Capitolo 1

Introduzione alla programmazione

> "L'arte di programmare è l'arte di organizzare la complessità ...", "...dobbiamo organizzare le computazioni in modo tale che le nostre limitate capacità siano sufficienti a garantire che gli effetti delle computazioni stesse siano quelli voluti."([5]).

1.1 Dal problema all'algoritmo

Il punto di partenza è un *problema*. I problemi possono essere di vario tipo: semplici operazioni aritmetiche (come moltiplicazione di due numeri interi, minimo comune multiplo, ecc.), operazioni aritmetiche più complesse (come estrazione della radice quadrata, calcolo delle radici di una equazione di secondo grado, ecc.), ma anche problemi in altri campi, non necessariamente matematici, come la ricerca di un nome in un elenco telefonico, la ricerca di un tragitto ferroviario tra due località, il prelievo di soldi con un bancomat, la preparazione di un piatto di cucina, la gestione di un magazzino (arrivo di prodotti già presenti / non presenti, calcolo giacenze, ...), e così via.[1]

Il nostro scopo è "risolvere" il problema che ci viene proposto, o più precisamente: trovare (se esiste) un *metodo risolutivo* per risolvere il problema dato, che possa essere capito ed applicato da un *esecutore* (umano, meccanico, ...).

Questo ci porta ad introdurre la seguente nozione di algoritmo.

[1] Per una trattazione più approfondita della nozione di problema nel contesto dello sviluppo di programmi si veda, ad esempio, il testo [3], Capitolo 15, in cui vengono introdotte, tra l'altro, la nozione di *specifica* di un problema, una classificazione dei problemi—di ricerca, di decisione, di ottimizzazione—, le nozioni di spazio di ricerca, di correttezza, ecc..

Algoritmo *Un algoritmo è una sequenza finita di* istruzioni *che specificano come certe* operazioni elementari *debbano susseguirsi nel tempo per risolvere una data* classe di problemi.

In questa definizione troviamo tre termini, in particolare, che richiedono qualche ulteriore precisazione: istruzioni, operazioni elementari e classe di problemi.

Istruzioni: indicano, in genere, richieste di azioni rivolte all'esecutore e che da questo devono essere capite ed eseguite: ad esempio, memorizza un dato valore, salta ad una certa istruzione, mostra un risultato all'esterno, ecc.

Operazioni elementari: si intendono le operazioni che si assume siano note all'esecutore, e che quindi quest'ultimo è in grado di eseguire; ad esempio, somma e sottrazione di numeri interi, and logico, ecc.

Classe di problemi: si intende una formulazione del problema indipendente dagli specifici dati su cui si opera; ad esempio, non la somma dei numeri 34 e 57, ma la somma di due numeri interi qualsiasi.

La Figura 1.1 mostra schematicamente la relazione intercorrente tra problema, algoritmo ed esecutore.

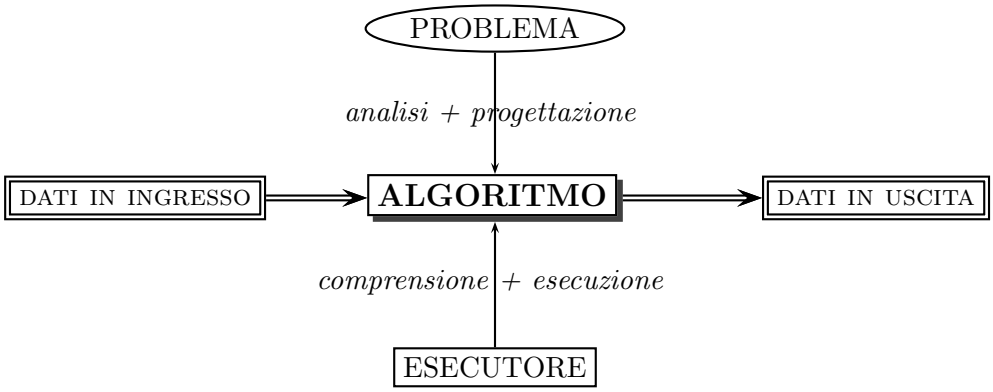


Figura 1.1: Dal problema all'algoritmo.

Alla formulazione dell'algoritmo si arriva attraverso una fase di *analisi* del problema (o fase di *concettualizzazione*) e quindi di *progettazione* del procedimento risolutivo, in cui si passa dalla specifica del problema alla definizione di un procedimento algoritmico in grado di risolvere il problema in accordo con le sue specifiche.

L'esecuzione dell'algoritmo comporta invece la capacità da parte dell'esecutore di capire le istruzioni e quindi di applicarle correttamente.

L'esecuzione dell'algoritmo richiede in generale che vengano forniti i ***dati*** specifici su cui operare (dati in ingresso o *input*) e produce, al termine della sua esecuzione, altri dati (*output*) che rappresentano i risultati attesi, ovvero le soluzioni del problema dato. In termini più astratti, un algoritmo può vedersi come un sistema per definire operativamente una *funzione* dall'insieme dei valori di input all'insieme dei valori di output. Da notare anche che i dati possono rappresentare non soltanto possibili valori di input o di output, ma anche possibili valori "intermedi", utilizzati durante l'esecuzione dell'algoritmo.

Algoritmi, più o meno semplici e più o meno formalizzati, si incontrano nella vita di tutti i giorni. Ad esempio:

- somma di due numeri in colonna—un'operazione elementare in questo caso può essere "la somma di due cifre", mentre un'istruzione può essere "passa alla colonna immediatamente a sinistra della attuale", oppure "se il risultato è maggiore di 10 allora ...", ecc.;
- una ricetta di cucina;
- le istruzioni di montaggio di un pezzo meccanico.

In tutti questi casi l'esecutore è, solitamente, umano.

Da notare che, in generale, si assume che l'esecutore esegua in modo *acritico* le istruzioni dell'algoritmo: non sa nulla del problema nella sua globalità; capisce soltanto le istruzioni e le operazioni elementari; è privo di "buon senso". Di conseguenza, la descrizione dell'algoritmo deve essere precisa, completa e non ambigua: le istruzioni devono poter essere interpretate in modo univoco dall'esecutore. Ad esempio, un'istruzione del tipo "scegli un ϵ piccolo a piacere" può essere ambigua: il significato di "piccolo a piacere" dipende in generale dal contesto in cui ci si trova. Analogamente, l'istruzione "aggiungi sale quanto basta" si basa chiaramente sul buon senso dell'esecutore e perciò non è adeguata per una formulazione algoritmica.

Oltre alle proprietà di eseguibilità e non ambiguità di un algoritmo, alcune altre proprietà fondamentali degli algoritmi sono:

- ***correttezza***: l'esecuzione dell'algoritmo porta realmente alla soluzione del problema dato;
- ***efficienza***: quanto "costa" l'esecuzione di un algoritmo in termini di risorse consumate (in particolare, se l'esecutore è un calcolatore, il tempo di CPU richiesto e lo spazio di memoria occupato);
- ***finitezza***: in pratica normalmente si richiede che l'algoritmo termini in un tempo finito, e cioè che la sequenza di istruzioni che caratterizzano l'algoritmo sia una sequenza finita.[2]

[2]Per una discussione sulla presenza o meno del requisito di finitezza nella definizione di algoritmo si veda ad esempio `http://en.wikipedia.org/wiki/Algorithm/\#Termination`.

Studio degli algoritmi (approfondimento) Dato un problema (o, meglio, una classe di problemi), esiste un algoritmo che permetta di risolverlo? A questa e ad altre domande simili risponde la *Teoria della computabilità*. Questa branca dell'informatica utilizza opportuni modelli di calcolo (macchine di Turing, automi a stati finiti, ecc.) per dimostrare l'esistenza o meno di algoritmi per una data classe di problemi. Ad esempio, stabilire in modo algoritmico se, dato un programma ed i suoi dati di input, l'esecuzione del programma con questi dati termina o no in un tempo finito non è calcolabile (cioè non ammette un algoritmo). Due testi classici su questi argomenti sono [7] e [10].

Se un problema è risolvibile in modo algoritmico, quanto "costa" eseguire l'algoritmo? Esistono algoritmi più efficienti per risolvere lo stesso problema? Quale è il limite inferiore di efficienza per questi algoritmi? A queste e ad altre domande simili risponde la *Teoria della complessità*. Questa branca dell'informatica si occupa di determinare le risorse richieste dall'esecuzione di un algoritmo per risolvere un dato problema. Le risorse più frequentemente considerate sono il tempo (quanti passi sono richiesti per completare l'algoritmo) e lo spazio (quanta memoria è richiesta per immagazzinare i dati usati dall'algoritmo). Per approfondire l'argomento si possono consultare, ad esempio, i testi [2] e [4]. Per una trattazione più snella si veda [3], Capitolo 8.

1.2 Descrizione di algoritmi

Una volta progettato, un algoritmo deve essere descritto utilizzando un opportuno linguaggio che risulti capibile ed eseguibile da parte dell'esecutore.

Linguaggio di descrizione algoritmi *Un* linguaggio di descrizione algoritmi *è un formalismo costituito da:*

- *un insieme di* istruzioni *primitive, dove il termine* primitive *(che ritroveremo spesso nel seguito) indica che si tratta di elementi propri, facenti parte, del linguaggio*
- *un insieme di* tipi di dato *primitivi, come ad esempio, numeri interi, numeri reali, caratteri, ecc.*
- *un insieme di* operazioni *primitive su tali dati, come ad esempio somma e sottrazione per i numeri interi e reali, ecc.*

Esempi di linguaggi di descrizione algoritmi (e relativi esecutori) sono:

- *Linguaggio naturale* (ad esempio, l'italiano, l'inglese), eventualmente limitato, pensato per un esecutore umano, in cui c'è grande libertà riguardo a istruzioni, dati e operazioni primitive previste. Ad esempio, se si vuol descrivere un algoritmo per il montaggio di un apparato da parte di un operatore umano, le operazioni primitive saranno del tipo "unisci due pezzi", "incolla", ecc. e le istruzioni saranno del tipo "ripeti una certa azione", "se hai questo pezzo, allora esegui la tale azione", ecc.
- *Linguaggi grafici*, come quello dei *diagrammi di flusso* o il più recente e generale *linguaggio UML* (Unified Modeling Language). Anche in questo caso si tratta di linguaggi pensati per un esecutore umano,

anche se molto più formalizzati e precisi dei precedenti. Tipiche istruzioni previste in questi linguaggi sono quelle di assegnamento, di test, di salto, ecc., mentre i tipi di dati e le operazioni primitive messe a disposizione sono quelle tipicamente previste dal dominio di applicazione che viene considerato (ad esempio, numeri interi e le solite operazioni aritmetiche). Vedremo nel sottocapitolo 1.3 una breve presentazione del formalismo dei diagrammi di flusso.

- *Linguaggi di programmazione*, come ad esempio C++, Pascal, Java, VisualBasic, ecc., che sono per loro natura adatti ad un esecutore automatico, specificatamente il calcolatore. Nel caso dei linguaggi di programmazione convenzionali le istruzioni primitive del linguaggio sono i tipici comandi (in inglese, *statement*) che realizzano, ad esempio, l'assegnamento e le strutture di controllo condizionale e iterativa, mentre i tipi di dato primitivi sono tipicamente numeri interi, numeri reali, caratteri e booleani, con le relative operazioni aritmetiche e logiche standard.

  Vedremo un esempio di linguaggio di programmazione, il C++, in dettaglio nei prossimi capitoli.

Macchina Astratta (approfondimento). La nozione di esecutore può essere meglio formalizzata tramite la nozione di Macchina Astratta (M.A.). Una ***Macchina Astratta*** per un linguaggio $\mathcal{L}$ (indicata con $\mathsf{MA}_{\mathcal{L}}$) è un 'entità in grado di "comprendere" ed eseguire algoritmi scritti nel linguaggio $\mathcal{L}$.

$\mathsf{MA}_{\mathcal{L}}$ rappresenta dunque un *esecutore* per algoritmi scritti in $\mathcal{L}$. Più precisamente, una macchina astratta $\mathsf{MA}_{\mathcal{L}}$ sarà costituita al suo interno da:

- una memoria in cui memorizzare l'algoritmo e i dati (ovvero "le variabili") su cui l'algoritmo opera;
- un interprete per $\mathcal{L}$ e cioè un "meccanismo" in grado di capire le istruzioni dell'algoritmo memorizzate nella memoria della M.A. (scritte in $\mathcal{L}$), e di applicarle ai dati anch'essi memorizzati nella memoria della M.A..

Il fatto che una M.A. sia *astratta*, comunque, significa che essa può essere definita, in generale, esclusivamente in base alle caratteristiche del suo linguaggio $\mathcal{L}$ (= insieme di istruzioni, dati, operazioni primitive), senza dover render nota la sua implementazione (ovvero, come la M.A. è fatta dentro, come funziona).

Ad esempio, nel caso dei linguaggi di programmazione, si potrà parlare semplicemente di M.A. del C, M.A. del Pascal, M.A. di Java, ecc., senza dover necessariamente specificare come questa sia implementata (in particolare, se basata su *interpretazione* o su *compilazione*).

1.3 I diagrammi di flusso (concetti di base)

I diagrammi di flusso (o "flow chart") sono un formalismo grafico di descrizione degli algoritmi. I diversi tipi di istruzioni che caratterizzano questo formalismo sono rappresentati tramite *blocchi* di varia forma, connessi da frecce (per questo si parla anche di *schemi a blocchi*). Si tratta di una descrizione di algoritmi rivolta principalmente ad un esecutore umano che, in

generale, ha il pregio di mettere ben in evidenza il "flusso del controllo" dell'algoritmo.

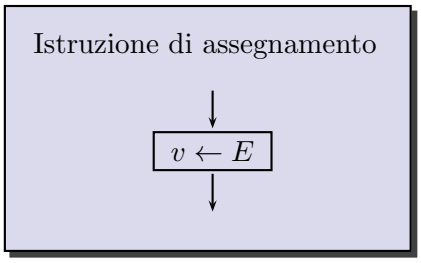
Flusso del controllo *Con flusso del controllo si intende il modo in cui si susseguono le diverse istruzioni dell'algoritmo. Solitamente vengono distinti alcuni schemi particolari di flusso di controllo, indicati genericamente come* **strutture di controllo**. *Le principali strutture di controllo sono:* sequenziale*, in cui le istruzioni si susseguono una dopo l'altra,* condizionale*, in cui si sceglie quali istruzioni eseguire in base ad una o più condizioni booleane,* iterativa*, in cui una o piu istruzioni vengono eseguite ripetutamente in base al valore di verita' di una certa condizione booleana.*

In questo capitolo descriveremo brevemente gli elementi base del formalismo dei diagrammi di flusso. In particolare, introdurremo tre tipi di blocchi—rettangolari, romboidali, ovali—e quindi le semplici regole con cui questi blocchi si possono comporre per ottenere diverse strutture di controllo e descrivere algoritmi completi. La descrizione di questo formalismo è volutamente non esauriente. Il suo scopo è essenzialmente quello di mostrare un formalismo di descrizione di algoritmi, alternativo ai linguaggi di programmazione, che sia semplice ed intuitivo e che ci permetta quindi di concentrare l'attenzione sulla nozione di algoritmo, piuttosto che sulle particolarità del formalismo stesso.

1.3.1 Istruzioni

Assegnamento

Un'istruzione fondamentale rappresentata tramite un blocco rettangolare è l'istruzione di *assegnamento*.



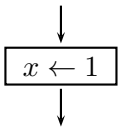
dove v è una variabile ed E è una espressione. Un'espressione, in generale, può essere una *costante* (ad es. 1), o una *variabile* (ad es. x, y), o un'*espressione composta*, cioè un'espressione ottenuta combinando costanti e variabili attraverso operatori (ad es., operatori aritmetici come in $x + 1$).

La semantica (ovvero, il significato) informale di tale istruzione è: *assegna alla variabile v il risultato della valutazione dell'espressione E.* Da

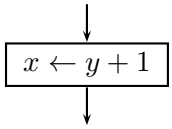
notare che prima viene valutata E e poi avviene l'assegnamento. In seguito all'assegnamento, il vecchio valore di v viene perso (ritorneremo sulla nozione di variabile e di assegnamento in un capitolo successivo).

Esempio 1.1 (Blocchi di assegnamento)

- *assegna alla variabile x il valore costante* 1*:*



- *assegna alla variabile x il risultato della valutazione dell'espressione $y + 1$:*

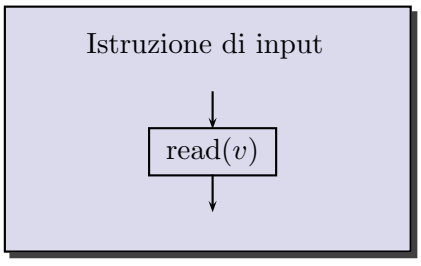


  Si noti che se l'espressione alla destra di $\leftarrow$ fosse $x+1$, l'assegnamento avrebbe semplicemente l'effetto di incrementare di 1 *il valore di x.*

I blocchi rettangolari possono essere utilizzati più in generale per rappresentare istruzioni che comportano una qualche modifica dello stato globale della computazione. Nel caso specifico dell'istruzione di assegnamento, questa modifica riguarda il valore della variabile che compare nella parte sinistra dell'assegnamento.

Qui di seguito utilizzeremo dei blocchi rettangolari anche per rappresentare due semplici *istruzioni di input/output*, il cui effetto sarà quello di modificare lo stato dell'ambiente esterno con cui l'algoritmo interagisce durante la sua esecuzione.[3]

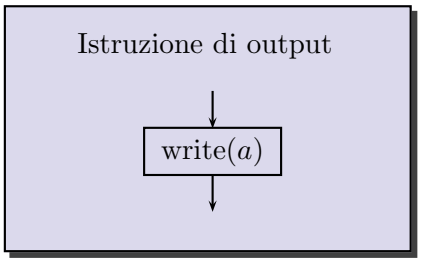
Istruzione di input



[3]Nell'uso comune queste istruzioni vengono spesso rappresentate da blocchi di forma speciale; noi preferiamo qui avvalerci dei blocchi rettangolari per non appesantire inutilmente la notazione utilizzata.

dove v è una variabile. Il significato di questa istruzione è: preleva il primo dato disponibile fornito dall'esterno ed assegnalo a v. Se il dato non è ancora disponibile, attendi.

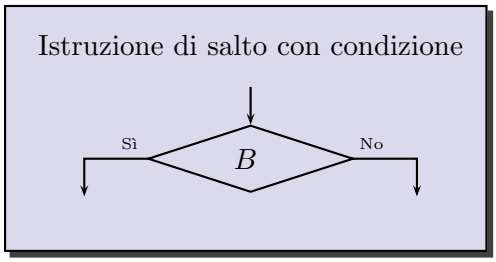
Istruzione di output



dove a è una variabile o una costante. Il significato di questa istruzione è: invia all'esterno il valore di a.

Istruzioni di test

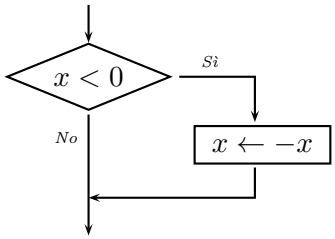
Per rappresentare istruzioni di test si utilizzano blocchi romboidali. La loro forma generale è la seguente:



dove B è un'espressione booleana, cioè un'espressione costruita tramite operatori di relazione (quali "=", ">", "<", ...) e connettivi logici (*and*, *or*, *not*), il cui risultato può essere soltanto di tipo booleano (*"vero"* o *"falso"*).

Il significato di tale blocco è: *valuta l'espressione B; se B ha valore "vero", continua dalla parte etichettata con* Sì*; altrimenti, cioè se B ha valore "falso", continua dalla parte etichettata con* No.

Esempio 1.2 (*Esecuzione condizionale*) *Se il valore della variabile x è minore di* 0 *assegna ad x il suo opposto e passa all'istruzione successiva; altrimenti, passa direttamente all'istruzione successiva:*

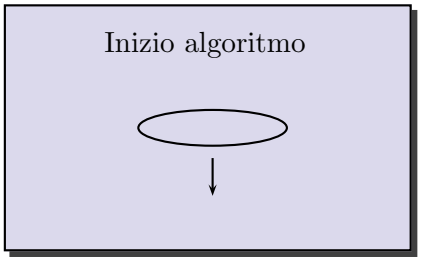


1.3.2 Esecuzione di un diagramma di flusso

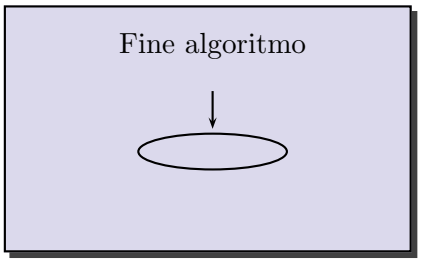
Oltre al modo di funzionare di ciascuna singola istruzione, per capire il significato (operazionale) di un intero diagramma di flusso bisogna anche specificare come il controllo passi da un'istruzione all'altra, e come si inizia e si finisce.

I diversi blocchi di un diagramma di flusso vengono tra loro combinati utilizzando in modo opportuno le frecce. Una freccia che va da un blocco A ad un blocco B indica che, terminata l'esecuzione del blocco A, si dovrà continuare con l'esecuzione del blocco B (nel caso di blocchi condizionali, la scelta di quale freccia considerare dipende dal valore della condizione B).

L'inizio e la fine di un algoritmo, invece, vengono indicati tramite *blocchi ovali*:



indica l'inizio dell'algoritmo e deve essere unico;



indica la fine dell'algoritmo e ne possono esistere più di uno all'interno dello stesso algoritmo.

Esecuzione di un diagramma di flusso *Il significato operazionale di un diagramma di flusso completo è il seguente: inizia dal blocco ovale iniziale; esegui i blocchi successivi, seguendo le frecce, in accordo con la semantica di ciascun blocco; termina quando raggiungi un blocco ovale finale.*

Vediamo un semplice esempio di algoritmo descritto con un diagramma di flusso.

Esempio 1.3 (Moltiplicazione per somme)

Problema*: dati due numeri interi maggiori o uguali a* 0, `a` *e* `b`*, determinarne il prodotto* `p`.

Procedimento risolutivo—*informale:*

$$\texttt{p} = \texttt{a} \cdot \texttt{b} = \texttt{a} + \texttt{a} + \cdots + \texttt{a}, \texttt{b} \text{ volte.}$$

Descrizione algoritmo*: con diagramma di flusso, assumendo:*

- *come* tipi di dato primitivi*: i numeri naturali*
- *come* operazioni primitive*: somma, uguaglianza, decremento unitario.*

Variabili*:* `a`*,* `b`*,* `p`*: interi maggiori o uguali a* 0*;* `a` *e* `b` *sono dati di input,* `p` *è un dato di output (il risultato fornito dall'algoritmo).*

Il diagramma di flusso che descrive l'algoritmo è mostrato in Figura 1.2.

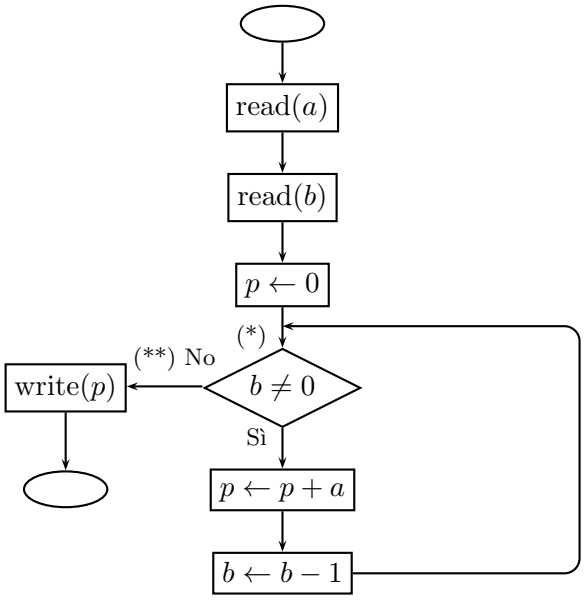


Figura 1.2: Diagramma di flusso dell'algoritmo di moltiplicazione per somme.

È possibile verificare "a mano" il funzionamento dell'algoritmo mostrato sopra, attribuendo dei valori numerici ai suoi dati di input. Ad esempio, se `a` vale 2 e `b` vale 3, i valori di `a`, `b` e `p` variano come mostrato nella tabella di Figura 1.3.

1.3.3 Strutture di controllo

La concatenazione dei blocchi tramite le frecce permette facilmente di realizzare diverse strutture di controllo.

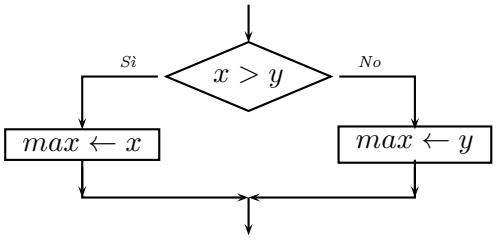
a	b	p
2	**3**	0
	2	2
	1	4
	0	**6**

Figura 1.3: Esecuzione di 2×3 con l'algoritmo di *moltiplicazione per somme*.

La *sequenza* è ottenuta banalmente concatenando l'uscita di un blocco con l'entrata del blocco successivo. Ad esempio, le prime tre istruzioni del diagramma di Figura 1.2 realizzano una struttura di controllo sequenziale.

Le strutture di controllo *condizionali* possono essere realizzate utilizzando blocchi romboidali. Abbiamo già visto una di tali strutture nell'esempio 1.2 (*esecuzione condizionale*). Un'altra struttura di controllo condizionale tipica è quella illustrata dal seguente esempio.

Esempio 1.4 (*Biforcazione*) *Se il valore della variabile x è maggiore del valore della variabile y assegna alla variabile max il valore di x; altrimenti assegna alla variabile max il valore di y:*



Le strutture di controllo *iterative* sono ottenute sfruttando in modo opportuno l'istruzione di "salto" rappresentata dalla frecce. Infatti, tramite le frecce è possibile richiedere l'esecuzione ripetuta di una o più istruzioni (ovvero, un *ciclo*). La ripetizione continuà finché una certa condizione (*condizione di uscita*) non cambia il suo valore booleano.

Ad esempio, nel diagramma di Figura 1.2 risulta chiara la presenza di una struttura di controllo iterativa. Il punto evidenziato con (*) è il *punto di ingresso* nel ciclo e quello evidenziato con (**) è il *punto di uscita* dal ciclo. "$b \neq 0$" rappresenta la *condizione di uscita* dal ciclo: l'iterazione prosegue finché "$b \neq 0$" non diventa falsa (e cioè diventa $b = 0$). La presenza di almeno una condizione d'uscita in un ciclo è condizione necessaria (anche se, in generale, non sufficiente) a garantire che il ciclo possa terminare in un tempo finito.

Osserviamo che in questo ciclo, nel caso in cui la condizione d'uscita sia falsa già dalla prima iterazione, i blocchi all'interno del ciclo vengono saltati

("eseguiti zero volte"). Ovviamente sono realizzabili anche altre strutture di controllo iterative, ad esempio con la condizione d'uscita posta al termine del ciclo invece che all'inizio.

Le istruzioni all'interno di un ciclo possono avere una semplice struttura sequenziale, come nell'esempio di Figura 1.2, oppure avere a loro volta una struttura di controllo più complessa, ad esempio condizionale o ciclica. In quest'ultimo caso, in particolare, si parlerà di *cicli "annidati"*, ovvero uno dentro all'altro.

Si noti che il fatto che il numero di istruzioni presenti nella descrizione di un algoritmo sia finito non implica necessariamente che l'algoritmo termini in un tempo finito. Ad esempio, se nell'algoritmo dell'Esempio 1.3 si immettesse un valore di b negativo, allora la condizione d'uscita dal ciclo "$b = 0$" non potrebbe mai verificarsi dato che il decremento unitario di b porterebbe b ad assumere valori via via più piccoli, ma sempre diversi da 0. Dunque, in questo caso l'algoritmo chiaramente non termina in un tempo finito e cioè si trova in quello che in gergo è chiamato un *ciclo infinito* (o, usando un termine inglese, un *loop* infinito).

Infine, osserviamo che tutte le strutture di controllo qui considerate sono caratterizzate dall'avere un'unica entrata ed un'unica uscita. Come vedremo in un successivo capitolo, questa è una condizione fondamentale per la realizzazione della cosiddetta *programmazione strutturata*.

Casi particolari, ottimizzazioni

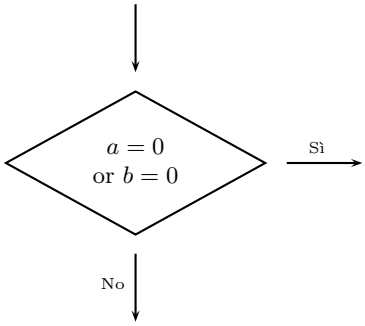
Uno stesso problema può essere risolto tramite più algoritmi, equivalenti dal punto di vista dei risultati prodotti, ma diversi dal punto di vista dell'efficienza di calcolo. Un'analisi (anche empirica) di un algoritmo prodotto può essere utile, in generale, per evidenziare casi particolari in cui l'algoritmo presenta un comportamento non soddisfacente (seppur corretto), e per i quali può essere opportuno prevedere una diversa formulazione.

Riferendoci ancora all'Esempio 1.3 analizziamo cosa accade nel caso particolare in cui uno dei due dati in ingresso sia `0`. La tabella in Figura 1.4 mostra che l'algoritmo funziona correttamente, ma nel secondo caso proposto risulta particolarmente inefficiente.

	(i)			(ii)	
a	b	p	a	b	p
3	**0**	**0**	**0**	**3**	0
				2	0
				1	0
				0	**0**

Figura 1.4: Esecuzione di 3×0 (caso (i)) e 0×3 (caso (ii)) con l'algoritmo di *moltiplicazione per somme*.

Per evitare che vengano ripetute somme con 0 si può modificare la condizione d'uscita dal ciclo utilizzando un'espressione booleana più complessa:



In questo caso, se a o b sono 0, l'algoritmo termina immediatamente restituendo (correttamente) 0 come suo risultato.

In generale si osserva che si avrebbe un miglioramento di efficienza dell'algoritmo se si usasse come moltiplicatore il minore tra a e b (tanto più rilevante quanto più grande è la differenza tra a e b). Possiamo perciò pensare ad una variante dell'algoritmo di moltiplicazione per somme che provveda eventualmente a scambiare tra loro i due termini a e b in modo tale da avere come moltiplicatore sempre il minore tra i due (l'operazione di scambio non pregiudica la correttezza dell'algoritmo grazie alla proprietà di commutatività dell'operazione di moltiplicazione). In Figura 1.5 è mostrato il diagramma di flusso relativo a questa versione migliorata dell'algoritmo di moltiplicazione per somme. Lo scambio tra le due variabili a e b viene fatto (quando richiesto) utilizzando una variabile temporanea t.

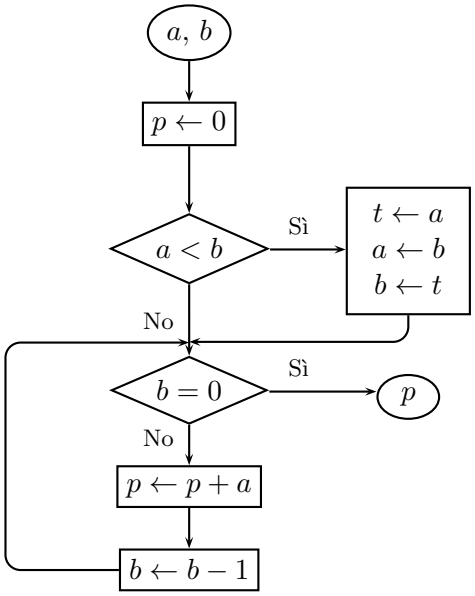


Figura 1.5: Algoritmo ottimizzato di moltiplicazione per somme.

1.4 I linguaggi di programmazione

Un linguaggio di programmazione è un formalismo di descrizione di algoritmi eseguibile dal calcolatore. La descrizione di un algoritmo per un problema $\mathcal{P}$ in un linguaggio di programmazione $\mathcal{L}$, e cioè un *programma in* $\mathcal{L}$, è costituito da una sequenza finita di istruzioni di $\mathcal{L}$ la cui esecuzione da parte del calcolatore porta (o dovrebbe portare) alla risoluzione di $\mathcal{P}$.

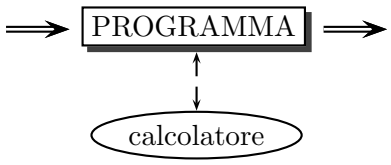
Un linguaggio di programmazione è caratterizzato formalmente da:

- una *sintassi*, la quale specifica la forma che possono avere le istruzioni e i programmi scrivibili con il linguaggio;
- una *semantica*, la quale specifica il significato e/o il funzionamento delle singole istruzioni e dei programmi realizzati con esse.

Per una trattazione approfondita delle problematiche inerenti la semantica dei linguaggi di programmazione si vedano, ad esempio, i testi [8] e [12].

Il fatto che un programma scritto in un linguaggio di programmazione sia *eseguibile* dal calcolatore significa che quest'ultimo sarà in grado di comprendere le istruzioni (sintatticamente corrette) e di eseguirle secondo la loro semantica.

Lo schema di Figura 1.1 assume perciò in questo caso la seguente forma:



Vedremo come questo schema generale possa essere ulteriormente precisato distinguendo tra vari tipi di linguaggi di programmazione e di modalitrà d'esecuzione di un programma da parte del calcolatore.

1.4.1 Linguaggi "a basso livello" e "ad alto livello"

Una prima (grossolana) classificazione dei linguaggi di programmazione si ottiene distinguendo tra linguaggi ad *alto livello* e linguaggi a *basso livello*, in base al livello di astrazione che essi forniscono rispetto al calcolatore.

La distinzione, in generale, non è così netta ed è possibile individuare varie gradazioni di "alto livello". In questo testo indicheremo semplicemente con linguaggi "a basso livello" quelli che sono strettamente dipendenti da una specifica macchina hardware, e cioè il *linguaggio macchina* e il *linguaggio*

assembly. Tutti gli altri linguaggi verranno genericamente indicati come linguaggi "ad alto livello".[4]

Ogni diverso tipo di macchina hardware ha un proprio *linguaggio macchina*: le istruzioni dei programmi scritti in tale linguaggio sono lette ed eseguite ("interpretate") direttamente dalla macchina hardware. Pertanto queste istruzioni sono scritte direttamente in notazione binaria (e cioè sequenze di 0 e 1) e quindi molto lontane dal linguaggio naturale e molto complesse da utilizzare per il programmatore.

Approfondimento (Linguaggio assembly). Generalmente per ogni linguaggio macchina esiste anche una versione più simbolica, detta *linguaggio assembly*. Essa utilizza nomi simbolici per codici di istruzioni ed indirizzi di memoria e la notazione decimale per rappresentare i numeri: questo la rende pertanto di più facile ed immediato utilizzo da parte del programmatore. Si tratta comunque ancora di un linguaggio a basso livello, dipendente dalla specifico tipo di macchina hardware. L'esecuzione di programmi scritti in linguaggio assembly richiede l'utilizzo di programmi traduttori, detti *assemblatori*, in grado di trasformare il programma scritto in linguaggio assembly nel corrispondente programma in linguaggio macchina (e quindi comprensibile per il calcolatore).

Per un approfondimento su linguaggio macchina e assembly, nonché, in generale, su tutto ciò che riguarda la macchina hardware, si veda ad esempio il testo [11].

I linguaggi ad alto livello nascondono, in modo più o meno completo, le caratteristiche proprie delle diverse macchine hardware ed offrono al programmatore una sintassi molto più vicina al linguaggio naturale rispetto a quanto possono fare i linguaggi macchina o assembly. Essi risultano pertanto più semplici da utilizzare ed indipendenti da una specifica macchina hardware.

L'esecuzione di un programma scritto in un linguaggio ad alto livello richiederà però la presenza di adeguati strumenti che rendano possibile la sua esecuzione sulla macchina hardware sottostante. Si distinguono solitamente due tecniche di realizzazione: la *compilazione* e l'*interpretazione* (si veda 1.4.3).

1.4.2 Linguaggi di programmazione esistenti

I linguaggi di programmazione ad alto livello esistenti attualmente sono moltissimi, sicuramente oltre il migliaio (per un elenco dettagliato si veda ad esempio la pagina Web dell'enciclopedia libera Wikipedia all'indirizzo `http://en.wikipedia.org/wiki/Timeline_of_programming_languages`). Qui di seguito diamo solo un brevissimo elenco indicando, in ordine cronologico, i nomi di alcuni dei linguaggi di programmazione più noti.

[4] Rispetto a questa classificazione il C ed il C++ risultano di livello "più basso" di altri linguaggi in quanto permettono, volendo, di accedere direttamente a certe caratteristiche della macchina hardware sottostante, rendendo, di fatto, i programmi che le utilizzano dipendenti da quest'ultima. In altri linguaggi, come ad esempio il Pascal, questa possibilità e del tutto impedita.

anni '50-'60:	Fortran
	Cobol
	LISP
	Basic
	PL/1
anni '70:	Pascal
	C
	Prolog
	Modula
anni '80:	C++
	Ada
	Perl
anni '90-2000:	Visual Basic
	Delphi
	Java
	C#

Rimandiamo alla letteratura specializzata sull'argomento (ad esempio [9]) qualsiasi descrizione ed analisi comparativa dei diversi linguaggi di programmazione oggi in uso.

Aggiungiamo soltanto che spesso si classificano i linguaggi in base al "paradigma di programmazione" da essi supportato. Si distingue principalmente tra quattro diversi paradigmi:

- *imperativo*
- *logico*
- *funzionale*
- *orientato agli oggetti*.

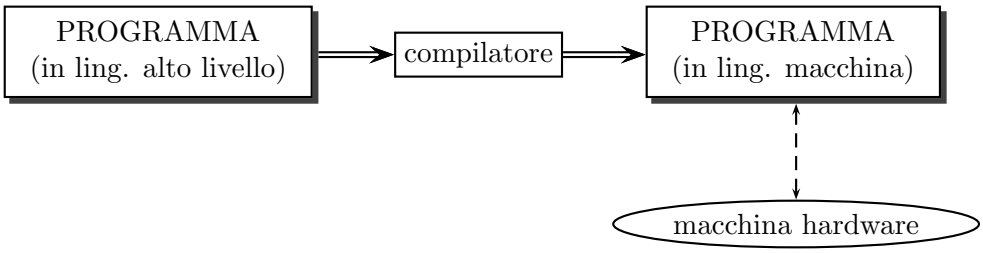
Anche per questo argomento rimandiamo alla letteratura specializzata.

In queste note utilizzeremo come linguaggio di riferimento il linguaggio C++, che può classificarsi come un linguaggio ad alto livello che supporta i paradigmi di programmazione imperativo e orientato agli oggetti. Precisamente, nella prima parte di queste note ci occuperemo soltanto degli aspetti del linguaggio relativi al paradigma imperativo, mentre nella seconda parte amplieremo il discorso anche a vari aspetti relativi alla programmazione orientata agli oggetti.

1.4.3 Modalità d'esecuzione: compilazione e interpretazione

La *compilazione* di un programma consiste nell'analisi delle istruzioni del programma che vengono lette e tradotte da un programma traduttore, il *compilatore*, e nella generazione delle istruzioni macchina in grado, quando

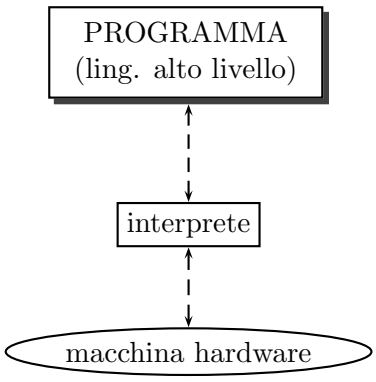
verranno successivamente eseguite dalla macchina hardware, di realizzare il comportamento voluto. Sono solitamente realizzati in questo modo la maggior parte dei linguaggi di programmazione "convenzionali", come ad esempio il Pascal ed il C.



L'*interpretazione* consiste nell'analisi delle istruzioni del programma e nell'immediata esecuzione di esse tramite un apposito programma, l'*interprete*, in grado di realizzare per ciascuna di esse il comportamento previsto. L'interprete è dunque un programma che svolge ciclicamente le seguenti azioni, fino ad incontrare un'istruzione di terminazione dell'esecuzione:

- legge ed analizza l'istruzione corrente;
- esegue l'istruzione;
- passa all'istruzione successiva.[5]

Sono solitamente realizzati in questo modo linguaggi di programmazione "non convenzionali", quali il Prolog o il LISP.

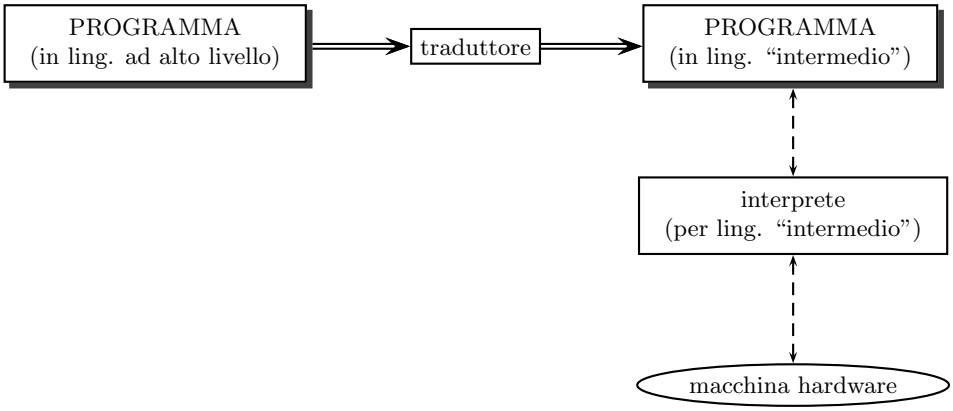


Se vogliamo fare un confronto tra queste due modalità d'esecuzione, seppur a livello molto superficiale, possiamo osservare che, poiché l'interpretazione richiede la presenza di un programma interprete che si frappone tra

[5]Si osservi che anche la macchina hardware è di fatto un interprete, ma realizzato a livello hardware, piuttosto che software, ed in grado di eseguire istruzioni di un linguaggio molto semplice, il cosiddetto linguaggio macchina.

il programma da eseguire e l'esecutore vero e proprio, ovvero la macchina hardware, l'esecuzione di un programma tramite questa modalità risulterà in generale più lenta e comporterà una maggiore occupazione di memoria. D'altra parte, l'esecuzione tramite interpretazione rende il linguaggio di programmazione più adatto ad uno sviluppo dei programmi interattivo, proprio perché non vi è una fase di traduzione separata ed il debugging del programma utente può essere svolto direttamente dall'interprete sulle istruzioni del linguaggio ad alto livello.

Approfondimento (Tecniche miste). Le due modalità descritte sopra, e cioè interpretazione e compilazione, rappresentano due situazioni estreme. Spesso, vengono adottate soluzioni "miste" che combinano, a vari livelli, le due tecniche. In queste soluzioni è presente una prima fase di traduzione dal linguaggio ad alto livello ad un *linguaggio "intermedio"*, un linguaggio molto più vicino ai linguaggi macchina, ma ancora indipendente dalle caratteristiche della specifica macchina hardware. Segue poi una seconda fase in cui il programma in linguaggio intermedio viene eseguito tramite interpretazione, su una specifica macchina hardware.



Questo è ad esempio l'approccio tipicamente adottato per l'implementazione del linguaggio Java: un programma Java viene tradotto in un programma equivalente nel linguaggio intermedio *Byte-code*, che verrà poi interpretato tramite un programma interprete, denominato "Java Virtual Machine" (JVM) (più precisamente, l'interprete realizza la *macchina astratta* JVM, che offre il Byte-code come proprio "linguaggio macchina"). Analogamente, un programma in linguaggio Prolog viene normalmente prima tradotto in un programma in un apposito linguaggio intermedio le cui istruzioni verranno poi interpretate da un interprete denominato "Warren Abstract Machine" (WAM).

In queste note ci occuperemo soltanto di linguaggi ad alto livello compilati (specificatamente del C++), senza peraltro addentrarci nelle problematiche specifiche dell'implementazione di un linguaggio di programmazione, che esulano dai nostri scopi. Per una trattazione più completa e formale delle nozioni di interpretazione, macchina astratta ed, in generale, di implementazione di un linguaggio, si veda, ad esempio, il testo [6].

1.5 Il linguaggio C++

1.5.1 Dal C al C++

Il C venne sviluppato nel 1972 da Dennis Ritchie all'interno dei Laboratori Bell. Il suo progetto è inizialmente strettamente legato a quello del Sistema Operativo UNIX: buona parte di UNIX e delle sue utilities di sistema sono scritte in C. Il C nasce dunque principalmente come linguaggio per la *programmazione di sistema* e quindi ad ampio spettro di applicabilità. Requisiti fondamentali nella definizione del linguaggio sono l'efficienza e la flessibilità d'uso. Come conseguenza di queste scelte di fondo il C risulta essere un linguaggio piuttosto "permissivo".

Il C++ venne sviluppato da Bjarne Stroustrup sempre all'interno dei Laboratori Bell AT&T agli inizi degli anni '80. Il C++ nasce come estensione del C: include tutte le possibilità del C (è quindi possibile usare un compilatore C++ anche per sviluppare ed eseguire programmi in C), più molte altre. In particolare, il C++ introduce in più rispetto al C i concetti di *classe*, *oggetto* ed *ereditarietà* tra classi, che fanno del C++ un ottimo supporto per la programmazione orientata agli oggetti. Altre nuove possibilità presenti in C++, non strettamente correlate con la programmazione ad oggetti, sono l'*overloading di operatori e funzioni*, il passaggio parametri per riferimento, nuove forme di gestione dell'input/output, molte nuove librerie, nuove possibilità per la gestione delle stringhe, la gestione delle eccezioni, ... Tutte queste caratteristiche permettono una programmazione più semplice, più chiara e più affidabile.

Nella prima parte di queste dispense analizzeremo un sottoinsieme del C++ che contiene la maggior parte delle possibilità offerte dal C, con l'aggiunta di alcune possibilità proprie del C++ e non presenti, o meno eleganti, in C (come ad esempio il passaggio dei parametri per riferimento e l'input ed output tramite gli operatori ">>" e "<<"). Resteranno invece escluse dalla I Parte tutte quelle possibilità strettamente connesse con la programmazione orientata agli oggetti (come ad esempio le nozioni di classe e quella di ereditarietà), ma anche alcune nozioni non relative alla programmazione orientata agli oggetti (come ad esempio l'overloading di funzioni ed operatori ed i puntatori), per le quali preferiamo rimandare la trattazione ad un momento successivo, allo scopo di rendere la presentazione più semplice e graduale. Queste ulteriori possibilità del linguaggio, insieme ai principali costrutti per il supporto alla programmazione orientata agli oggetti, verranno introdotte nella II Parte.

Nella I come nella II Parte, comunque, utilizzeremo come linguaggio di riferimento il C++, senza ulteriori precisazioni.

Nota. Il linguaggio C, e di conseguenza anche il C++, è un linguaggio intenzionalmente molto "permissivo", che consente ad esempio di mescolare liberamente numeri interi con caratteri e booleani o addirittura con puntatori. Questo motivato, essenzialmente, dal desiderio di avere un linguaggio ad ampio spettro di applicabilità. Molto diverse sono le

motivazioni che stanno alla base di un altro linguaggio di programmazione molto comune, "coetaneo" del C: il Pascal. Quest'ultimo, infatti, ha come suo obiettivo principale la scrittura di programmi "sicuri" e cerca di ottenere questo imponendo una programmazione disciplinata. Una conseguenza di tali diverse impostazioni di fondo tra i due linguaggi è, ad esempio, una filosofia profondamente diversa di trattamento e controllo dei tipi.

1.5.2 Un esempio di programma C++

In questo sottocapitolo mostriamo un primo semplice esempio di programma scritto in C++. Tutti i diversi costrutti del C++ qui introdotti in modo intuitivo verranno ripresi ed analizzati in modo più approfondito nei capitoli successivi.

Un programma C++—così come i programmi scritti nella maggior parte dei linguaggi di programmazione convenzionali—è costituito fondamentalmente da due tipi di costrutti linguistici: le *dichiarazioni* e gli *statement* (o comandi). Le prime servono a descrivere i dati utilizzati nel programma; gli statement invece costituiscono le istruzioni vere e proprie del linguaggio e permettono di specificare le azioni da compiere per giungere alla soluzione del problema considerato.

Si noti che nei diagrammi di flusso ci siamo basati su una descrizione dei dati informale, mentre nei linguaggi di programmazione questo aspetto è formalizzato in modo preciso e quindi anche la descrizione dei dati, oltre che quella delle azioni, dovrà rispettare precise regole sintattiche.

Il problema che vogliamo risolvere con questo primo programma C++ è un problema estremamente semplice, ma sufficiente a mettere già in evidenza alcune caratteristiche salienti di un tipico programma in un linguaggio di programmazione convenzionale.[6]

Problema: dati in input 3 numeri interi, calcolare e dare in output la loro media.
Input: 3 numeri interi.
Output: un numero reale.

Programma:

```
#include <iostream>
using namespace std;
int main() {
  int x, y, z;
  cout << "Dammi 3 numeri interi" << endl;
  cin >> x >> y >> z;
  float m;
```

[6]Tutto il codice presentato in queste note è stato compilato ed eseguito con compilatore gcc 3.3.3 20040412 (Red Hat Linux 3.3.3-7) Copyright (C) 2003 Free Software Foundation, Inc.

```
  m = (x + y + z) / 3.0;
  cout << "La media e' " << m << endl;
  return 0;
}
```

Osservazioni sul programma.

- `#include <iostream>` (direttiva *include*) indica che il programma ha bisogno di utilizzare ("includere") delle funzioni predefinite per la gestione dell'input/output le cui dichiarazioni sono contenute nel file `iostream` della libreria standard del C++.

- `using namespace std;` indica che il programma utilizza degli *identificatori*, non dichiarati nel programma stesso, che appartengono allo "spazio dei nomi" standard (`std`); in particolare, gli identificatori `cin` e `cout`, usati per identificare l'input e l'output del programma, nonché l'identificatore `endl`, usato per produrre un "a capo" in output, devono essere cercati nello spazio dei nomi standard.

- `int main()` indica che si tratta del *programma principale*; `int` e `main` sono due *parole chiave* e, come tali, fissate nel linguaggio; le parentesi tonde aperte e chiuse dopo `main()` indicano assenza di parametri dall'esterno.

- La *dichiarazione di variabile*

  ```
  int x, y, z;
  ```

  definisce tre nuove variabili, di nome `x`, `y` e `z`, tutte e tre di *tipo intero* (`int`). Il carattere "`;`" è usato in C++ come "terminatore" di dichiarazioni e statement.

  La dichiarazione di variabile

  ```
  float m;
  ```

  definisce una nuova variabile di nome `m` e *tipo reale* (`float`).

  Si noti che qualsiasi variabile, prima di essere usata, deve essere dichiarata in una parte del programma che precede l'uso della variabile stessa.

- Lo statement

  ```
  cout << "Dammi 3 numeri interi" << endl;
  ```

indica un'operazione di output: la stringa `"Dammi 3 numeri interi"` viene inviata sul dispositivo di output standard (normalmente il monitor), qui identificato dal nome `cout`, seguita dall'invio del carattere speciale di "a capo". "`<<`" è l'operatore di output (detto anche *operatore di inserimento*).

L'esecuzione continua immediatamente.

Lo statement

```
cin >> x >> y >> z;
```

indica un'operazione di input: si attende che tramite il dispositivo di input standard (solitamente la tastiera), qui identificato dal nome `cin`, vengano forniti al programma tre numeri interi, che verranno assegnati rispettivamente alle variabili `x`, `y` e `z`. "`>>`" è l'operatore di input (detto anche *operatore di estrazione*).

L'esecuzione continua soltanto quando tutti i valori attesi (tre numeri interi in questo caso) sono stati forniti.

- Lo statement

```
m = (x + y + z) / 3.0;
```

indica un'operazione di assegnamento: l'espressione a destra dell'uguale, `(x + y + z) / 3.0`, viene valutata ed il suo risultato viene assegnato alla (ovvero diventa il nuovo valore della) variabile `m`.

Se ad esempio `x` vale 4, `y` vale 6 e `z` vale 3, il valore di `m` diventa 4.33333.

- Lo statement

```
return 0;
```

indica che il programma termina restituendo all'*ambiente esterno* (cioè al sistema operativo, da cui il programma è stato inizialmente attivato) il valore 0 (il valore 0 viene tipicamente usato per indicare che il programma è terminato correttamente).

Nota. Alcuni compilatori C/C++ (in particolare Dev-C++ e Borland C++) prevedono che la finestra di output venga chiusa automaticamente al termine dell'esecuzione del programma. Per evitare che il programma termini senza dare la possibilità all'utente di prendere visione degli eventuali risultati visualizzati sulla finestra di output, è opportuno inserire al termine del programma un'istruzione che ne blocchi l'esecuzione. Questo si può ottenere, ad esempio, inserendo prima dell'istruzione `return 0`, le seguenti due istruzioni:

```
cout << "Digita un carattere qualsiasi per terminare ";
cin.get();
```

L'esecuzione dell'istruzione `cin.get()` pone il programma in attesa che l'utente digiti sulla tastiera un qualsiasi carattere (compresi il carattere "spazio" e "a capo"). Finché questo non avviene il programma non termina e quindi la finestra di output permane.

In alternativa alla `get()` si può utilizzare la funzione `getch()`, offerta da alcuni compilatori (tra cui i già citati Dev-C++ e Borland C++, ma non il `gcc` sotto Linux) come facente parte della libreria `conio.h`. Questa funzione si comporta essenzialmente come la `get()`, ma, a differenza di quest'ultima, evita di fare l'"echo" del carattere digitato sul monitor, risultando così, in questo specifico caso, più "naturale" per l'utente (in questo caso, infatti, non è necessario "vedere" quale carattere si è digitato, ma interessa soltanto averlo digitato). Volendo utilizzare questa alternativa il codice da inserire prima dell'istruzione `return 0`, diventa:

```
cout << "Digita un carattere qualsiasi per terminare ";
getch();
```

mentre, all'inizio del programma, andrà inserita la direttiva `#include <conio.h>`.

Per completezza, e per permettere un facile confronto, descriviamo l'algoritmo relativo al programma C++ appena mostrato anche con un diagramma di flusso (Figura 1.6).

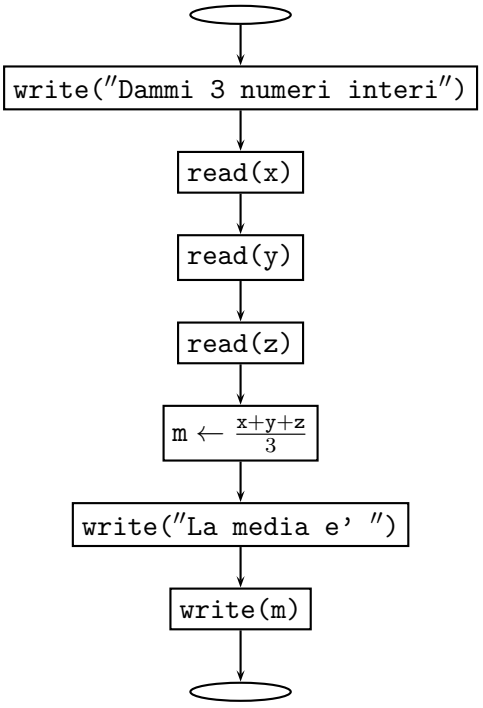


Figura 1.6: Diagramma di flusso per il calcolo della media di tre numeri interi.

1.5.3 Convenzioni di programmazione

Nella stesura di un programma è opportuno, al fine di migliorarne la leggibilità, attenersi ad alcune convenzioni di scrittura.

- Evidenziazione delle "parole chiave".
  Le "parole chiave" sono elementi costitutivi del linguaggio di programmazione e come tali non modificabili. Ad esempio, `main`, `int`, `return` sono parole chiave del C++. Per aumentare la leggibilità di un programma si adotta solitamente la convenzione di evidenziare le parole chiave, ad esempio sottolineandole. Quando il programma viene scritto utilizzando un editor fornito da un ambiente di programmazione per uno specifico linguaggio solitamente l'editor riconosce le parole chiave e provvede automaticamente ad evidenziarle (ad esempio con il carattere "grassetto" oppure colorandole in modo diverso). Notare che l'evidenziazione delle parole chiave è utile all'utente umano, ma del tutto ignorata dal compilatore.

- "Indentatura".
  La separazione delle diverse parti di un programma su più righe ed il loro spostamento ed allineamento rispetto al margine sinistro (e cioè l'*"indentatura"*) sono essenziali per il programmatore, e per chiunque debba leggere il programma, per comprenderne la struttura, e quindi migliorarne la comprensibilità. Ad esempio è tipico in un programma C++ allineare sulla stessa colonna (e cioè alla stessa distanza dal margine sinistro) una parentesi graffa aperta e la corrispondente graffa chiusa, oppure spostare verso destra ed allineare sulla stessa colonna tutte le dichiarazioni e gli statement compresi tra due parentesi graffe. L'utilizzo di queste convenzioni è già visibile nel semplice programma mostrato in questo capitolo, ma diventerà via via più evidente man mano che verranno mostrati nuovi programmi d'esempio. Si noti comunque che, come per l'evidenziazione delle parole chiave, anche per l'"indentatura" si tratta di una convenzione utile all'utente umano ma del tutto ignorata dal compilatore (in linea di principio un programma potrebbe anche essere scritto tutto su un'unica riga).

- Commenti.
  I commenti sono frasi in formato libero che possono essere aggiunte in qualsiasi punto di un programma e che possono servire al programmatore per fornire informazioni aggiuntive sul significato e l'utilizzo del programma o di parti di esso (ad esempio, l'utilizzo di una certa variabile). I commenti servono esclusivamente per rendere il programma più facile da capire sia per chi lo scrive che per chi deve leggerlo e comprenderlo, e sono del tutto ignorati dal compilatore.

In C++ sono possibili due modalità per aggiungere commenti ad un programma:

- `/* qualsiasi frase,`
  `anche su piu' righe */`
- `// qualsiasi frase` (fino a fine riga).

Ad esempio:

```
//programma di prova
#include <iostream>
using namespace std;
int main() { //inizio il programma
     ...
     cin >> x >> y >> z;
     /* A questo punto x, y, z contengono
     i tre numeri interi forniti tramite tastiera. */
     m = (x + y + z) / 3.0;
     ...
}
```

1.6 Ambiente di sviluppo programmi

Per lo sviluppo e messa a punto di un programma in un certo linguaggio di programmazione si ha bisogno normalmente non soltanto di un compilatore per quel linguaggio, ma anche di un insieme di altri strumenti software che siano di ausilio alla scrittura e quindi all'esecuzione ed eventuale modifica di un programma. Questo insieme di strumenti—più o meno sofisticati, più o meno integrati fra loro, e più o meno correlati ad uno specifico linguaggio—è indicato come un *ambiente integrato di sviluppo* (in inglese *"Integrated Development Environment"* o, più semplicemente, *IDE*). In Figura 1.7 è schematizzato un tipico semplice IDE per un linguaggio compilato, come ad esempio il C/C++.

Per poter essere eseguito un programma passa attraverso 5 fasi principali: *"editing"*, *compilazione*, *"linking"*, *caricamento* ed *esecuzione*.

La prima fase consiste nella scrittura in un linguaggio di programmazione $\mathcal{L}$ del programma (*programma sorgente*) e nel suo salvataggio in un file. Questa fase viene eseguita attraverso un programma chiamato *editor*, quale, ad esempio, `vi` o `emacs` in ambiente Unix/Linux, e `Blocco note` in ambiente Windows. Spesso l'editor è integrato con l'ambiente di programmazione e permette di riconoscere ed evidenziare alcuni elementi sintattici del linguaggio, come ad esempio le parole chiave, e di richiamare gli altri strumenti dell'ambiente in modo diretto.

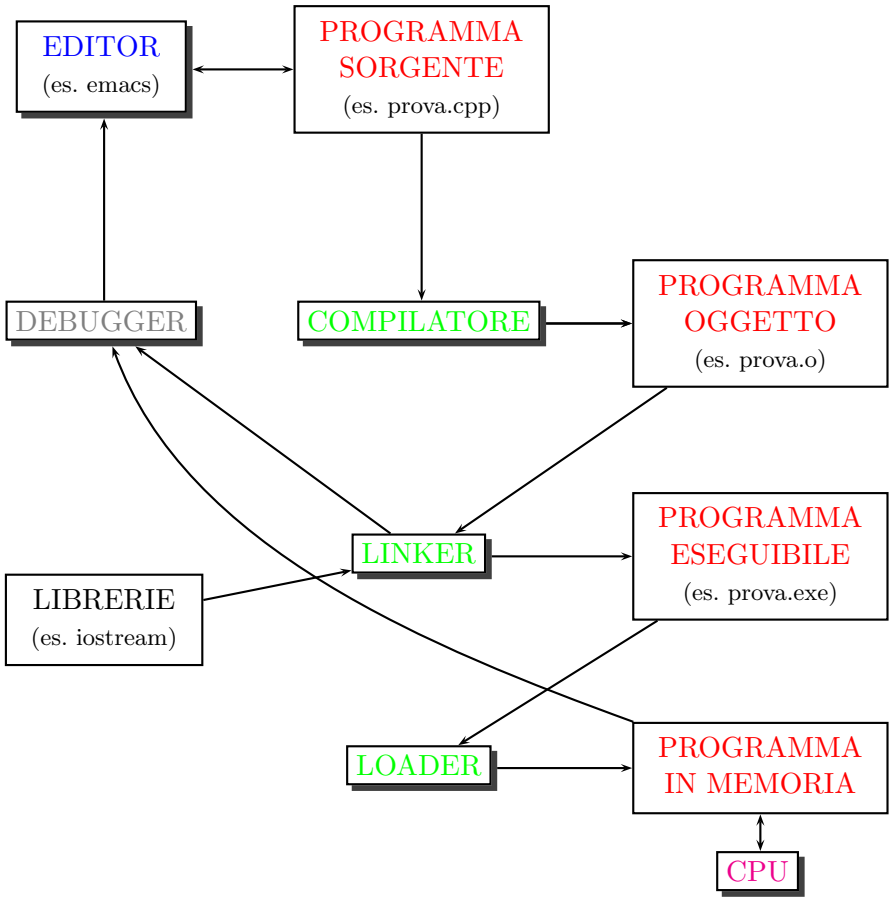


Figura 1.7: Schema di un ambiente di programmazione.

Nella seconda fase, quella di *compilazione*, il programma in linguaggio ad alto livello viene trasformato nel corrispondente programma in linguaggio macchina (*programma oggetto*), tramite un programma traduttore, detto *compilatore*. La fase di compilazione è, in generale, molto complessa e si suddivide a sua volta in altre sotto-fasi che prevedono, tra l'altro, l'*analisi lessicale*, *sintattica* e *semantica* del codice da tradurre. Non ci soffermeremo su questi argomenti rimandando il lettore interessato ad approfondirli, ad esempio, in [1].

La terza fase è quella del *linking* (che in italiano si può tradurre con *collegamento*) e prevede l'utilizzo di uno strumento software detto *linker*. Solitamente i programmi sorgenti contengono riferimenti a funzioni compilate separatamente, per esempio quelle fornite dalle *librerie standard*. Il linker provvede a "cucire" insieme il programma oggetto corrente, prodotto

dal compilatore, con il codice oggetto delle altre funzioni esterne a cui il programma fa riferimento, andando anche a completare i riferimenti esterni presenti in quest'ultimo che erano stati necessariamente lasciati in sospeso durante la fase di compilazione. Il linker crea così un *programma eseguibile* completo (solitamente di dimensioni molto maggiori rispetto al codice oggetto del programma utente).

La quarta fase è chiamata *caricamento*. Un programma eseguibile, perché possa essere eseguito dalla macchina hardware sottostante, deve essere caricato nella memoria centrale. Questa operazione viene svolta da un programma di sistema detto *loader*. Il loader, una volta terminato il caricamento in memoria del programma utente, provvede anche ad attivarne l'esecuzione, facendo in modo che la CPU continui la sua esecuzione a partire dall'istruzione del programma indicata come punto iniziale di esecuzione (*entry point*).

Il comando *run* (o *execute*, o simili), tipicamente fornito dall'ambiente di programmazione, permette di eseguire in un colpo solo tutte le fasi che vanno dalla compilazione all'esecuzione del programma. L'ambiente fornisce tipicamente anche altri comandi che permettono di svolgere solo alcune delle fasi sopra indicate. Ad esempio, è possibile eseguire solo la compilazione di un programma sorgente (comando *compile*, o simili), ottenendo come risultato il corrispondente programma oggetto o la generazione di opportune segnalazioni nel caso siano presenti errori sintattici.

Approfondimento ("Preprocessing"). Il linguaggio C/C++ prevede anche una fase di trasformazione del programma sorgente, precedente a quella di compilazione, detta fase di *preprocessing*. Si tratta di una *trasformazione di programma*, effettuata da uno strumento detto *preprocessore*, che avviene all'interno dello stesso linguaggio: il programma in input è in linguaggio $\mathcal{L}$ e tale sarà anche il programma prodotto. Nel caso del C/C++, il preprocessor individua ed esegue comandi speciali, chiamati *direttive del preprocessore*, riconoscibili sintatticamente per il carattere iniziale "`#`", che indicano che sul programma devono essere eseguite alcune manipolazioni, come per esempio l'inclusione di codice sorgente contenuto in altri file (direttiva `#include`), o la sostituzione di stringhe con altre stringhe (direttiva `#define`).

I nomi dei file contenenti il codice del programma nelle sue varie fasi di traduzione terminano solitamente con opportune *estensioni* che indicano il tipo di informazione contenuta nel file. Ad esempio, nel caso del C++, i nomi dei file contenenti il codice sorgente terminano tipicamente con estensioni quali `.cc`, `.cpp`, `.c++` (ad esempio, `prova.cpp`), mentre i nomi dei file contenenti il codice oggetto terminano tipicamente con estensione `.o` oppure `.obj` (ad esempio, `prova.o`), e quelli contenenti codice eseguibile con l'estensione `.exe` (ad esempio, `prova.exe`).

In Figura 1.7 è evidenziata anche la presenza di un altro strumento tipico di un ambiente di programmazione, il *debugger*. Si tratta di un programma che è in grado di eseguire passo passo il programma utente caricato in memo-

ria, comunicando con l'utente in modo interattivo, e mostrando l'evoluzione dei valori delle variabili presenti nel programma durante la sua esecuzione.

1.7 *Domande per il Capitolo 1*

1. Dare la definizione generale di algoritmo. Cosa si intende con i termini "istruzioni", "operazioni elementari" e "classe di problemi" usati nella definizione di algoritmo?
2. Indicare almeno due proprietà importanti degli algoritmi.
3. Di cosa si occupano la "teoria della computabilità" e la "teoria della complessità"?
4. Da quali componenti è costituito in generale un linguaggio di descrizioni algoritmi?. Citare brevemente alcuni esempi di linguaggi di descrizione algoritmi.
5. Descrivere sinteticamente forma (sintassi) e significato (semantica) dei principali tipi di blocchi nei diagrammi di flusso.
6. Cosa si intende per sintassi e per semantica di un linguaggio di programmazione? Quali aspetti di un linguaggio di programmazione descrivono?
7. Cosa significa "basso" e "alto livello" riferito ad un linguaggio di programmazione? Come si può classificare il C++? Giustificare la risposta.
8. Indicare almeno due vantaggi dell'uso di un linguaggio "ad alto livello" rispetto ad uno "a basso livello".
9. Cosa si intende con compilazione di un programma?
10. Cosa si intende con interpretazione di un programma scritto in un linguaggio ad alto livello? Qual è il ciclo base di interpretazione?
11. Quali sono le differenze di base e i vantaggi/svantaggi tra compilazione ed interpretazione di un programma scritto in un linguaggio ad alto livello?
12. Descrivere in modo schematico (eventualmente con l'aiuto di un disegno) in cosa consiste l'esecuzione di un programma rispettivamente tramite compilazione e tramite interpretazione.
13. Descrivere in modo schematico (eventualmente con l'aiuto di disegni) la tecnica di esecuzione di un programma basata su combinazione di compilazione ed interpretazione. Come si attua nel caso di Java?
14. Quali sono i principali paradigmi di programmazione? Quali sono gli aspetti caratterizzanti del paradigma "imperativo"?

15. In che senso il C++ è un linguaggio di programmazione “permissivo”?

16. Cosa si intende con i termini “preprocessing” e “direttiva di preprocessor” in C++?

17. Cosa si intende con Ambiente di Sviluppo Integrato (IDE)? Quali sono i principali strumenti software che lo compongono e come interagiscono tra loro (illustrare con un disegno)?

18. Quali sono le funzioni di base di un “linker”?

19. Cosa si intende con i termini metodologie e strumenti di programmazione?