capacità di dividere il codice in sottoprogrammi che sono unità di codice parametrizzate e riutilizzabili in contesti diversi; le modalità di uso dei sottoprogrammi sono determinate dal meccanismo di chiamata e di ritorno e dai meccanismi per l'associazione tra i parametri effettivi e i parametri formali; tali modalità sono illustrate graficamente e in modo sintetico nella figura 10.3.

Prima di descrivere con maggiore dettaglio le funzioni e il modo di operate dei programmi che effettuano la traduzione in linguaggio macchina, è però opportuno illustrare come vengono usati i meccanismi della macchina reale per emulare i meccanismi dell'esecutore astratto.

Come è noto un'istruzione in linguaggio macchina consiste in una stringa di bit che specifica la funzione dell'istruzione (codice operativo) e quali sono i suoi operandi. Gli operandi di una istruzione possono essere altre istruzioni (nel caso di istruzioni di salto) o dati. A parte i casi in cui gli operandi sono indicati in modo esplicito nell'istruzione (per esempio quando si tratta di costanti), l'istruzione contiene solo un riferimento agli operandi, e poiché istruzioni e dati sono memorizzati nella stessa memoria fisica, in entrambi i casi tale riferimento consiste in un indirizzo di memoria.

La situazione è illustrata dalla figura 10.4 dove sono evidenziate un'istruzione che accede a una variabile globale attraverso il suo indirizzo e un'istruzione che effettua la chiamata a un sottoprogramma attraverso un salto al suo *entry point* (cioè all'indirizzo della sua prima istruzione).

La figura evidenzia che, durante l'esecuzione del programma, il codice macchina viene memorizzato in una porzione contigua della memoria (detta *area codice*) e che ogni istruzione è associata ad un indirizzo assoluto. La figura mostra che anche le variabili globali occupano una porzione contigua di memoria (detta *area dati statici*) e che ognuna di esse è pure associata ad un indirizzo assoluto. Si noti che nella figura, in conformità a quanto accade nella maggior parte dei casi reali, l'area dati statici è separata dall'area codice. Analogamente alle variabili globali, anche le variabili locali (non mostrate in figura) sono associate a indirizzi assoluti di memoria. Poiché però tali variabili sono contenute nei record di attivazione, l'associazione non è fissa, ma viene stabilita ogni volta che il record di attivazione viene creato, e viene meno quando il record di attivazione viene distrutto.

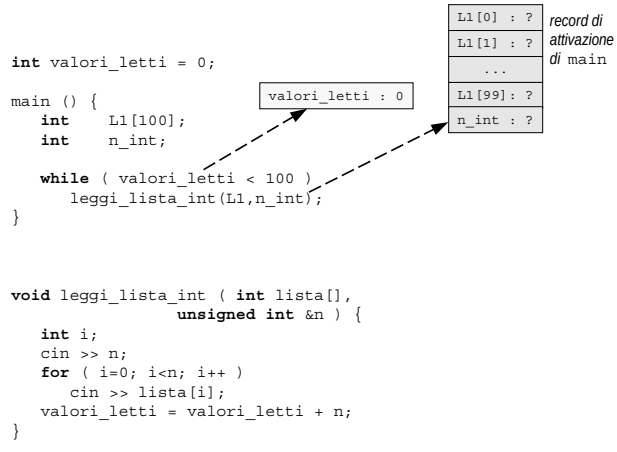
L'associazione a una istruzione o a un dato di un indirizzo di memoria evidenziata dalla figura 10.4 viene indicata con il termine *allocazione*.

10.3.1 Il problema del binding

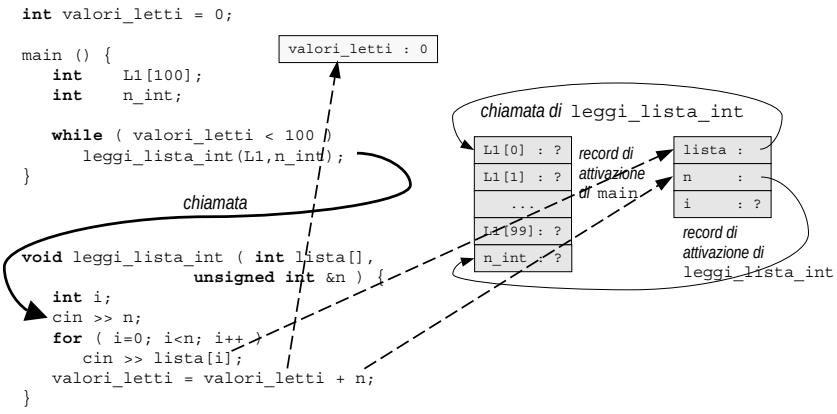
Da quanto detto risulta evidente che l'esecuzione di una qualunque istruzione richiede la conoscenza dell'indirizzo associato in quel momento ai suoi operandi (esclusi quelli direttamente specificati nell'istruzione stessa). Il procedimento per cui si determina l'indirizzo di un operando (istruzione o dato) viene detto *binding*. Esso può essere effettuato prima di iniziare l'esecuzione di un programma per tutti gli operandi, o essere riniviato al momento in cui l'esecuzione di una determinata istruzione lo richieda. Nel primo caso si parla di binding *statico*, nel secondo di binding *dinamico*.

È del tutto evidente che il binding può essere effettuato solo dopo che è nota l'allocazione dell'operando. Pertanto la scelta di effettuare staticamente il binding implica che l'allocazione di tutte le istruzioni e di tutte le variabili debba essere decisa prima di iniziare l'esecuzione del programma.

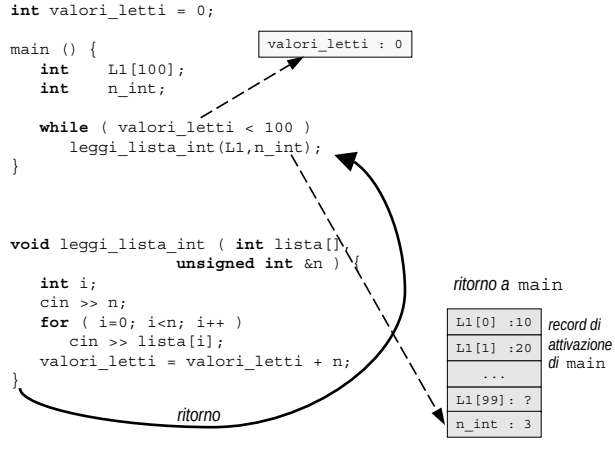
Sebbene questo modo di procedere semplifichi l'esecuzione del programma perché le istruzioni possono fare riferimento agli indirizzi assoluti degli operandi, essa implica che le istruzioni e i dati di un programma devono essere



(a)



(b)



(c)

Figura 10.3: Meccanismi di esecuzione dei sottoprogrammi: (a) durante l'esecuzione del `main`; (b) chiamata ed esecuzione del sottoprogramma `leggi_lista_int`; (c) ritorno dal sottoprogramma e ripresa dell'esecuzione del `main`.

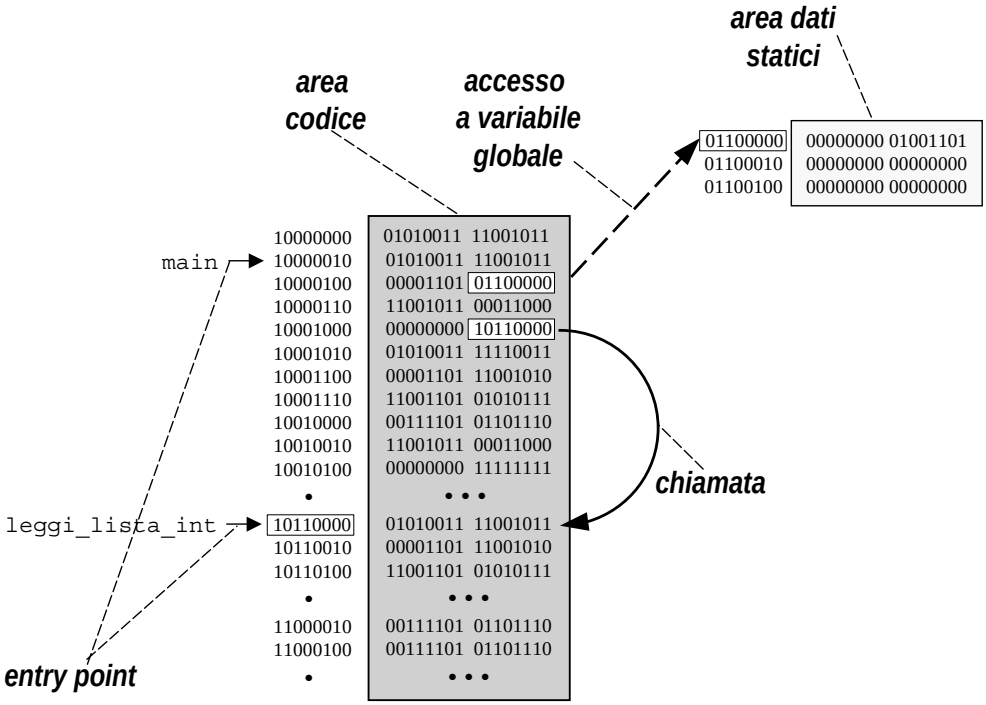


Figura 10.4: Esecuzione su una macchina reale.

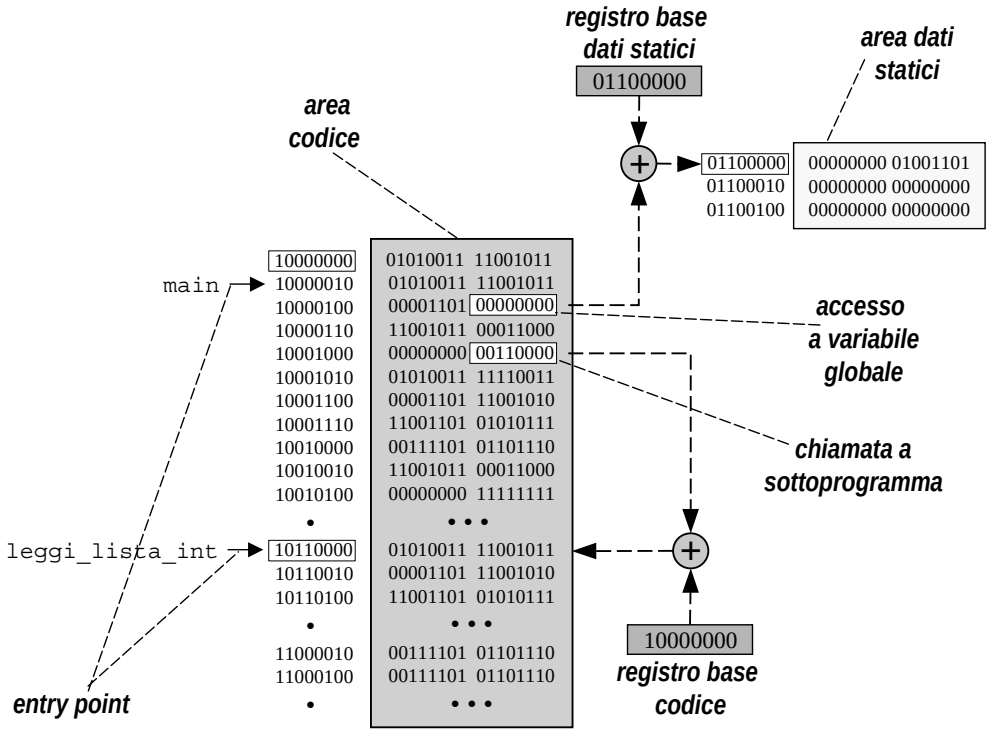


Figura 10.5: Esecuzione di programmi con indirizzi relativi e impiego del binding dinamico.

allocati in celle di memoria prefissate. In caso contrario, infatti, le istruzioni di salto e gli accessi ai dati non potrebbero essere eseguiti correttamente. Questo introduce un'eccessiva rigidità nell'uso della macchina. Tra gli inconvenienti del binding statico si possono infatti segnalare i seguenti:

- non sarebbe di fatto possibile eseguire più programmi contemporaneamente, a meno che l'insieme dei programmi da eseguire non sia deciso a priori in modo da allocarli in zone disgiunte della memoria;
- per quanto osservato al punto precedente, lo spazio assegnato al sistema operativo dovrebbe essere delimitato a priori in modo statico;
- i record di attivazione dei sottoprogrammi dovrebbero essere pure allocati staticamente rendendo impossibile l'esecuzione ricorsiva dei sottoprogrammi stessi.

Per questi ed altri motivi, generalmente l'esecuzione dei programmi sulle macchine reali impiega tecniche dinamiche di binding. Tali tecniche sono tutte basate sull'impiego di indirizzi relativi nelle istruzioni e nell'effettuazione del binding man mano che queste vengono eseguite con l'aiuto di dispositivi hardware dedicati a questo scopo.

Nella figura 10.5 viene mostrata l'esecuzione di un programma le cui istruzioni non contengono l'indirizzo assoluto degli operandi, ma solo la loro posizione relativa rispetto alla prima cella occupata rispettivamente dall'area codice e dall'area dati statici.

Nella CPU sono presenti due registri speciali detti *registro base codice* e *registro base dati statici* che contengono, rispettivamente, l'indirizzo iniziale dell'area codice e l'indirizzo iniziale dell'area dati statici. Quando viene eseguita un'istruzione di salto, la posizione relativa dell'istruzione a cui si deve saltare contenuta nell'istruzione viene sommata al contenuto del registro base codice ottenendo l'indirizzo effettivo dell'istruzione a cui saltare. Analogamente,

quando viene eseguita un'istruzione che deve accedere ad un dato, la posizione relativa del dato a cui si deve accedere contenuta nell'istruzione viene sommata al contenuto del registro base dati statici ottenendo l'indirizzo effettivo del dato a cui accedere.

L'impiego di binding dinamico richiede pertanto che nella fase di traduzione del programma venga effettuato sugli operandi delle istruzioni un binding statico assumendo che dati e istruzioni vengano allocati a partire dall'indirizzo 0 (*indirizzi relativi*). Quando, al momento dell'esecuzione del programma, viene decisa la sua allocazione, il vero indirizzo iniziale dell'area codice e dell'area dati statici viene memorizzato nei rispettivi *registri base* e, con l'aiuto di un circuito sommatore dedicato, il contenuto di tali registri viene usato durante l'esecuzione delle istruzioni per effettuare il binding degli operandi.

10.3.2 Rilocazione del codice

Con il termine *rilocazione* del codice si intende la traslazione del codice da una zona di memoria a un'altra. Come si è accennato nel paragrafo precedente, se il codice contiene indirizzi assoluti, la rilocazione richiede di ripetere il binding statico per tutti gli operandi e viene pertanto detta *rilocazione statica*.

Quando invece si fa uso nel codice di indirizzi relativi, mediante l'uso dei registri base descritta in precedenza, un programma può essere allocato in una zona qualsiasi della memoria, effettuando la rilocazione inizializzando opportunamente il contenuto dei registri base. Tale forma di rilocazione viene detta *rilocazione dinamica* e può essere effettuata perfino a metà dell'esecuzione del programma, interrompendola, copiando codice e dati in altra zona di memoria e aggiornando di conseguenza i registri base.

Si noti che però la rilocazione dinamica richiede, oltre ad un procedimento di traduzione più complesso, la predisposizione di hardware dedicato (i registri base e il sommatore). Se la CPU non prevede tale possibilità non è quindi possibile fare ricorso alla rilocazione dinamica e gli inconvenienti di gestione accennati nel paragrafo precedente non possono essere evitati.

10.4 Strumenti di sviluppo

10.4.1 Compilatore

Il compito principale del compilatore è quello di tradurre il programma scritto dal programmatore in linguaggio ad alto livello (file *sorgente* o, più brevemente, *sorgente*) in un equivalente programma in linguaggio macchina. Il prodotto del compilatore è un file che contiene il prodotto della traduzione e che viene detto *file oggetto* o, più brevemente, *oggetto*.

Nell'effettuare la traduzione il compilatore assegna a istruzioni e dati degli indirizzi relativi (cioè a partire dall'indirizzo 0) e utilizza tali indirizzi relativi nel generare le istruzioni in linguaggio macchina. In altre parole viene effettuato un binding statico sugli operandi con riferimento ad una allocazione fittizia a partire dall'indirizzo 0.

Poiché ogni file viene compilato separatamente, tale binding statico relativo viene effettuato solo sugli operandi *locali*, cioè sulle istruzioni e sui dati definiti nell'unità di compilazione in esame. Nel caso siano presenti operandi per i quali non è disponibile la definizione (come per esempio sottoprogrammi o variabili globali definiti in un'altra unità di compilazione), il compilatore non è in grado di ricavare l'allocazione, neppure relativa, di tali operandi, e pertanto non ne può effettuare il binding. Il compilatore lascia pertanto indefiniti i riferimenti a questi operandi nelle istruzioni. In altre parole, le istruzioni contenenti riferimenti a istruzioni o dati non locali sono tradotte solo parzialmente.

I riferimenti a istruzioni o dati non locali sono detti *riferimenti esterni* e producono l'aggiunta di alcune informazioni ausiliarie al file oggetto. Tali aggiunte consistono in una lista di riferimenti esterni, ciascuno consistente nell'identificatore associato al riferimento e nell'indirizzo relativo dell'istruzione incompleta che lo contiene. In figura 10.6 è mostrato schematicamente il contenuto del file oggetto corrispondente all'unità di compilazione che contiene la definizione del sottoprogramma `leggi_lista_int` e fa riferimento ad una variabile globale `valori_letti` definita in altra unità. Il codice sorgente dell'unità e riportato nel paragrafo 10.2.4.

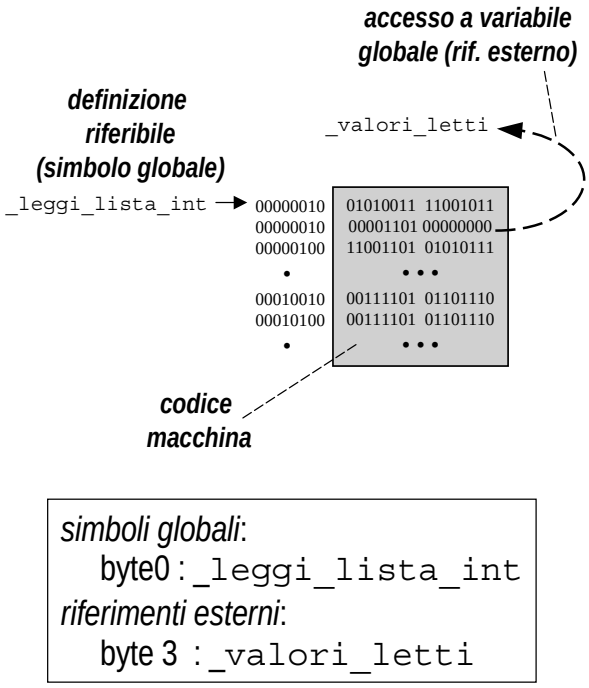


Figura 10.6: Formato del file oggetto prodotto dal compilatore.

Oltre al codice macchina, per il quale è evidenziato l'entry point del sottoprogramma `leggi_lista_int`, il file oggetto contiene due liste evidenziate graficamente nel riquadro. La seconda è la lista dei riferimenti esterni (uno solo in questo caso) menzionata in precedenza. La prima, detta lista dei *simboli globali* contiene l'identificatore e l'indirizzo relativo dei sottoprogrammi e delle variabili locali definiti nel file sorgente e ai quali quindi il compilatore ha assegnato un'allocazione (relativa). In questa lista, come suggerisce il nome, sono contenute le informazioni relative alle entità che potrebbero essere riferite da altre unità di compilazione, con lo scopo di agevolare la fase di collegamento descritta nel prossimo paragrafo.

Nell'esempio riportato nella figura la lista dei simboli globali contiene un solo simbolo che corrisponde all'entry point del sottoprogramma `leggi_lista_int`, definito nell'unità di compilazione.

Concludiamo il paragrafo notando che nella figura gli identificatori associati ai simboli globali e ai riferimenti esterni coincidono con quelli indicati nel testo sorgente dell'unità di compilazione preceduti dal carattere '_'. Anche se non si tratta di una regola generale, è infatti frequente che il compilatore e il collegatore modifichino leggermente gli identificatori assegnati dal programmatore a sottoprogrammi e variabili globali, premettendo ad esempio il carattere '_' o il carattere '@'.

10.4.2 Il collegatore

Nella figura 10.7 è mostrato schematicamente il contenuto dei due file oggetto corrispondenti alle unità di compilazione mostrate nella figura 10.2. Si noti come nel primo file la lista dei simboli globali, oltre all'identificatore associato alla variabile globale, comprende anche l'identificatore del programma principale. Il `main` infatti viene tradotto dal compilatore come un normale sottoprogramma e, come vedremo, l'informazione del file oggetto in cui è contenuto è necessaria al collegatore per generare correttamente il programma eseguibile. Per quanto riguarda i riferimenti esterni, nel primo file oggetto la lista include il riferimento a `leggi_lista_int` (chiamata al sottoprogramma definito nella

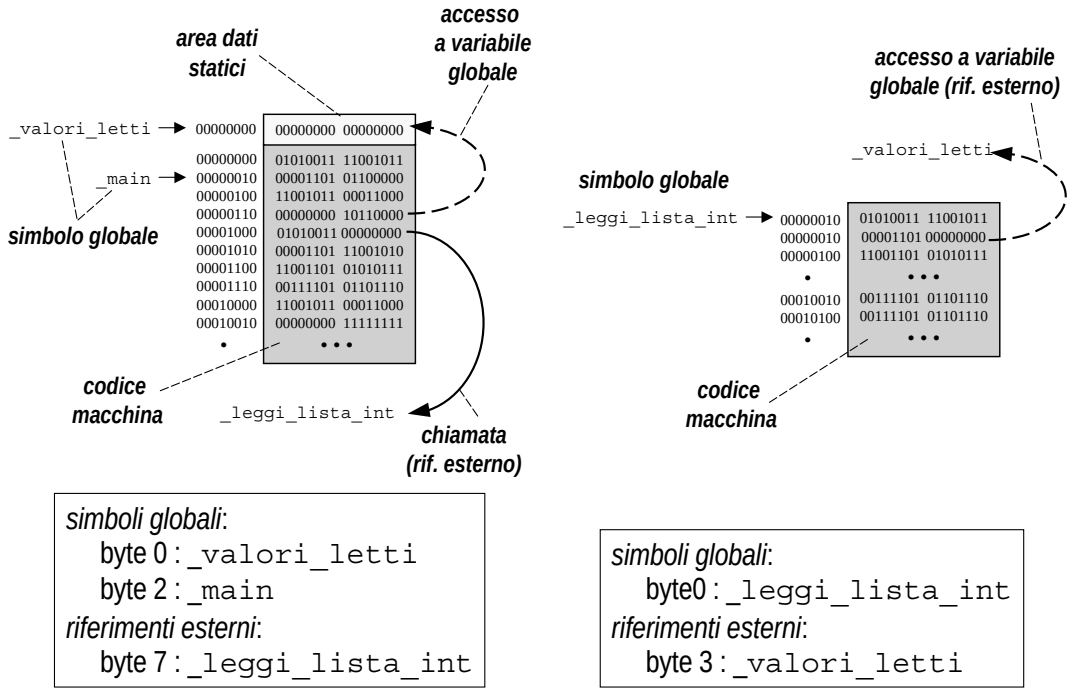


Figura 10.7: File oggetto corrispondenti alle due unità di compilazione di figura 10.2.

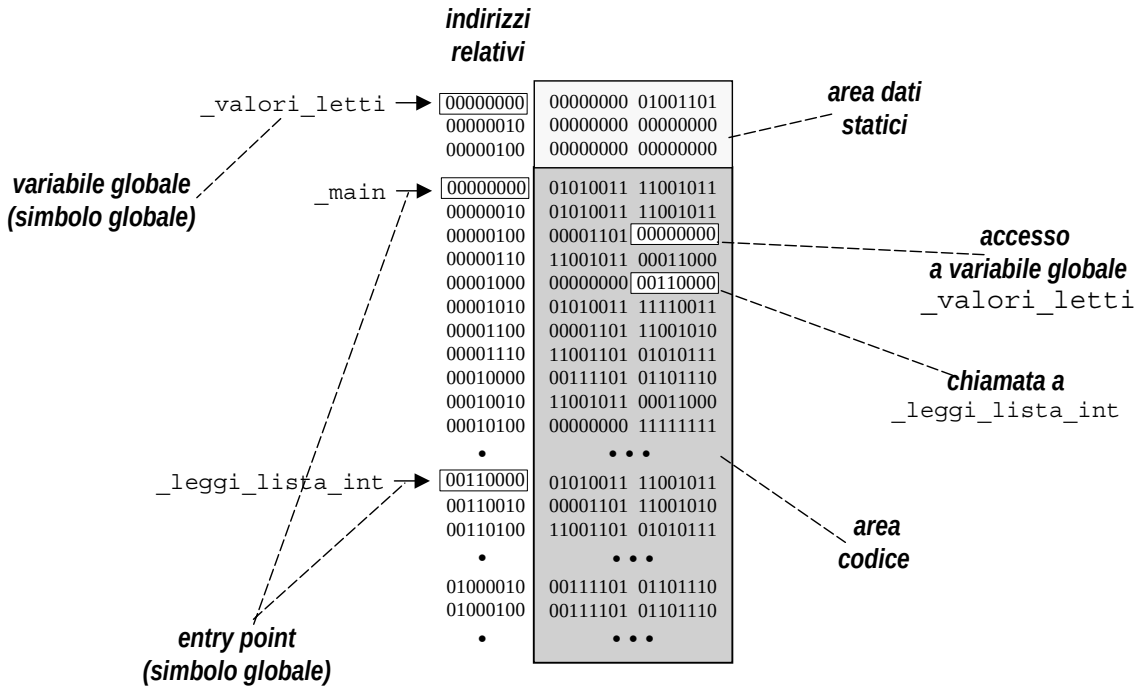


Figura 10.8: Prodotto finale della fase di collegamento.

seconda unità di compilazione), mentre nel secondo file oggetto la lista include il riferimento a `valori_letti` (accesso alla variabile globale definita nella prima unità di compilazione).

In definitiva, l'esempio mostra come la lista dei simboli globali sia in qualche modo complementare a quella dei riferimenti esterni, e in effetti il collegatore usa le liste dei simboli globali di tutti i file oggetto per *risolvere* i riferimenti esterni di ciascun file oggetto.

Più specificamente, il collegatore effettua i seguenti passi:

- Identifica le unità di compilazione da collegare. Tale fase viene condotta sulla base di una lista di file oggetto (file con suffisso `.obj` o `.o`) e di librerie (file con suffisso `.lib` o `.a` di cui parleremo successivamente) che viene solitamente fornita dall'utente.
- Dispone i file oggetto uno dopo l'altro, separando in genere l'area codice dall'area dati statici. Dopo tale fase il contenuto dei file oggetto risulta unificato in due blocchi, uno di istruzioni e uno di variabili statiche (cioè di variabili che vengono create all'inizio dell'esecuzione del programma e che vengono distrutte solo quando il programma termina).
- Ripete il binding statico di tutti gli operandi (esclusi quelli lasciati indefiniti perché corrispondenti a riferimenti esterni) sulla base dei nuovi indirizzi relativi. Tale operazione è richiesta perché, dopo l'unificazione di istruzioni e dati statici, l'allocazione relativa di entrambi risulta modificata (si confrontino ad esempio gli indirizzi relativi degli entry point nelle figure 10.7 e 10.8). Si noti che, per ogni unità di compilazione, tale modifica consiste semplicemente in una traslazione degli indirizzi relativi generati dal compilatore di una quantità pari alla posizione di tale unità nei blocchi generati al passo precedente.
- Effettua il binding dei riferimenti esterni sulla base dei nuovi indirizzi relativi (si veda ad esempio la chiamata a `_leggi_lista_int` in figura 10.8). Come accennato in precedenza, in questa fase il collegatore scorre per ciascuna unità di compilazione la corrispondente lista dei riferimenti esterni e per ogni istruzione da completare effettua il binding degli operandi indefiniti sulla base delle informazioni contenute nelle liste di simboli globali di tutte le unità di compilazione.

Al termine del collegamento, viene prodotto un file contenente codice macchina e dati statici in cui non sono più presenti riferimenti esterni e tutte le istruzioni sono completamente tradotte. In figura 10.8 è mostrato il risultato del collegamento delle due unità di compilazione considerate precedentemente. Si noti che tutti gli operandi contengono esclusivamente indirizzi relativi e pertanto il programma può essere facilmente allocato in una qualsiasi zona della memoria, rilocandolo attraverso un'opportuna inizializzazione del registro base codice e del registro base dati statici, come descritto in precedenza.

10.4.3 Funzionalità del collegatore

Quanto detto finora sul funzionamento del collegatore ha validità generale e si applica a tutte le tipologie di collegamento. Tuttavia, il collegamento di più file oggetto può dare luogo a più prodotti finali che rispondono a esigenze diverse. In questo paragrafo descriveremo brevemente l'impiego del collegatore per produrre due tipi di file: il file eseguibile e la libreria statica.

Il file eseguibile

Con il termine *file eseguibile* si intende un file che contenga il codice e le informazioni necessari per l'esecuzione di un programma in un fissato ambiente operativo. Si noti che tale definizione non implica né che un file eseguibile debba contenere esclusivamente codice macchina, né che debba contenere tutto il codice macchina effettivamente eseguito durante una particolare esecuzione del programma. In particolare, il file eseguibile contiene sempre alcune informazioni necessarie all'ambiente operativo per decidere come allocare e come caricare il programma in memoria.

Il file eseguibile è solitamente il prodotto di default del collegatore. Quando il collegatore viene invocato senza particolari opzioni esso genera pertanto un file eseguibile, caratterizzato dal suffisso *.exe* nei sistemi windows, e senza particolari suffissi nei sistemi UNIX.

Per produrre un file eseguibile occorre che la lista dei file oggetto da collegare sia completa. Essa cioè deve includere tutti i file oggetto che contengono la definizione di simboli (sottoprogrammi e variabili globali) usati nel programma. Ulteriore condizione per produrre un file eseguibile è che tra i simboli definiti dai file oggetto collegati esista il simbolo globale `_main`. Tale simbolo corrisponde all'entry point del programma e viene usato dal collegatore per informare l'ambiente operativo sul punto da cui iniziare l'esecuzione.

Come conseguenza di quanto detto, le principali segnalazioni di errore che può generare il collegatore quando deve produrre un file eseguibile sono due: la prima riguarda il caso in cui nessuno dei file oggetto da collegare contiene la definizione di un simbolo globale riferito da un altro file oggetto, la seconda riguarda il caso in cui nessun file oggetto contiene la definizione del simbolo `_main`.

Ad esempio, se si tenta di produrre un file eseguibile solo dal primo file oggetto di figura 10.7, si ottiene un messaggio di errore del tipo:

> linker error — symbol `_leggi_lista_int` undefined

che segnala che il simbolo `_leggi_lista_int` è rimasto indefinito.

Viceversa, se si tenta di produrre un file eseguibile solo dal secondo file oggetto di figura 10.7, oltre al messaggio di errore del tipo:

> linker error — symbol `_valori_letti` undefined

che segnala che il simbolo `_valori_letti` è rimasto indefinito, si ottiene un secondo messaggio di errore del tipo:

> linker error — symbol `_main` undefined

che segnala che il simbolo `_main`, richiesto per generare le informazioni di caricamento per l'ambiente operativo, non è presente nei file oggetto indicati nella lista.

La libreria statica

Con il termine *libreria statica* si intende un file che contenga un insieme di file oggetto insieme ad alcune informazioni ausiliarie (come il numero dei file oggetto, l'ordine con cui sono disposti nella libreria, ecc.).

La libreria statica non è solitamente il prodotto di default del collegatore, ma richiede la presenza di un'opportuna opzione. Quando il collegatore viene invocato specificando l'opzione che abilita la generazione di una libreria statica, esso genera una libreria statica, caratterizzata dal suffisso *.lib* nei sistemi windows e dal suffisso *.a* nei sistemi UNIX.

Anche nel caso di generazione di una libreria statica occorre che la lista dei file oggetto da collegare sia completa. Essa cioè deve includere tutti i file oggetto che contengono la definizione di simboli (sottoprogrammi e variabili globali) usati negli altri file oggetto che compongono la libreria statica. Diversamente dal programma eseguibile non è richiesto (ma non è neppure proibito) che sia definito il simbolo `_main`. Nel caso della libreria statica infatti non viene generata alcuna informazione di caricamento per l'ambiente operativo, dal momento che essa non rappresenta in alcun modo una porzione di codice eseguibile autonomamente.

Una volta generata, una libreria statica contiene tutti i file oggetto presenti nella lista inizialmente fornita al collegatore e può essere usata dal collegatore per generare file eseguibili o altre librerie statiche. Il vantaggio di raggruppare un insieme di file oggetto in una libreria statica è duplice. Da un lato, per usare in collegamenti successivi i file oggetto in essa contenuta è sufficiente specificare solo il nome della libreria, lasciando al collegatore il compito di estrarre automaticamente i file oggetto necessari. Dall'altro lato, quando i file oggetto da collegare sono contenuti in una libreria statica, il collegatore estrae i file in modo selettivo, partendo dai riferimenti esterni non risolti e aggiungendo i file oggetto in modo incrementale fino a che tutti i riferimenti esterni non siano stati risolti. I file oggetto non interessati dal procedimento non vengono estratti dalla libreria.

Ad esempio, si supponga che la libreria `esempio.lib` contenga i seguenti file oggetto:

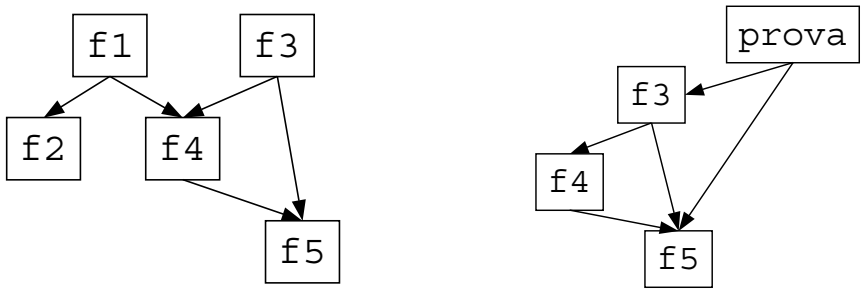


Figura 10.9: Relazioni tra i file oggetto contenuti in una libreria e componenti di un programma eseguibile che ne utilizza alcuni.

File oggetto	Riferimenti esterni
`f1.obj`	contiene riferimenti esterni a simboli definiti in `f2.obj` e in `f4.obj`
`f2.obj`	non contiene riferimenti esterni
`f3.obj`	contiene riferimenti esterni a simboli definiti in `f4.obj` e in `f5.obj`
`f4.obj`	contiene riferimenti esterni a simboli definiti in `f5.obj`
`f5.obj`	non contiene riferimenti esterni

La struttura interna della libreria è descritta graficamente nella parte sinistra della figura 10.9, dove le frecce indicano la presenza in un file oggetto di riferimenti esterni a simboli definiti in un altro file oggetto.

Se il file oggetto `prova.obj` contiene la definizione del simbolo `_main` insieme e riferimenti esterni a simboli definiti in `f3.obj` e in `f5.obj`, invocando il collegatore su una lista che comprende il file oggetto `prova.obj` e la libreria statica `esempio.lib`, viene prodotto il file eseguibile `prova.exe` che comprende il codice macchina contenuto in `prova.obj`, `f3.obj`, `f4.obj`, e `f5.obj` (parte destra della figura 10.9). Come si può facilmente notare, il file eseguibile finale non comprende il codice incluso in `f1.obj` e in `f2.obj` che in effetti non è utilizzato dal programma. L'effetto finale è lo stesso che si avrebbe avuto collegando esplicitamente tutti i file oggetto richiesti, col vantaggio che l'utente oltre a non doverli elencare singolarmente, non è tenuto neppure a conoscere la loro identità. L'unica informazione richiesta è il nome della libreria che contiene i file oggetto potenzialmente necessari.

10.4.4 Compilatori e interpreti

Nei paragrafi precedenti, dopo aver descritto le modalità con cui i programmi sono eseguiti su una macchina reale, è stato mostrato come la macchina astratta definita da un linguaggio di programmazione ad alto livello viene realizzata attraverso l'impiego combinato del compilatore e del collegatore.

Il procedimento è descritto graficamente dalla figura 10.10 dove, per semplicità, il codice sorgente è rappresentato da un unico file sorgente, e la coppia compilatore/collegatore è rappresentata come un unico programma. La generalizzazione al caso di più unità di compilazione è immediata.

La figura a sinistra mostra che, una volta redatto il file sorgente, esso viene usato come dato di ingresso per eseguire l'algoritmo descritto dal codice del compilatore/collegatore sulla macchina reale (esecutore). Il prodotto di questa esecuzione è il programma eseguibile.

A questo punto, l'algoritmo descritto dal file eseguibile viene eseguito sulla macchina reale, usando come ingresso i dati del problema da risolvere, e ottenendo i dati di uscita richiesti (figura a destra).

Questo modo di procedere presenta alcuni vantaggi e svantaggi. Gli svantaggi sono che:

- occorre attendere che sia completata la traduzione dell'intero programma prima di poterne iniziare l'esecuzione;

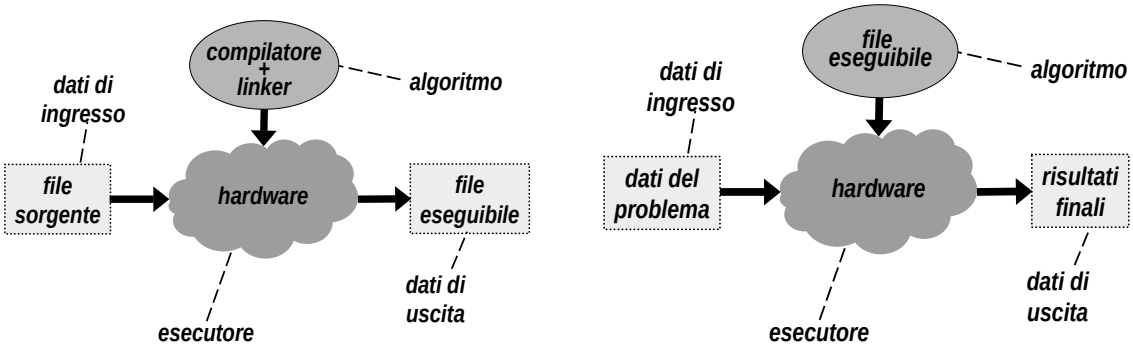


Figura 10.10: Traduzione ed esecuzione di programmi compilati.

- ogni modifica al programma sorgente richiede una nuova compilazione dell'unità interessata e la ripetizione della fase di collegamento;
- per eseguire il programma su una macchina reale diversa occorre ripetere sia la compilazione che il collegamento;
- la necessità di far corrispondere in qualche modo i meccanismi linguistici del linguaggio ad alto livello ai meccanismi elementari messi a disposizione dalla macchina reale costringe a introdurre anche nei linguaggi ad alto livello restrizioni che ne limitano la facilità d'uso.

A fronte di questi svantaggi, il principale vantaggio è rappresentato dalla possibilità di utilizzare metodi molto sofisticati per effettuare la traduzione, in modo da ottenere un programma eseguibile molto efficiente in termini di velocità di esecuzione e di spazio (quantità di memoria) impegnato. Inoltre, l'approccio compilato richiede di effettuare l'analisi del testo sorgente del programma una sola volta prima dell'esecuzione, contribuendo così ulteriormente a ridurre i tempi di esecuzione. Poiché in molti casi i programmi, una volta compilati devono essere eseguiti molte volte sullo stesso esecutore, tali vantaggi sono stati ritenuti per molto tempo sufficienti a giustificare l'impiego di linguaggi *compilati* nella maggior parte delle applicazioni.

È possibile tuttavia seguire un approccio diverso per realizzare la macchina astratta corrispondente al linguaggio ad alto livello sulla macchina reale. Invece di tradurre il programma sorgente in un programma eseguibile, viene utilizzato un programma, detto *interprete*, in grado di ricostruire il significato del codice sorgente e di eseguire direttamente le operazioni della macchina astratta che esso descrive.

L'impiego degli interpreti è descritto graficamente nella figura 10.11. Nella figura a sinistra, il programma interprete rappresenta l'algoritmo eseguito dalla macchina reale (esecutore). Sia il file sorgente che i dati di ingresso del problema che esso risolve rappresentano dati di ingresso dell'interprete. L'esecuzione di quest'ultimo produce in uscita i dati richiesti sulla base dell'algoritmo descritto dal file sorgente. Si noti che non viene generata alcuna traduzione in linguaggio macchina del file sorgente.

La figura a sinistra mostra anche come sia possibile immaginare di non distinguere tra interprete e macchina reale e considerare come esecutore il loro effetto combinato. L'effetto finale è allora quello descritto dalla figura a destra, dove il programma sorgente (e non una sua traduzione in linguaggio macchina) è l'algoritmo eseguito da un esecutore (detto *macchina virtuale* e coincidente con la macchina astratta del linguaggio ad alto livello) per trasformare i dati di ingresso nei dati di uscita richiesti.

Come nel caso dei programmi compilati, anche l'impiego di programmi interpretati ha i sui vantaggi e svantaggi. Sotto questo punto di vista la situazione è sostanzialmente complementare rispetto al caso dei programmi compilati. I principali vantaggi degli interpreti sono:

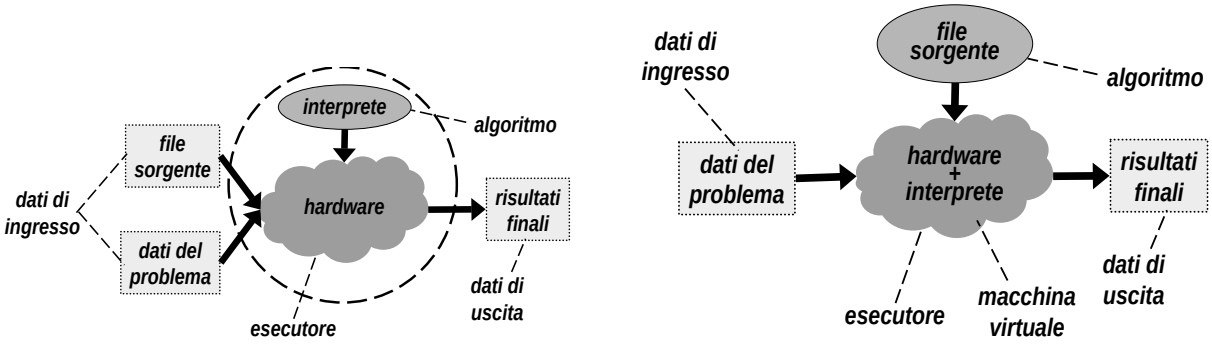


Figura 10.11: Esecuzione di programmi interpretati.

- l'inizio immediato dell'esecuzione senza bisogno di effettuare prima la traduzione dell'intero programma;
- la completa portabilità dei programmi su qualunque ambiente operativo, purché sia disponibile il programma l'interprete in tale ambiente;
- la possibilità di introdurre facilmente nel linguaggio qualsiasi costrutto, dal momento che la macchina astratta che lo deve comprendere ed eseguire è realizzata in modo diretto attraverso un programma e non traducendo il costrutto in un altro linguaggio.

Di contro, il principale svantaggio è la minore efficienza di esecuzione, legata sia al fatto che i meccanismi della macchina astratta sono realizzati dall'interprete in modo generale e quindi, almeno in parte, ridondante, sia perché l'analisi del testo sorgente del programma, necessaria per la sua comprensione, deve essere ripetuta ad ogni esecuzione.

La minore efficienza dell'approccio basato su interpreti ha determinato per molto tempo la tendenza a limitare il loro impiego. In particolare, gli interpreti sono tradizionalmente preferiti ai compilatori nelle seguenti situazioni:

- durante lo sviluppo del programma, quando le frequenti modifiche e l'impiego di casi di test di dimensioni relativamente modeste implicano che, da un lato l'eliminazione della fase di traduzione ad ogni modifica si traduca in un risparmi complessivo di tempo, e dall'altro che le inefficienze di esecuzione siano poco rilevanti;
- per l'implementazione di macchine astratte per linguaggi di livello molto elevato (per esempio i linguaggi funzionali e i linguaggi logici) i cui costrutti sono molto difficili da tradurre in linguaggio macchina;
- in applicazioni particolari che non richiedono particolare efficienza di esecuzione; a tale classe appartengono i cosiddetti linguaggi di comandi che consentono all'utente di interagire con l'ambiente operativo.

Questa situazione sta però in parte cambiando per diverse ragioni. Da un lato le tecniche di realizzazione degli interpreti si sono molto evolute riducendo le inefficienze di esecuzione. Ad esempio, nella maggior parte dei casi, oggi gli interpreti non operano direttamente sul linguaggio sorgente, ma su una sua traduzione in un linguaggio intermedio molto semplice da analizzare. In questo modo, che è in realtà un approccio intermedio tra quello compilato e quello interpretato, l'analisi del testo sorgente viene effettuata una sola volta e comunque non pesa sulla normale esecuzione del programma. Dall'altro lato, l'impressionante aumento di prestazioni delle macchine reali e la riduzione di costi delle memorie, rende sempre meno necessario ricorrere a ottimizzazioni spinte del codice, almeno nella maggior parte delle applicazioni di interesse comune. Tale fenomeno, assieme alla tendenza a usare linguaggi sempre più evoluti e potenti che mal si prestano ad un'approccio completamente basato sull'uso di traduttori ha determinato un'ulteriore spinta all'impiego degli interpreti.