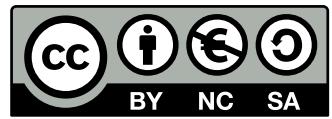


# **Fondamenti di Informatica**

*Gabriele de Capoa*

25 agosto 2022  
Versione 1.1

Quest'opera è distribuita con licenza Creative Commons "Attribuzione – Non commerciale – Condividi allo stesso modo 4.0 Internazionale".



# Prefazione

Questi fascicolo nasce dalla sistemazione degli appunti cartacei presi durante l'anno accademico 2008-2009, seguendo le lezioni del Professor Alfonso Miola e della Professoressa Carla Limongelli, tenutesi per l'omonimo corso della laurea triennale di Ingegneria Informatica dell'Università degli Studi Roma Tre. Non seguono in modo preciso l'ordine delle lezioni, ma piuttosto sono stati raggruppati per argomento, aggiungendo approfondimenti da ulteriori testi di consultazione e rimuovendo parti, in modo da renderlo auto-contenuto.

Poiché non sono stati revisionati né dai docenti del corso né da altre persone competenti in questo argomento, eventuali errori di concetto sono da ritenersi mie imprecisioni nella comprensione della materia.

# Indice

<b>Prefazione</b>	<b>iii</b>
<b>1 Nozioni di base</b>	<b>1</b>
1.1 Cos'è l'Informatica? . . . . .	1
1.1.1 I termini di uso comune . . . . .	2
1.2 Storia del calcolo automatico . . . . .	5
1.2.1 I primi strumenti di calcolo . . . . .	5
1.2.2 I precursori del calcolatore elettronico e dell'Informatica . . . . .	6
1.2.3 I primi del Novecento: la prima generazione di calcolatori . . . . .	10
1.2.4 La seconda generazione e la nascita dei linguaggi di programmazione . . . . .	13
1.2.5 La terza generazione dei calcolatori . . . . .	15
1.2.6 La quarta generazione: i <i>personal computer</i> . . . . .	16
<b>2 Architettura di un calcolatore</b>	<b>19</b>
2.1 Hardware e software . . . . .	19
2.2 La Macchina di Von Neumann . . . . .	21
2.2.1 La memoria . . . . .	21
2.2.2 Unità centrale di elaborazione . . . . .	26
2.2.3 Le interfacce di input/output . . . . .	28
2.2.4 Il bus . . . . .	28
2.3 La rappresentazione dell'informazione . . . . .	29
2.3.1 I sistemi di numerazione . . . . .	29
2.3.2 La rappresentazione binaria . . . . .	30
2.4 Il sistema operativo . . . . .	33
<b>3 Problemi, algoritmi e programmi</b>	<b>37</b>
3.1 Cos'è un problema? . . . . .	37
3.2 La metodologia di lavoro: dal problema all'algoritmo . . . . .	38
3.3 Dall'algoritmo al programma . . . . .	40
3.3.1 Diagrammi a blocchi e pseudocodifica . . . . .	40
3.3.2 La programmazione strutturata e il teorema di Böhm-Iacopini	42
<b>4 La programmazione</b>	<b>45</b>
4.1 Costruzione analitica di un linguaggio di programmazione . . . . .	45
4.1.1 Costruzione algebrica di un linguaggio di programmazione	46
4.1.2 Il formalismo BNF . . . . .	47

4.2 I sottoprogrammi . . . . .	47
4.3 Paradigmi di programmazione . . . . .	49
4.3.1 Traduzione ed Esecuzione dei Programmi . . . . .	50
4.4 Leggibilità del codice . . . . .	51
4.4.1 I commenti . . . . .	51
4.4.2 Scelta degli identificatori . . . . .	51
4.4.3 Indentazione . . . . .	52
<b>5 Tipi dati ed espressioni</b>	<b>53</b>
5.1 Tipo di dato . . . . .	53
5.1.1 Tipi primitivi . . . . .	54
5.1.2 Tipi derivati . . . . .	54
5.1.3 Conversione tra tipi . . . . .	55
5.1.4 Tipo di dato astratto . . . . .	56
5.2 Descrizione dei tipi derivati . . . . .	57
5.2.1 I puntatori e i riferimenti . . . . .	57
5.2.2 Gli array . . . . .	58
5.2.3 I record . . . . .	60
5.2.4 Tipi generici . . . . .	60
5.3 Strutture collegate . . . . .	60
5.3.1 La lista . . . . .	60
5.3.2 La pila . . . . .	62
5.3.3 La coda . . . . .	62
5.4 Espressioni . . . . .	63
<b>6 Correttezza e complessità</b>	<b>65</b>
6.1 Cosa si intende per correttezza? . . . . .	65
6.1.1 Errori nella programmazione . . . . .	67
6.1.2 Verifica di correttezza . . . . .	68
6.2 Complessità di un programma . . . . .	69
6.2.1 Analisi asintotica della complessità . . . . .	71
<b>7 Gestione della memoria e ricorsione</b>	<b>73</b>
7.1 Gestione della memoria a runtime . . . . .	73
7.2 La ricorsione . . . . .	75
7.2.1 Definizioni induttive in Matematica . . . . .	75
7.2.2 Tipi di dato ricorsivi . . . . .	76
7.2.3 Sottoprogrammi ricorsivi . . . . .	76
<b>8 Ordinamento</b>	<b>79</b>
8.1 Ordinamento per selezione . . . . .	79
8.2 Ordinamento per inserzione . . . . .	79
8.3 Ordinamento a bolle . . . . .	80
8.4 Ordinamento per fusione . . . . .	80
8.5 Ordinamento veloce . . . . .	80
<b>Bibliografia</b>	<b>81</b>

<b>Elenco delle figure</b>	<b>83</b>
<b>Indice analitico</b>	<b>85</b>

# Capitolo 1

## Nozioni di base

Al giorno d'oggi si parla spesso di *digitalizzazione*, di aumentare l'*alfabetizzazione informatica*. Questo perché l'Informatica è diventata sempre più presente nella nostra vita, dal posto di lavoro alla vita di tutti i giorni, grazie a cellulari, elettrodomestici, videogiochi, etc. Per questo motivo, studiare l'Informatica come una scienza è molto importante.

Tradizionalmente, le attività connesse al settore dell'Informatica venivano indicate con la sigla **EDP** (*Electronic Data Processing, elaborazione elettronica dei dati*). Oggi, vista l'estensione degli ambiti di applicazione dell'Informatica, si parla più spesso di **ICT** (*Information and Communication Technology*), per indicare l'insieme delle tecnologie utili a conservare, elaborare e trasmettere le informazioni, raggruppando anche l'Elettronica e le Telecomunicazioni.

Queste informazioni possono arrivare da qualsiasi fonte. Si va dai sensori degli impianti industriali ai filmati, dalle immagini alle chiamate VoIP, dai messaggi sui social network ai documenti testuali.

In questo capitolo vedremo in cosa consiste l'Informatica e qualche cenno storico sull'evoluzione dei calcolatori.

### 1.1 Cos'è l'Informatica?

L'Informatica non è qualcosa che nasce in Italia, ma che noi importiamo dall'estero, principalmente dalla Francia e dagli Stati Uniti. Il termine in italiano, infatti, deriva direttamente dal francese, nel cui vocabolario viene introdotto nel 1962 come crasi delle parole *Informazione* e *Automatica*. Sebbene nel 1962 i primi calcolatori elettronici già esistevano, il termine non fa alcun riferimento specifico all'uso di essi.

Nei Paesi anglo-sassoni, ove la scienza ha origine, il termine usato è *Computer Science* (scienza dei calcolatori), ma in realtà non è solo la scienza e la tecnologia dei calcolatori. Il termine fu usato per la prima volta in un articolo del 1959 della rivista "Communications of the ACM", nel quale Louis Fein discusse la creazione di una *Graduate School in Computer Sciences* analoga alla *Harvard Business School*, giustificando il nome dicendo che, come la *management science*, la *computer science*

è per sua natura una materia di studio applicata e interdisciplinare, avendo allo stesso tempo le caratteristiche tipiche di una disciplina accademica.

Dall'unione di queste due etimologie, nasce la definizione italiana dell'Informatica.

**Definizione 1.1** (Informatica). Si definisce **Informatica** la scienza della rappresentazione, dell'organizzazione e del trattamento (eventualmente automatico) dei dati.

Come affermato dall'articolo di Louis Fein, l'Informatica cerca di definire un approccio sistematico e disciplinato alla soluzione (automatica ove possibile) dei problemi grazie all'elaborazione dell'informazione.

Questo approccio è suddiviso in due ambiti:

- quello *astratto* e *metodologico*, legato all'informazione, ai problemi e agli algoritmi;
- quello *concreto* e *tecnologico*, legato al calcolatore, ai dati e ai programmi.

Ciò significa che l'Informatica si concentra molto sia sul *sapere* (cioè ragionare, conoscere, analizzare), sia sul *saper fare* (sapere come arrivare alla soluzione conoscendo in cosa consiste).

È curioso far notare il differente significato di origine tra l'italiano, il francese e l'inglese nel denominare il calcolatore elettronico.

- In italiano, infatti, si usa molto spesso *elaboratore* (o *calcolatore*), indicando così le sue svariate capacità di elaborazione (anche se oggi il termine più utilizzato è *computer*).
- In francese, il termine usato è *ordinateur* (*ordinatore*), a sottolineare le sue capacità di organizzare i dati e le informazioni.
- Il termine inglese, *computer* (letteralmente *calcolatore*, mostra la diretta discendenza delle calcolatrici, prima meccaniche, poi elettromeccaniche, poi elettroniche.

### 1.1.1 I termini di uso comune

Il fatto che l'Informatica non sia strettamente legata all'uso di un calcolatore è ben spiegato da un racconto del 1958, scritto da Isaac Asimov, intitolato "Nove volte sette" (in Inglese "*Feeling of Power*", presente nella raccolta "Racconti Matematici" edita da Einaudi nel 2006), che dimostra come sia necessario non far atrofizzare nessuna conoscenza e nessuna capacità.

Il racconto affronta il tema del rapporto tra scienza ed etica, proponendo come paradosso quello di smettere di usare i calcolatori elettronici e tornare all'uso di carta e penna. Il protagonista, infatti, scopre come fare i calcoli aritmetici manualmente, permettendo di sostituire tutti i calcolatori della nazione che stavano

controllando tutto.

Nel mondo moderno, smettere di usare un calcolatore elettronico ci sembrerebbe una eresia, tanto è diventato pervasivo nella nostra quotidianità. Lo spunto che offre il racconto, però, è che occorre capire bene cosa si intende per *informazione* per poi decidere come *rappresentarla*, come *trattarla* e *trasmetterla*. Se il trattamento coinvolge qualche automazione, occorre capire bene la struttura e il funzionamento dello strumento di elaborazione automatica, ma anche la formalizzazione del *problema*, in modo da poter definire una soluzione corretta ed efficiente (detta *algoritmo*) e le regole per automatizzarne la risoluzione (il *programma*).

Iniziamo a definire alcuni di questi concetti.

### L'informazione

Il termine *informazione* viene utilizzato in modo ampio in contesti diversi, per indicare elementi talvolta diversi tra loro.

**Definizione 1.2 (Informazione).** Si definisce con **informazione** tutto ciò che possiede un significato per l'uomo e che viene conservato o comunicato in vista di una utilità pratica, immediata o futura.

Una informazione presuppone da una parte un fenomeno del mondo reale o una persona che possiede una conoscenza, mentre dall'altra parte si presuppone una persona che vuole ottenere maggiore conoscenza.

L'informazione, inoltre, è legata ad uno scopo, un *fine*, per il quale viene raccolta o comunicata.

Le informazioni si presentano secondo diverse forme, dette *configurazioni* o **rappresentazioni**, come ad esempio:

- caratteri alfabetici;
- numeri;
- grafici;
- immagini;
- luci;
- suoni;
- gesti.

Alcune rappresentazioni sono di facile comprensione a livello universale, mentre altre sono comprensibili solo ad un gruppo specifico di persone. È dunque implicita l'esistenza di un insieme di regole comuni tra l'emittente dell'informazione e il ricevente, denominato **codice**.

## L'elaborazione delle informazioni

Nella nostra definizione iniziale di Informatica, abbiamo fatto riferimento al concetto di dato.

Essendo una scienza, l'Informatica è strettamente correlata all'osservazione del mondo reale. Questa osservazione ci consente di catturare gli elementi utili per poter esprimere una opinione, caratterizzare un fenomeno o risolvere un problema. Questi elementi sono i **dati**.

Il concetto di dato presuppone la volontà di voler conservare qualcosa nella memoria o negli archivi, in vista di un successivo trattamento per produrre le informazioni. Esiste, dunque, una distinzione tra il concetto di dato e quello di informazione.

- Un dato descrive aspetti elementari di entità o fenomeni.
- Una informazione è un insieme di dati elaborati e presentati sulla base di una esigenza di utilizzo pratico.

I dati possono essere *elementari* (come il prezzo di una bottiglia) oppure *composti* o *aggregati* (come una scatola di 12 bottiglie). I dati possono poi essere *primitivi* (l'importo di un assegno) o *risultanti da un calcolo* (il saldo del conto corrente), *organizzati in strutture standard* o *rappresentati in forma libera*.

**Definizione 1.3** (Elaborazione dei dati). Il trattamento dei dati per ottenere le informazioni prende il nome di **elaborazione**.

L'esempio più semplice di elaborazione è costituito dalla trascrizione dei dati, ma si parla di elaborazione anche nel caso del calcolo aritmetico, della trasformazione di figure piane o della costruzione di una teoria scientifica. Ogni tipo di elaborazione necessita dei dati in ingresso (*input*) per produrre dati in uscita (*output*).

## La comunicazione

Quando due soggetti comunicano, si stanno trasmettendo informazioni tramite un *canale*, avendo stabilito prima della trasmissione le regole per la trasformazione (il *codice*) in *segnali*. A seconda del mezzo trasmittivo, i segnali possono essere analogici o digitali.

La comunicazione è, di solito, un processo interattivo e bidirezionale, costituito da domande e risposte. Ciò presuppone che esista un insieme di parole chiave conosciute da entrambi i soggetti e un insieme di regole per combinare queste parole, cioè un *linguaggio*. In Informatica, si parla di:

- *linguaggio di comandi* per rappresentare l'insieme delle parole che un utente può usare per chiedere l'attivazione di un lavoro ad un calcolatore;
- *linguaggio di programmazione* per rappresentare l'insieme delle parole e delle regole usate nei programmi.

## 1.2 Storia del calcolo automatico

L'idea del calcolo automatico nasce con l'esigenza umana di evitare calcoli numerosi e ripetitivi, in modo veloce e senza errori. Analizzarne l'evoluzione storica aiuta a capire come è stato possibile arrivare al concetto di Informatica come lo conosciamo oggi.

### 1.2.1 I primi strumenti di calcolo

La storia del computer ha inizio con l'introduzione e l'utilizzo dei *numeri*: possiamo infatti vedere il calcolatore elettronico come una evoluzione della calcolatrice che usano gli studenti per fare i calcoli aritmetici. La prima realizzazione di uno strumento che semplificasse i calcoli fu l'*abaco* all'epoca dell'Antica Roma (Figura 1.1). L'abaco era una taboletta, suddivisa in colonne, su cui si spal-

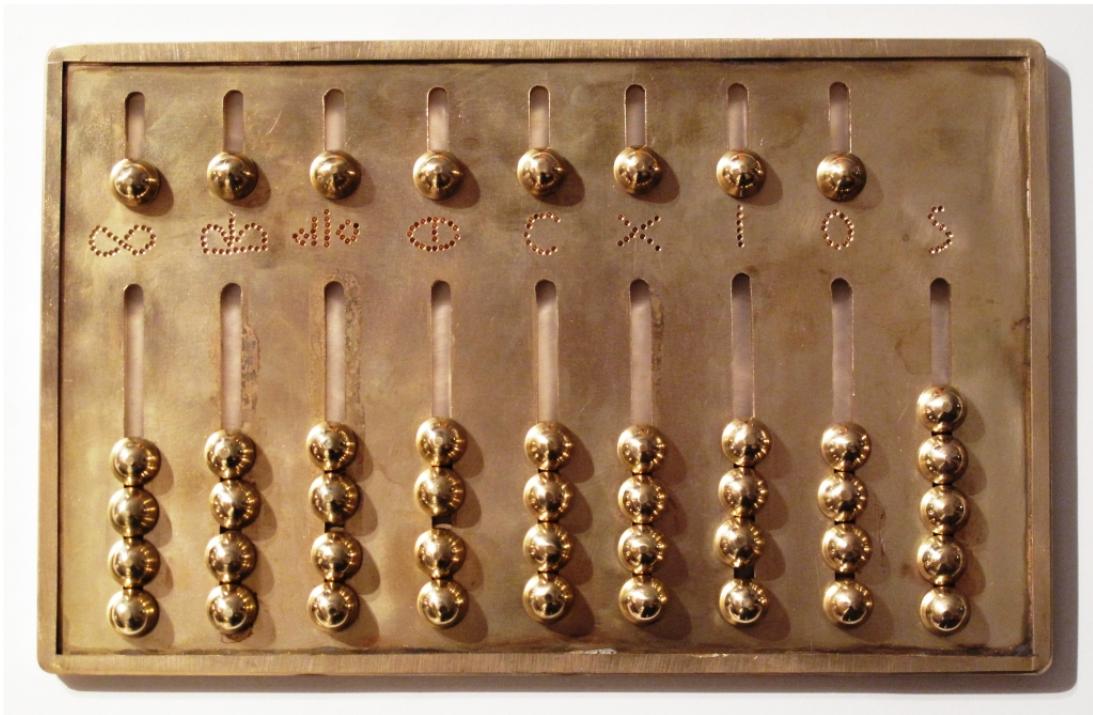


Figura 1.1: L'abaco nell'Antica Roma

mavano sostanze come cera e sabbia e si incidevano con lo stilo i numeri (così come facevano i Greci), oppure si usavano dei sassolini (detti in Latino *calculi*). Il funzionamento dell'abaco era come quello dell'attuale pallottoliere, con un comparto (quello più a destra) per le unità, uno per le decine, e così via.

L'uso dell'abaco continuò per tutto il Medioevo, ma si fece un notevole passo avanti coniando dei gettoni con sopra incisi dei simboli, ciascuno dei quali rappresentava un numero. Questi gettoni non contenevano alcuna rappresentazione per la cifra 0, la cui rappresentazione è stata introdotta intorno all'anno 820 dall'algebrista arabo **Al-Khuwarizmi**.

Verso gli inizi del 1600, il calcolo automatico vege un ulteriore passo in avanti grazie al matematico inglese *John Napier*, noto anche come *Nepero*. Questo matematico, poco prima di morire, pubblicò un libro nel quale espone un metodo per eseguire la moltiplicazione e la divisione con l'aiuto di alcune astucciole, note come *bastoncini di Nepero* (Figura 1.2), e illustrò anche come usarle per facilitare l'estrazione di radici quadrate e cubiche.



Figura 1.2: I bastoncini di Nepero (versione del XVIII secolo)

### 1.2.2 I precursori del calcolatore elettronico e dell'Informatica

Il XVII secolo fu veramente denso di innovazioni che possiamo considerare antenate dell'Informatica.

Nel 1642, il matematico, filosofo e letterato francese **Blaise Pascal** inventò la famosa *pascalina* (Figura 1.3), un congegno basato su ruote dentate capace di svolgere solo addizioni e sottrazioni, operando automaticamente i riporti. Pascal inventò questa macchina per aiutare il padre, esattore delle tasse, nei noiosi calcoli che la sua carica gli imponeva.

Per molti anni si ritenne la pascalina il primo esempio di calcolatore, quindi il progenitore di tutti i calcolatori elettronici attualmente esistenti, per cui si ritiene il 1642 come l'*anno zero* dell'Informatica. Solo recentemente si è scoperto che, prima di Pascal, ci furono altri esempi di macchina calcolatrice, alcuni anche più complessi della pascalina stessa.

Uno di questi esempi fu prodotto dal tedesco **Wilhelm Schickard** che mise a punto, nel 1623, un calcolatore meccanico noto come *orologio calcolatore*. Se ne ebbe notizia nel 1950, quando furono scoperte delle lettere di Schickard all'amico Keplero, nelle quali era descritto il progetto del meccanismo.

Un tentativo ancora precedente fu fatto da **Leonardo da Vinci**: nel 1490 sono state trovate alcune sue note che includevano la descrizione di una macchina somigliante a quella di Pascal.

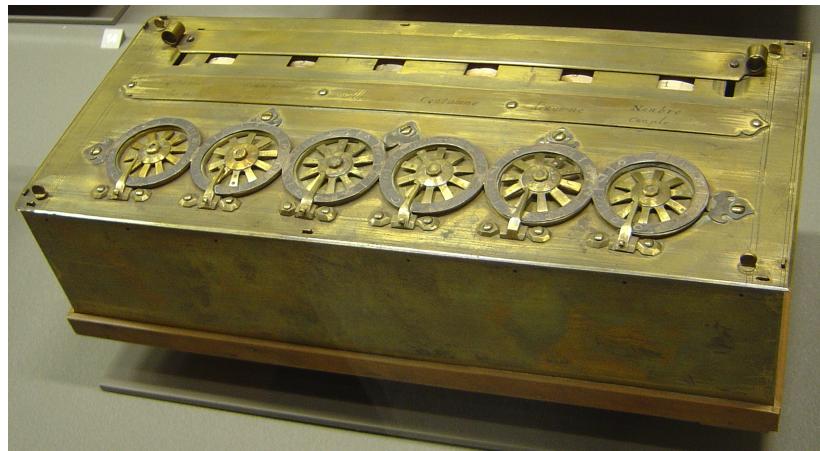


Figura 1.3: La pascalina

Nello stesso periodo in cui Pascal iniziò a diffondere la sua pascalina, il filosofo e matematico tedesco **Gottfried Wilhelm von Leibniz** progettò una macchina meccanica molto sofisticata, con meccanismi di precisione molto elevata per l'epoca. Leibniz, infatti, entrò fin da giovane in rapporti con i più illustri matematici dell'epoca, arrivando all'elaborazione di metodi semplici e generali per trattare i problemi infinitesimali.

Tra il 1664 e il 1694, la sua *calcolatrice di Leibniz* (Figura 1.4) fu completamente progettata, permettendo di svolgere tutte e quattro le operazioni aritmetiche elementari grazie all'ausilio di un pignone dentato. Diversi problemi meccanici,

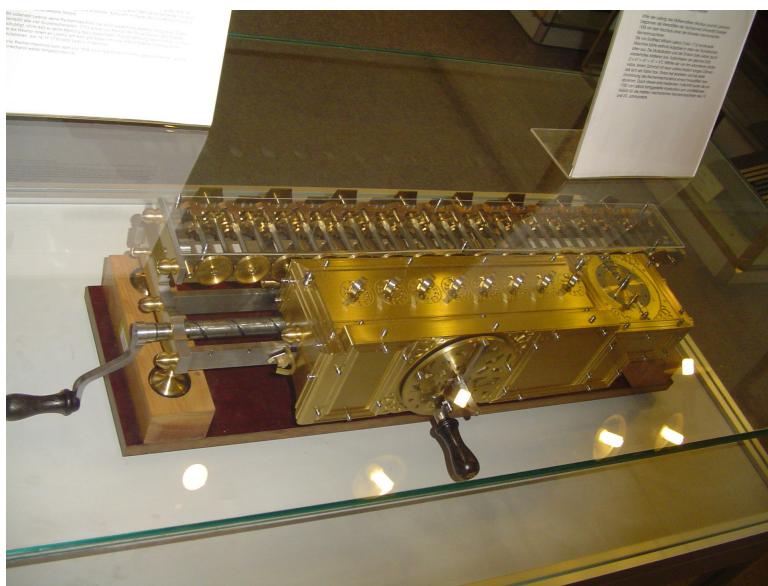


Figura 1.4: Un prototipo della calcolatrice di Leibniz

oltre a un difetto di progettazione nel meccanismo di riporto, impedirono alla macchina di funzionare in modo affidabile. Ciononostante, il progetto fu la base per i futuri progettisti di calcolatori.

Leibniz fu inoltre importante per la storia dell'Informatica perché fu colui che introdusse in Europa il *sistema di numerazione binario*, mediante il quale ogni numero viene espresso per mezzo di due soli simboli, 0 e 1, e che è alla base della memorizzazione delle informazioni nei calcolatori elettronici.

Lo sviluppo delle macchine automatizzate proseguì, ma una pietra miliare nella storia dell'Informatica fu quella del *telaio meccanico programmato*, progettato dall'inventore francese **Joseph Marie Jacquard** nel 1801. Jacquard applicò ai telai da tessitura meccanici, mossi da motori a vapore, un meccanismo a **schede perforate** che contenevano la sequenza esatta di operazioni da svolgere in maniera automatica.

Il XIX secolo provocò una nuova accelerazione nello sviluppo di ciò che poi divenne l'Informatica.

Tra il 1623 e il 1820, infatti, ci furono circa 25 fabbricanti di macchine calcolatrici. La maggior parte di queste furono il lavoro di un solo uomo, alcune funzionavano correttamente, ancor meno raggiunsero la linea di produzione.

Sul modello della macchina di Leibniz, il francese **Thomas de Colmar** costruì intorno al 1820 l'*aritmometro*, un apparecchio finalmente pratico, portatile, di facile uso e, soprattutto, correttamente funzionante. Essa fu la prima calcolatrice commercializzata con vero successo, tanto che ne furono venduti più di 1500 esemplari in trent'anni, fino ad ottenne una medaglia d'oro sia all'Esposizione Universale di Parigi del 1855 sia all'Esposizione Universale di Londra del 1862. Nel frattempo, il matematico inglese **Charles Babbage** progettò una macchina, denominata *macchina analitica* (Figura 1.5), che avrebbe consentito di eseguire calcoli di una certa complessità tramite guida di schede perforate. Il progetto,

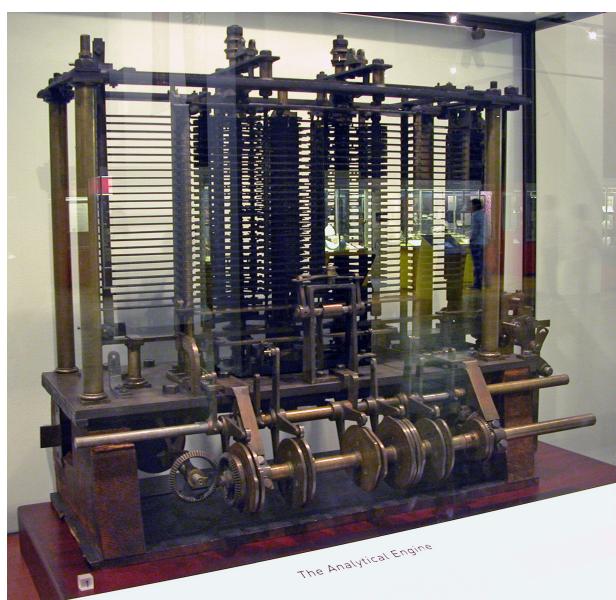


Figura 1.5: Parte dell'Analytical Engine di Babbage

completato nel 1833, aveva molte caratteristiche dei moderni calcolatori elettron-

nici, espresse naturalmente in forma meccanica.

La macchina fu progettata per trattare numeri di 50 cifre, ed aveva un metodo per far avanzare le cifre di riporto che permetteva di sommare contemporaneamente tutte le coppie di cifre corrispondenti nei due numeri a 50 cifre. Secondo le previsioni di Babbage, la sua macchina analitica avrebbe richiesto un secondo per l'addizione di due numeri a 50 cifre e un minuto per la loro moltiplicazione. L'importanza di questa macchina nella storia dell'Informatica sta nelle ipotesi di progettazione effettuate da Babbage. All'interno della macchina, infatti, si potevano individuare essenzialmente due parti: l'unità di calcolo dei procedimenti matematici e la memoria, che conteneva i dati su cui lavorare ed i risultati intermedi. La memoria era costituita da 1000 registri, ognuno dimensionato per la sua capienza massima (50 cifre), così da permettere di selezionare un numero dalla memoria, eseguire le operazioni e riportare il risultato in memoria.

Tutto il processo, come abbiamo detto, era controllato da una serie di schede perforate. La posizione dei fori in ogni scheda era letta da aghi che li attraversavano. Per Babbage, queste schede dovevano avere duplice utilità:

- indicare le operazioni da effettuare (*operations cards*);
- fornire l'indirizzo dell'operando in memoria (*variable cards*).

Purtroppo, dati i limiti tecnologici dell'epoca, la macchina rimase un progetto. Occorre ricordare, insieme a Charles Babbage, anche **Lady Ada Augusta, contessa di Lovelace**, sua amica che comprese l'importanza della macchina analitica e propose anche alcuni programmi per il suo funzionamento. Ciò significa che possiamo considerare la contessa come la prima progamatrice della storia dell'Informatica.

Un altro contributo importante allo sviluppo dell'Informatica così come la conosciamo ora fu dato dal matematico inglese **George Boole**, che nel 1847 definì quella che successivamente venne denominata *algebra di Boole* o *algebra booleana* e che oggi costituisce la base del funzionamento logico dei calcolatori elettronici.

In tutti i tentativi effettuati da Pascal fino a Babbage, l'elaborazione dei dati fu sempre concepita come una serie di operazioni aritmetiche, a volte anche onerose e complicate, che coinvolgevano una piccola quantità di dati. In pratica, tutti i progettisti si concentrarono più sulla quantità di calcoli che sulla quantità dei dati in ingresso.

Tutto ciò fu anche agevolato dallo sviluppo meccanico dell'epoca, che permise ottenere rendimenti sempre migliori delle loro macchine.

Sono da segnalare alcuni esempi di questi sviluppi migliorati. Nel 1872, **Frank Stephen Baldwin** elaborò, negli Stati Uniti, una specie di meccanismo interno. Nel 1889, il francese **Leon Bollée** elaborò una macchina che disponeva di una tabella di moltiplicazione interna.

Altri inventori ugualmente celebri furono gli americani **Door E. Felt** e **Williams S. Burroughs**, che contribuirono a costruire un vasto mercato per le calcolatrici. Felt adattò alle calcolatrici il principio della tastiera, che cominciava a essere

utilizzato nelle macchine per scrivere. La sua, chiamata il *contometro*, presentava infatti dei tasti che corrispondevano a cifre, là dove, tradizionalmente, c'erano sempre state delle ruote da far girare o dei cursori da far scivolare in scanalature.

Verso la fine dell'Ottocento, però, un problema stimolò l'interesse dei tecnici, che cercarono di progettare macchine che, al contrario delle calcolatrici, fossero predisposte per operare su moltissimi dati applicandoci un numero limitato di applicazioni aritmetiche.

Nel 1880, infatti, si tenne negli Stati Uniti il nono censimento della popolazione; trattando i dati manualmente, fu possibile ottenere i risultati definitivi solo dopo 7 anni. Quando, nel 1890, venne indetto il decimo censimento nazionale, molti espressero dei dubbi di ottenere i risultati definitivi entro la fine della decade, visto anche l'incremento demografico che gli Stati Uniti osservarono all'epoca. Fu così che l'ingegnere americano **Herman Hollerith** riprese il concetto delle schede perforate, lo adatto alle esigenze censitarie e suggerì di registrare su schede tutti i dati, in modo da poterli sottoporre ad una parziale elaborazione meccanica. Il metodo seguito consisteva nel perforare su ogni scheda i dati riguardanti un individuo o un nucleo familiare, in modo da poter poi creare una corrispondenza biunivoca tra la posizione delle perforazioni con i numeri decimali. La scheda veniva passata sopra una vaschetta contenente del mercurio e sulla sua superficie si faceva scendere una fila di aghi, i quali andarono ad immergersi nel mercurio. Poiché gli aghi erano percorsi da corrente elettrica, si stabiliva così un circuito che faceva avanzare un indicatore di tante unità quante corrispondevano alla perforazione.

### 1.2.3 I primi del Novecento: la prima generazione di calcolatori

Il Novecento fu il vero secolo in cui si ebbe la svolta per l'Informatica, poiché si definirono e si svilupparono tutte le teorie e le componenti degli attuali calcolatori elettronici.

Nel 1928, i matematici tedeschi **David Hilbert** e **Wilhelm Ackermann** pubblicarono il testo "*Principi della Logica Matematica*", con cui introdussero il formalismo noto ora come *logica del primo ordine*.

Grazie a questo formalismo, tutte le operazioni di decisione si poterono ridurre all'effetto della scelta tra due sole possibilità, cui vennero assegnate arbitrariamente le notazioni 0 e 1. Successivamente, questo formalismo fu rappresentato artificialmente tramite *circuiti logici*, ovvero circuiti di commutazione, muniti di uno o più terminali di ingresso e di uscita, ciascuno dei quali può assumere solo due potenziali cui sono stati associati i valori 0 e 1.

La presenza dell'energia elettrica e la nascita dell'Elettronica stimolò ulteriormente lo sviluppo del calcolo automatico.

A partire dal 1937, l'ingegnere tedesco **Konrad Zuse** iniziò a progettare macchine di calcolo usando prima memorie elettromagnetiche (lo Z1, uscito nel

1938, Figura 1.6) e poi relè elettromeccanici (lo Z2 nel 1939 e lo Z3 nel 1941). Le caratteristiche fondamentali di queste macchine erano:

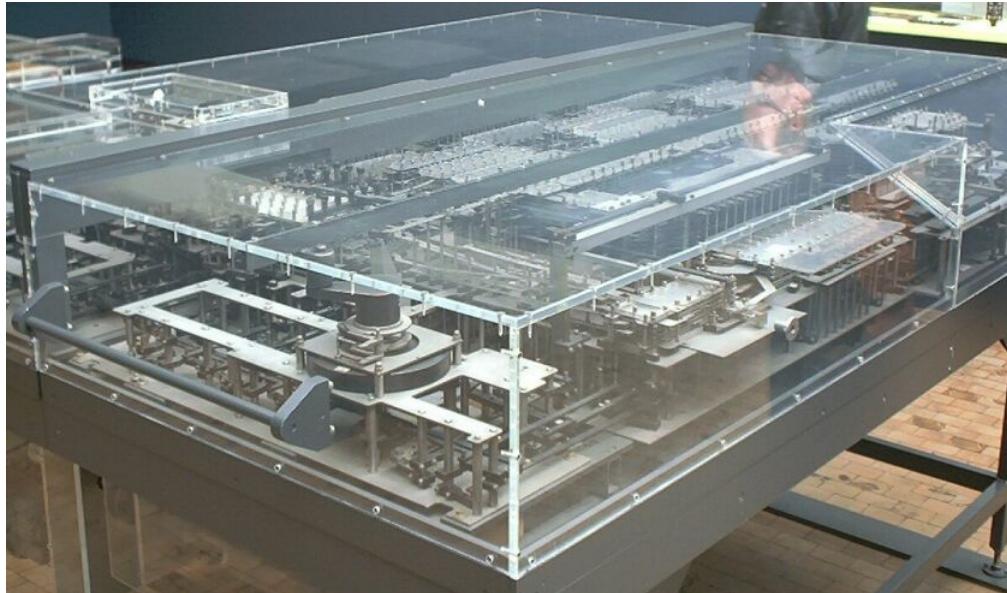


Figura 1.6: Riproduzione dello Z1

- l'uso del sistema numerico binario;
- l'uso di una parola di memoria composta di 22 bit;
- capacità di memoria di 64 parole;
- comandi forniti alla macchina per mezzo di un nastro perforato a 8 canali;
- visualizzazione del risultato mediante una serie di lampade;
- velocità di circa 3 secondi per moltiplicazioni, divisioni ed estrazioni di radici quadrate.

Le fondamenta teoriche dell'Informatica furono però fornite da due matematici: **Alan Turing** e **Kurt Gödel**.

Gödel, con i suoi *teoremi di incompletezza* pubblicati nel 1931, ha fornito le basi delle dimostrazioni logiche e delle *funzioni ricorsive*. Turing, invece, concentrando le sue ricerche sulla *computabilità* di un problema, arrivò a definire un modello teorico, noto come *macchina universale di Turing*, che costituirà il fondamento progettuale dei moderni calcolatori.

Un altro importante passo in avanti nello sviluppo dell'odierno calcolatore elettronico fu l'invenzione dei **tubi a vuoto** (o *valvole termoioniche*), che iniziarono a sostituire i relè nella costruzione delle macchine. A differenza di un relè, infatti, un tubo a vuoto aveva tempi di commutazione dello stato notevolmente inferiori rispetto a un relè, permettendo quindi di velocizzare i tempi di calcolo.

L'uso delle valvole termoioniche viene considerato l'inizio della *prima generazione dei calcolatori elettronici*.

Il più famoso calcolatore elettronico di quel periodo fu l'*ENIAC* (*Electronic Numerical Integrator And Computer*, Figura 1.7), realizzato nel 1946 presso la Pennsylvania University. L'impulso alla realizzazione dell'*ENIAC* arriva dalle vicende

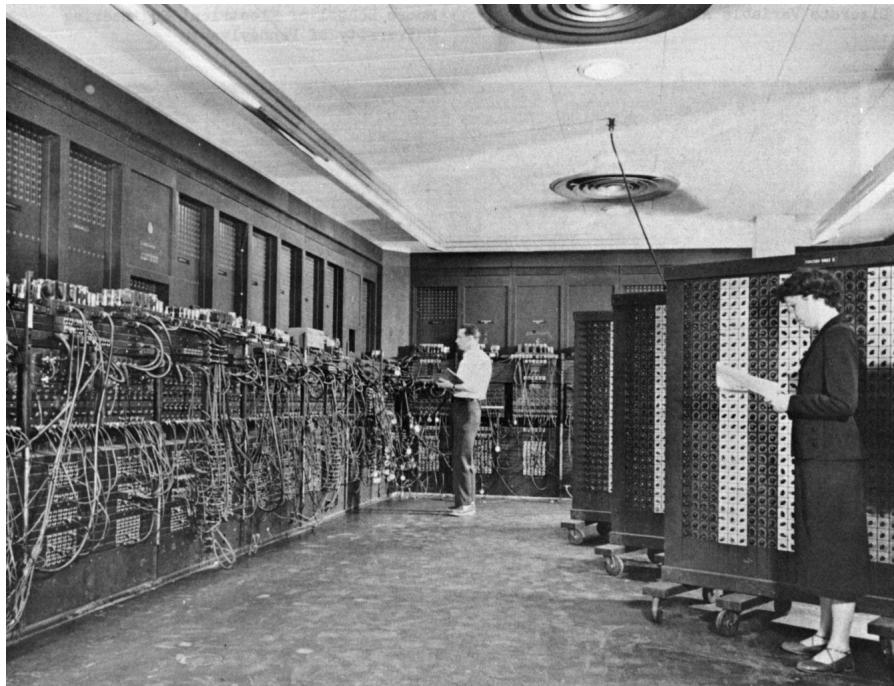


Figura 1.7: L'*ENIAC*

della Seconda Guerra Mondiale. Nel 1943, infatti, l'università aiutava lo sforzo bellico effettuando i calcoli per le tavole di tiro dell'artiglieria, usando un calcolatore elettronico. Tuttavia, i risultati non erano soddisfacenti, per cui si realizzò l'*ENIAC*.

Nonostante le sue dimensioni elevate (occupava una superficie di 180 m<sup>2</sup>, con un peso di 30 t e una potenza di 150 kW) e l'alto tasso di guasto (occorreva cambiare almeno tre valvole a settimana), riusciva a moltiplicare numeri a 10 cifre in 28 ms. Il vero difetto di questo calcolatore era l'impossibilità di programmarlo nel senso classico del termine: per cambiare l'elaborazione che il calcolatore doveva eseguire, era necessario modificare un gran numero di circuiti, richiedendo l'intervento di numerosi tecnici per alcuni giorni.

Per questo motivo, nel 1952 fu completato a Princeton un progetto iniziato 7 anni prima: quello dell'*EDVAC* (*Electronic Discrete Variable Automatic Computer*). Questo calcolatore era in grado di registrare nella propria memoria non solo i dati da elaborare, ma anche le istruzioni da eseguire, definendo così la base per la programmazione moderna.

Al progetto dell'*EDVAC* collaborò anche il matematico ungherese **John Von Neumann**, che definì la famosa *macchina di Von Neumann*. Grazie alla sua in-

tuzione di memorizzare il programma in modo da rendere più veloce l'elaborazione, il modello da lui ideato è ancora usato.

### 1.2.4 La seconda generazione e la nascita dei linguaggi di programmazione

Gli anni '50 videro l'introduzione di sistemi di media e grande potenza, appositamente progettati per smalire grandi volumi di lavoro amministrativo. Per migliorare le prestazioni vennero inventati i *i transistor* (Figura 1.8), che sostituirono le valvole. I transistor (crasi di *Transfer resistor*, cioè dispositivo di resisten-

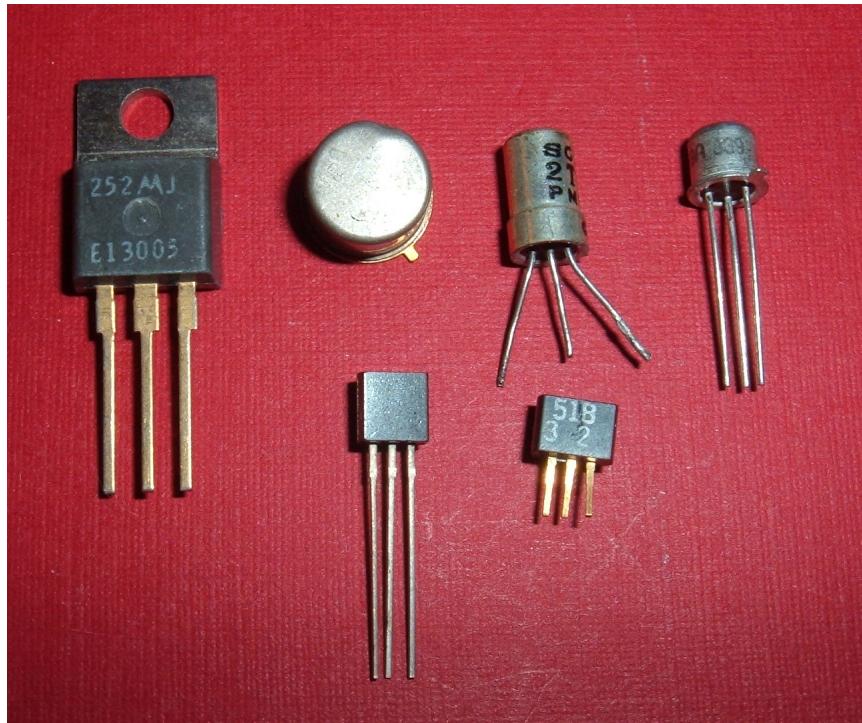


Figura 1.8: Varie tipologie di transistor

za) è un dispositivo a semiconduttore con tre elettrodi che amplifica correnti e tensioni elettriche senza uso di calore.

Ciò consentì la progettazione delle *memorie ad accesso diretto* (**RAM**, acronimo di *Random Access Memory*), la quale consentì di interagire con il calcolatore in modo tale da ottenere i dati durante l'esecuzione del programma, velocizzando l'esecuzione.

L'uso della RAM permise la costruzione nel 1951 dell' *UNIVAC I* (*Universal Automatic Computer I*), da parte degli stessi progettisti dell'*ENIAC*. L'*UNIVAC I* fu il primo calcolatore elettronico commerciale, visto che finora tutti i calcolatori elettronici furono ad uso militare o universitario.

Anche l'Italia si attivò nella progettazione dei calcolatori elettronici. Nel 1957, il Centro Studi dell'Università di Pisa realizza la macchina pilota, in

scala ridotta, della *Calcolatrice Elettronica Pisana* (CEP). Il progetto nacque dall'esigenza di disporre di una calcolatrice elettronica per la ricerca scientifica e dalla contemporanea difficoltà finanziaria di acquisto.

Alla progettazione partecipò anche la Olivetti, che usò le idee per produrre poi nel 1959 il primo calcolatore elettronico a transistor completamente italiano: l'*ELEA 9000*.

Nel frattempo, l'uso commerciale dei calcolatori elettronici prese piede negli Stati Uniti. Una piccola compagnia, la **DEC** (*Digital Equipment Corporation*), decise di investire tutto in questa tecnologia, commercializzando nel 1960 il *PDP-1* (Figura 1.9). Il PDP-1 fu tecnicamente un successo: aveva un monitor a tubo



Figura 1.9: IL PDP-1

catodico integrato, poteva trovare posto in una piccola stanza e forniva una rispettabile potenza di calcolo per il suo costo di "soli" 120.000 dollari.

Contemporaneamente allo sviluppo tecnologico ci fu uno sviluppo dal punto di vista software.

Fino alla fine degli anni '50, infatti, la programmazione era sostanzialmente la scrittura di lunghe sequenze di cifre binarie in *linguaggio macchina*. L'utilizzo di questi linguaggi non era molto funzionale, per cui si svilupparono linguaggi di programmazione più semplici.

Fra tutti quelli che furono creati, quello che esercitò una influenza maggiore fu il **FORTRAN** (*Formula Translation*), sviluppato tra il 1954 e il 1957 da **John Backus** e dai suoi collaboratori presso IBM. Questo linguaggio, nato per eseguire calcoli scientifici e numerici, all'inizio fu poco usato perché si riteneva poco ottimizzato; quando Backus e i suoi collaboratori riuscirono a dimostrare che la qualità del codice era pari a quella dei programmi codificati in linguaggio macchina, allora il FORTRAN prese piede.

Nello stesso periodo, fu sviluppato il **Flow-Matic** per elaborare problematiche commerciali. Il Flow-Matic era meno evoluto del FORTRAN, ma molti dei suoi concetti furono di ispirazione per la realizzazione del **COBOL** (*Common Business Oriented Language*).

Un altro linguaggio introdotto verso la fine degli anni '50 fu l'**ALGOL** (*Algorithmic Language*), basato sulla sintassi del FORTRAN. L'ALGOL è un linguaggio allo stesso tempo leggibile e pratico, che funse da presenza per l'ideazione di linguaggi di programmazione successivi come il **PASCAL**.

Contemporaneamente allo sviluppo dei linguaggi di programmazione, ci si rese conto che per usare al meglio un calcolatore elettronico serviva un programma di base che fornisse agli utenti vari servizi, astraendo le possibilità dei calcolatori. Nacquero dunque i primi *sistemi operativi*, fornendo pochi semplici servizi, come la gestione dell'input/output.

### 1.2.5 La terza generazione dei calcolatori

Nel 1958 l'invenzione dei *circuiti integrati* da parte di **Robert Noyce** permise di comprimere su un'unica scheda, molto piccola, il circuito del transistor. In questo modo fu possibile costruire calcolatori più piccoli, più veloci e più economici rispetto ai suoi predecessori.

Il primo modello commerciale ad utilizzare i circuiti integrati fu il *System/360* (Figura 1.10), prodotto da IBM. Il System/360 era una famiglia di macchine dotate dello stesso linguaggio assemblativo, ma di dimensione e potenza crescenti.



Figura 1.10: Un modello di System/360

Un altro modello commerciale che usò i circuiti integrali fu il *PDP-11* della **DEC**, un successore a 16 bit dei precedenti modelli prodotti.

È possibile considerare i calcolatori della serie PDP-11 come una sorta di "fratello

"minore" del System/360, poiché entrambi avevano i registri orientati alla parola e la memoria orientata al byte.

Contemporaneamente allo sviluppo hardware, anche i sistemi operativi subirono una evoluzione, grazie all'introduzione della *multiprogrammazione*. La multiprogrammazione è una tecnica che permette di avere più programmi in memoria nello stesso momento, così da ottimizzare il tempo eseguendo calcoli di un programma mentre un altro è in attesa di operazioni di input/output.

### 1.2.6 La quarta generazione: i *personal computer*

Negli anni '80 la tecnologia *VLSI* (*Very Large Scale Integration*, integrazione a larghissima scala) permise di inserire in un unico circuito prima decine di migliaia, poi centinaia di migliaia e poi milioni di transistor. Nacquero così i primi *microprocessori*, molto rudimentali rispetto a quelli attuali.

I primi calcolatori a usare la nuova tecnologia furono gli *APPLE* e gli *APPLE II* (Figura 1.11), ideati da **Steve Jobs**. Questi modelli rappresentarono i primi *personal computer*, cioè calcolatori personali, a diffusione di massa, il cui prezzo era contenuto e facili da usare per il suo speciale sistema operativo.



Figura 1.11: L'APPLE II

Successivamente, IBM immise sul mercato il suo personal computer, il *PC-IBM* (Figura 1.12). Questo calcolatore segnò la svolta nel mercato dei personal

computer: il sistema operativo installato, **MS-DOS**, ha creato infatti i presupposti per l'adozione di uno standard usato ancora oggi.



Figura 1.12: L'IBM 5150, un esempio di PC-IBM



# Capitolo 2

## Architettura di un calcolatore

Come abbiamo descritto nel Capitolo 1, l'Informatica non è strettamente legata all'uso dei calcolatori elettronici. Eppure, senza un calcolatore elettronico ci sembra assurdo parlare di Informatica.

Ma effettivamente in cosa consiste un calcolatore? Come funziona?

In questo capitolo cercheremo di rispondere a queste domande, approfondendo l'architettura di un calcolatore e del programma principale che lo fa funzionare, il sistema operativo. In aggiunta, vedremo anche come le informazioni vengono rappresentate all'interno di un calcolatore, così da capire meglio come funziona.

### 2.1 Hardware e software

L'esempio più semplice di calcolatore elettronico che vediamo comunemente è la calcolatrice.

La calcolatrice è di dimensioni e di prezzo molto ridotto; tutte le tipologie di calcolatrici in commercio riescono ad eseguire le quattro operazioni fondamentali, quasi tutte eseguono l'estrazione di radice quadrata e il calcolo della percentuale e hanno una memoria per conservare i risultati parziali.

Possiamo distinguere una parte visibile, formata dal display e dalla tastiera, e da una parte interna, formata dal dispositivo che controlla il funzionamento, da quello che effettua i calcoli, una casella che contiene un operando e una casella che contiene il numero che compare sul display.

Le calcolatrici più comuni non sono dispositivi programmabili, ma esistono dei dispositivi particolari che sono *programmabili*. Una **macchina programmabile** è un dispositivo che:

- può essere utilizzato per vari tipologie di problemi, non solo per calcoli;
- è in grado di comprendere ed eseguire una serie di operazioni indicate dall'esterno.

**Definizione 2.1** (Calcolatore elettronico). Un **calcolatore** è una macchina programmabile, cioè in grado di eseguire programmi.

**Definizione 2.2** (Programma). Un **programma** (o **applicazione**) consente di far svolgere al calcolatore varie operazioni, ciascuna delle quali permette agli utenti di perseguire lo scopo voluto.

Esistono numerose tipologie di applicazioni, che vanno dai giochi alla gestione di immagini e documenti e alla gestione di sistemi informativi. Il fatto che su un medesimo calcolatore si possano eseguire applicazioni diverse rende il calcolatore una macchina che può essere usata per risolvere problemi molto diversi tra loro, purché l'utente fornisca al calcolatore istruzioni dettagliate su come il problema possa essere risolto.

Ma per comprendere a pieno come funziona un calcolatore elettronico, bisogna analizzare il suo funzionamento interno, cioè la sua *architettura*. Facendo ciò, possiamo notare che un calcolatore elettronico è un **sistema**, cioè un insieme complesso di elementi, detti *componenti*, di natura differente che interagiscono tra loro in modo dinamico per ottenere un certo comportamento.

Analizzare i vari componenti serve proprio per definire il funzionamento dell'intero sistema. È chiaro ora perché si usa il termine **sistema di elaborazione** per indicare il calcolatore elettronico, proprio perché il lavoro con il calcolatore si avvale non di un solo oggetto ma di un insieme organizzato di apparecchiature e di programmi che interagiscono tra loro per elaborare automaticamente le informazioni. Allo stesso modo, il termine *sistema operativo* indica l'insieme dei programmi di base che fanno funzionare le varie componenti del calcolatore.

Nel corso di questo capitolo, ciò che faremo è:

- l'individuazione di ciascun componente;
- la comprensione del funzionamento di ciascun componente;
- la comprensione dei meccanismi di interazione tra componenti.

Iniziando l'analisi architettonica di un calcolatore, è possibile individuare due macro-componenti di alto livello:

- l'**hardware**, cioè la struttura fisica del calcolatore, costituita da componenti elettroniche ed elettromeccaniche;
- il **software**, cioè l'insieme di programmi (di base o applicativi) che consentono all'hardware di svolgere compiti utili.

In particolare, si parla di:

- *software di base* per indicare l'insieme di quei programmi che mostrano all'utente il calcolatore come una macchina virtuale più semplice da gestire e programmare rispetto all'hardware usato realmente;
- *software applicativo* per indicare l'insieme di quei programmi che mostrano il calcolatore come una macchina virtuale adatta per essere usata nella risoluzione di problemi.

Questi due macro-componenti sono organizzati a livelli, ciascuno dei quali corrisponde ad una macchina dotata di un proprio insieme di funzionalità ed un proprio linguaggio. In aggiunta, ciascun livello fornisce un linguaggio più semplice da usare rispetto a quello del livello sottostante.

## 2.2 La Macchina di Von Neumann

Dal punto di vista hardware, l'architettura di un calcolatore hardware è molto complessa. Per questo motivo venne introdotta la **macchina di Von Neumann** (Figura 2.1), un modello semplificato dei calcolatori, composto da quattro componenti:

- l'**unità centrale di elaborazione**, detta anche **processore** o **central processing unit** (CPU), che segue le istruzioni per l'elaborazione dei dati e svolge funzioni di controllo delle altre componenti;
- la **memoria centrale**, che memorizza e fornisce l'accesso a dati e programmi;
- le **interfacce di ingresso e uscita**, che permettono di collegare periferiche esterne al calcolatore in modo da permettere lo scambio di dati tra il calcolatore e l'utente;
- il **bus**, che consente di trasferire dati e informazioni di controllo tra le varie componenti.

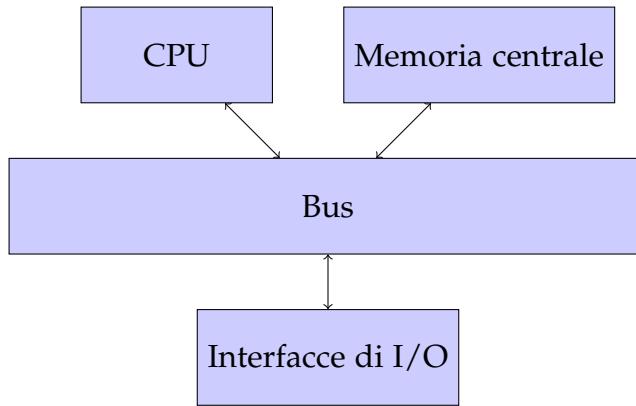


Figura 2.1: Macchina di Von Neumann

Analizziamo le varie componenti di questa macchina.

### 2.2.1 La memoria

La *memoria* è la componente del calcolatore in cui vengono immagazzinati dati e programmi per poi accederci, ovvero possono essere effettuate due operazioni:

- *scrittura*, cioè la memorizzazione di un dato;

- *lettura*, cioè l'accesso del dato memorizzato.

Dal punto di vista fisico, la memoria è un dispositivo elettronico costituito da uno o più *chip*, piastrine di silicio incapsulate in materiale plastico e fornite di contatti metallici (i *piedini* o *pin*) che ne consentono l'inserimenti in circuiti più estesi, le *schede*.

Il funzionamento dei calcolatori elettronici, come abbiamo detto, si basa sul sistema di numerazione binario per il trattamento delle informazioni. Possiamo dunque dire che l'unità di informazione è il **bit** (dall'inglese *binary digit*). Il bit, purtroppo, non è sufficiente per codificare alcuna quantità di informazioni, per cui si usa un suo multiplo, il **byte** (pari a 8 bit), che permette di codificare fino a  $2^8$  diverse informazioni.

Nella macchina di Von Neumann veniva descritto un solo tipo di memoria, interno al calcolatore e quindi acceduta direttamente dal processore. Questo tipo di memoria viene denominata ora come **memoria centrale**.

Esiste però anche un secondo tipo di memoria, esterno alla macchina, il cui accesso passa attraverso interfacce: la **memoria secondaria**.

### Memoria centrale

Dal punto di vista logico, la *memoria centrale* è composta da *celle* (o *locazioni*), ognuna delle quali è in grado di memorizzare una *parola*, cioè una sequenza di bit di lunghezza fissata. Una cella è identificata da un *indirizzo* e il suo contenuto prende il nome di *valore*.

Gli indirizzi di memoria sono gli oggetti sui quali un calcolatore elettronico deve lavorare: se per esempio un programma richiede di recuperare un dato memorizzato, deve specificare l'indirizzo in cui è contenuto, e per fare questo deve ricordare anche l'indirizzo che gli serve registrandolo in una locazione di memoria particolare. È dunque comodo rappresentare gli indirizzi tramite il sistema binario, identificando con *spazio di indirizzamento* il numero di bit usati dal processore per identificare gli indirizzi.

Ma come facciamo a distinguere una memoria centrale rispetto ad un'altra? Esistono alcune caratteristiche di base.

- La dimensione della parola.
- La **capacità**, cioè il numero di bit che possono essere immagazzinati rappresentati come byte e suoi multipli
- La **velocità di accesso**.
- La **volatilità**, cioè la capacità di immagazzinare i dati anche allo spegnimento del calcolatore.

**Capacità** Dal punto di vista della capacità, possiamo notare che le informazioni sono tutte rappresentate usando il sistema di numerazione binario. Per tanto l'utilizzo delle unità di misura derivanti dal byte per rappresentare la capacità di una memoria centrale è logico.

Nella Tabella 2.1 vediamo alcuni multipli del byte e i loro valori.

Unità di misura	Simbolo	Valore
Kilobyte	kB	1024 B
Megabyte	MB	1024 kB
Gigabyte	GB	1024 MB
Terabyte	TB	1024 GB
Petabyte	PB	1024 TB

Tabella 2.1: Multipli del byte

**Velocità** La velocità del dispositivo è legata al tempo che intercorre tra la richiesta di accedere ad una certa parola di memoria e l'istante in cui l'operazione è eseguita. Questo tempo è, in genere, dell'ordine dei nanosecondi e viene identificato come frequenza di scambio di informazioni con il bus (in genere, multipli di 33 MHz).

**Volatilità** Volendo valutare la volatilità delle informazioni all'interno della memoria centrale, possiamo identificarne due tipologie:

- *Random Access Memory* (RAM), che può essere sia letta sia scritta ma è volatile;
- *Read Only Memory* (ROM) , che può essere solo letta ma è persistente.

Le memorie ROM sono più economiche, in quanto realizzate con circuiti più semplici. Per questo motivo vengono ampiamente usate in quegli scenari ove non serve modificare il contenuto di memoria, ad esempio la fase di accensione e avvio del sistema operativo (*bootstrap*). Inoltre, le memorie ROM contengono programmi in linguaggio macchina eseguiti per la gestione standard di dispositivi di input/output, o tabelle matematiche di frequente consultazione.

La RAM, invece, è più costosa perché deve permettere la modifica. Per fare ciò, è composta da oggetti elettronici *bistabili*, il cui funzionamento è subordinato alla presenza di tensione elettrica (senza tensione elettrica, il contenuto viene perduto).

### La memoria secondaria

La *memoria secondaria*, detta anche *memoria di massa*, è una memoria persistente, la cui capacità è notevolmente maggiore della memoria centrale. Per la sua

!htb

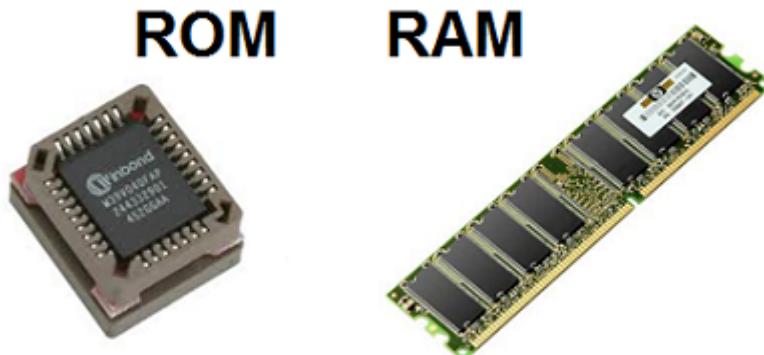


Figura 2.2: RAM e ROM

natura persistente, il materiale usato per la memorizzazione ha costi di produzione notevolmente minori rispetto a quelli della memoria centrale. In aggiunta, la velocità di accesso ai dati di una memoria secondaria è minore di quella della memoria centrale.

Data la sua alta capacità, si tende ad usare la memoria secondaria per la memorizzazione dei dati e dei programmi non in esecuzione, spostando poi sia i dati sia i programmi nella memoria centrale non appena vengono eseguiti.

Le memorie secondarie sono caratterizzate da alcuni parametri fondamentali, che ne descrivono le caratteristiche:

- il **tempo di accesso**, espresso tramite sottomultipli del secondo, che indica il tempo richiesto affinché il calcolatore possa ritrovare i dati salvati per poterli poi elaborare in memoria centrale;
- la **capacità**, ovvero la quantità di informazioni (espresse in multipli del byte) che la memoria secondaria può contenere;
- la **velocità di trasferimento dei dati**, ovvero la rapidità con la quale i dati vengono trasferiti dalla memoria secondaria alla memoria centrale e si misura in multipli del byte per secondo (kB/s, MB/s, ecc.).

Il tipo di memoria secondaria più diffusa è composta da un **disco magnetico** (Figura 2.3), ovvero uno o più piatti di alluminio rotanti, ricoperti da materiale magnetico, che vengono letti da **testine**; le testine cambiano lo stato di polarizzazione del materiale magnetico per memorizzare i dati.

Ciascun piatto è composto da due superfici (dette *facce*) o *piatto*, ciascuna delle quali è suddivisa in *tracce* (circolari) e *settori* (a spicchio), come si può vedere nella Figura 2.4. L'unità logica di memorizzazione è il settore, identificato tramite la sua posizione radiale (*cilindro*), la faccia e il numero di settore, e la sua capacità



Figura 2.3: Un esempio di disco magnetico

è generalmente di 512 B.

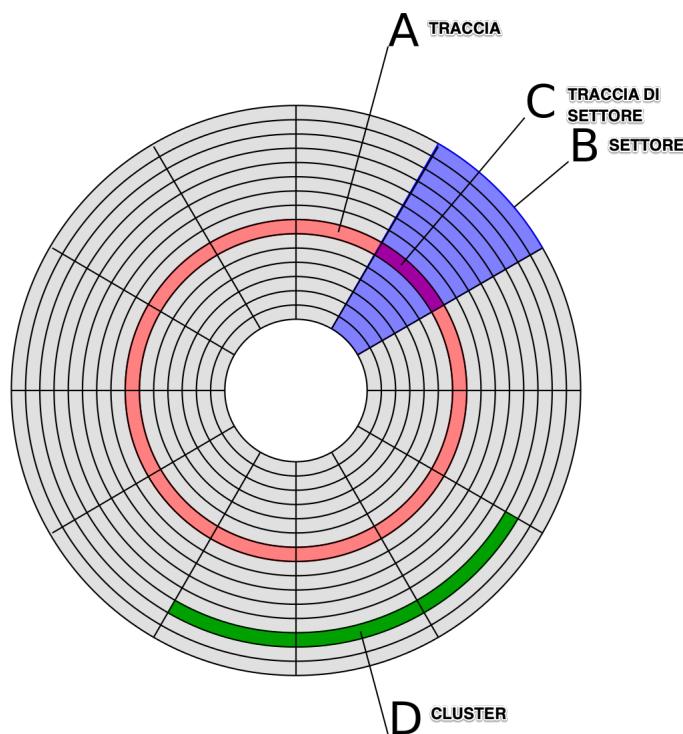


Figura 2.4: Struttura delle superfici di una faccia

Negli anni sono stati inventati ulteriori dispositivi di memoria secondaria, per la distribuzione delle informazioni o garantire l'aumento della capacità dei calcolatori.

- I **CD-ROM** (*Compact Disk - Read Only Memory*) sono dei dischi di grandissi-

ma capacità basati su un sistema di lettura di tipo ottico. I CD-ROM possono contenere fino a 700 MB, per cui risulta adatto per conservare documenti multimediali consultabili facilmente e in tempi brevi.

- I **DVD** (*Digital Versatile Disc*) sono anch'essi dischi di grandissima capacità, letti tramite dispositivi ottici, ma la loro capacità è maggiore rispetto ai CD-ROM (tipicamente 4,7 GB).

## 2.2.2 Unità centrale di elaborazione

L'**unità centrale di elaborazione** è la componente responsabile dell'esecuzione di un programma, ovvero esegue materialmente l'insieme di operazioni elementari (logiche, aritmetiche e di trasferimento sui dati), controllando che queste operazioni vengano eseguite in maniera ordinata.

Questa componente si può considerare costituita da due sotto-componenti:

- l'**unità logico-aritmetica** (*Arithmetic Logic Unit, ALU*) , in grado di eseguire le operazioni richieste in maniera efficiente;
- l'**unità di controllo** (*Control Unit, CU*) , che coordina le varie componenti stabilendo anche quali operazioni debbano essere eseguite.

La coordinazione effettuata dall'unità di controllo avviene in maniera sincrona con un orologio interno del calcolatore, denominato *clock*: in pratica, ad ogni scatto del clock (denominato *tick*) viene inviato un segnale. La frequenza di questi *tick* fornisce, perciò, un'importante indicazione sulla velocità cui opera l'unità centrale.

Generalmente, la frequenza dei *tick* viene indicata usando multipli dell'hertz, l'unità di misura della frequenza in Fisica (MHz o GHz). Esiste però un'altra unità di misura della frequenza dei *tick*, che rappresenta il numero di operazioni che possono essere effettuate in un secondo: il **MIPS** (*Millions Instructions Per Second*).

Occorre tenere presente che più il valore della frequenza (in MHz, GHz o MIPS) è elevato, maggiore è la probabilità che insorgano nei circuiti fenomeni di riscaldamento ed elettromagnetici che interferiscono con il normale funzionamento di essi. Per questo motivo, la progettazione di processori sempre più veloci ha richiesto notevoli sforzi per ovviare a questi problemi.

Dal punto di vista fisico, la CPU è realizzata su un chip di silicio, che prende il nome di *microprocessore*. Ogni microprocessore si identifica in base alle seguenti caratteristiche:

- il repertorio di istruzioni;
- la velocità di esecuzione;
- l'ampiezza del bus;
- il co-processore, specializzato per certe operazioni;

- la *cache* (memoria veloce e locale).

La CPU, inoltre, deve contenere elementi di memoria e dispositivi che permettono di eseguire le operazioni. Abbiamo dunque:

- memorie di tipo ROM, che contengono le informazioni permanenti del dispositivo;
- **registri**, ovvero celle di memoria locali nelle quali si può leggere e scrivere dato e operazioni;
- singoli bistabili isolati, nei quali risiedono i cosiddetti *bit di controllo*.

Tra i principali registri possiamo individuare:

- **registro contatore delle istruzioni** (*program counter, PC*), che contiene l'indirizzo della prossima istruzione da eseguire;
- **registro delle istruzioni** (*instruction register, IR*), che contiene l'istruzione codificata da eseguire;
- **registro di indirizzamento della memoria** (*memory address register, MAR*), che contiene l'indirizzo della cella di memoria che deve essere acceduta;
- **registro dati di memoria** (*memory data register, MDR*), che contiene il dato letto o da scrivere;
- **parola di stato del processore** (*Processor State Word, PSW*), che contiene informazioni, opportunamente codificate, circa l'esito dell'ultima istruzione eseguita.

Finora abbiamo definito un processore dal punto di vista fisico e logico, ma non abbiamo definito il suo funzionamento interno. Per eseguire un programma, la CPU preleva le istruzioni dalla memoria centrale una alla volta, secondo un ciclo noto come *fetch-decode-execute* (Figura 2.5).

- Nella fase di **fetch**, il processore legge dalla memoria centrale l'istruzione successiva.
- Nella fase di **decode**, il processore determina quale sia il tipo di istruzione da eseguire.
- Nella fase di **execute**, l'unità logico-aritmetica svolge le azioni necessarie per eseguire l'istruzione.

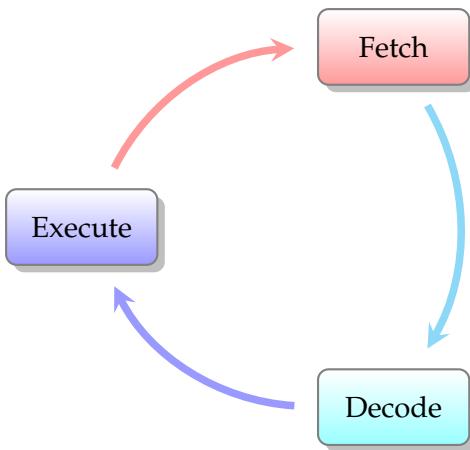


Figura 2.5: Il ciclo *fetch-decode-execute*

### 2.2.3 Le interfacce di input/output

Le interfacce di input/output, indicate comunemente come *unità di I/O*, consentono l’acquisizione dall’esterno dei dati che devono essere elaborati dal calcolatore e la comunicazione verso l’esterno dei risultati dell’elaborazione.

Nei primi calcolatori, il supporto più diffuso per i dati di input era, come abbiamo detto, la *scheda perforata*, poiché non erano disponibili strumenti più avanzati per l’interazione tra utente e il calcolatore.

Attualmente, le interfacce di input più usate sono:

- la *tastiera*, del tutto simile a quella di una macchina da scrivere, che permette l’immissione di caratteri nel calcolatore;
- il *mouse*, che permette di muovere un puntatore sul video del calcolatore;
- lo *scanner*, che consente di acquisire immagini o di leggere parti scritte di documenti trasformandoli in segnali digitali immessi nel calcolatore.

Le interfacce di output più usate sono:

- il *video*, o *monitor*, che è l’unità di output standard, riportando i messaggi di risposta e di errore del calcolatore all’utente;
- la *stampante*, che riproduce caratteri su carta o supporti.

### 2.2.4 Il bus

La macchina di Von Neumann, come abbiamo detto, ha una caratteristica molto utile: è *modulare*, ovvero le varie componenti possono lavorare anche in maniera isolata. Per collaborare tra loro, le varie componenti inviano e ricevono dati tramite un canale di comunicazione noto come *bus*.

È possibile distinguere due tipologie di bus:

- il *bus di I/O*, che è un canale di comunicazione specializzato per eseguire in maniera autonoma trasferimenti di dati tra le interfacce di I/O e la memoria centrale;
- il *bus locale*, usato per trasferire informazioni dalle e verso le interfacce di I/O a velocità elevate.

## 2.3 La rappresentazione dell'informazione

Rappresentare una informazione all'interno di un calcolatore è una operazione vincolata a come vengono memorizzati i dati e alla capacità a disposizione. Questi due vincoli sono indipendenti.

Infatti, anche se il calcolatore avesse capacità infinita, data la natura elettronica del calcolatore dovremmo sempre studiare come rappresentare le informazioni (quindi il codice binario). D'altra parte, se il calcolatore fosse capace di memorizzare le informazioni nel formato naturale, la capacità della memoria sarebbe comunque finita.

Analizziamo meglio i vari sistemi di numerazione e di rappresentazione possibili.

### 2.3.1 I sistemi di numerazione

Per rappresentare informazioni numeriche, da sempre l'essere umano ha usato dei **sistemi di numerazione**, ovvero insiemi di oggetti e regole organizzati per rappresentare le grandezze numeriche.

Il sistema di numerazione più usato è il *sistema di numerazione decimale*, il cui nome deriva dal fatto che gli oggetti usati per rappresentare le grandezze numeriche sono dieci cifre numeriche, il cui insieme prende il nome di *base* ( $b = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ). Tra le regole proprie del sistema decimale troviamo la *posizionalità*, ovvero ogni oggetto rappresenta un valore diverso a seconda della posizione occupata nella scrittura.

$$123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

Questo sistema di numerazione proviene molto probabilmente dall'India, ma solo con l'invasione delle popolazioni arabe in Europa fu iniziato a essere usato anche in Occidente. Prima di questo sistema di numerazione, infatti, veniva usato ancora il *sistema di numerazione romano*, dove al posto delle cifre erano usate delle lettere (I per indicare uno, V per indicare cinque, X per indicare dieci, e così via).

Il fatto che la regola di posizionalità abbia preso piede nell'uso comune è dovuto al fatto dell'esistenza di algoritmi di risoluzione semplici per le operazioni aritmetiche basilari.

Il sistema di numerazione decimale non è l'unico possibile. Nell'ambito informatico, infatti, vengono usati spesso, per motivi diversi, altri sistemi di numerazione, quale il *sistema di numerazione binario*, il *sistema di numerazione ottale* e il *sistema di numerazione esadecimale*.

Il *sistema di numerazione binario* è un sistema di numerazione posizionale in base 2, ovvero  $b = \{0, 1\}$ . È posizionale perché la cifra di posizione  $n$  da destra si considera moltiplicata per  $2^n$ .

Il *sistema di numerazione ottale* è un sistema di numerazione posizionale in base 8, ovvero  $b = \{0, 1, 2, 3, 4, 5, 6, 7\}$ . È posizionale perché la cifra di posizione  $n$  da destra si considera moltiplicata per  $8^n$ .

Il *sistema di numerazione esadecimale* è un sistema di numerazione posizionale in base 16, ovvero  $b = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$ . Come si vede, per ovviare al problema dell'assenza di sedici cifre, per i numeri da 10 a 15 vengono usate le lettere da A a F. Questo sistema è anch'esso posizionale perché la cifra di posizione  $n$  da destra si considera moltiplicata per  $16^n$ .

### 2.3.2 La rappresentazione binaria

Come abbiamo detto, si è scelto di rappresentare i dati nei calcolatori elettronici tramite bit e byte, per via della loro struttura fisica. Siccome questo intervallo di valori è molto esiguo, mettendo in sequenza più byte è possibile rappresentare molte più informazioni.

Per codificare una informazione in codice binario si può procedere secondo due approcci:

- un approccio *tabellare*, applicabile quando l'insieme dei valori che può assumere l'informazione è finito;
- un approccio che implica una *conversione di base*, applicabile quando il dato è numerico ed è decimale.

#### Approccio tabellare

L'approccio tabellare sfrutta il concetto matematico delle *disposizioni con ripetizione*<sup>1</sup> che si possono avere con  $m$  bit ( $2^m$ ) ed è utile per rappresentare tutti quei valori il cui dominio è finito, come i caratteri.

Concentrandosi sui caratteri alfanumerici, storicamente la codifica tabellare più vecchia è quella nota come **ASCII** (acronimo di *American Standard Code for Information Interchange*), pubblicato nel 1968. Lo standard ASCII usa 7 bit per usare tutti i caratteri alfanumerici latini, i principali caratteri di punteggiatura e

---

<sup>1</sup>Nel calcolo combinatorio, dati due numeri interi non negativi  $n$  e  $k$ , si definisce *disposizione di  $n$  elementi a  $k$  a  $k$*  ogni sottoinsieme ordinato di  $k$  elementi estratti da un insieme di  $n$  elementi tale che i sottoinsiemi differiscano almeno in un elemento oppure, in presenza degli stessi elementi, nel modo in cui sono ordinati.

alcuni caratteri di controllo.

Alla specifica originaria è seguita una seconda specifica, nota come *Extended ASCII*, che usa un byte completo per rappresentare gli stessi caratteri.

Questa codifica divenne col tempo molto limitata, proprio per la loro rappresentazione dei soli caratteri latini. Per questo motivo, a partire dal 1989 prese piede la codifica **Unicode**, la quale usa due byte per rappresentare i caratteri alfanumerici latini, alfabeti particolari come il cirillico, gli ideogrammi, ecc.

Dal punto di vista pratico, è previsto l'uso di codifiche con unità da 1, 2 o 4 B, descritte rispettivamente come *UTF-8*, *UTF-16* e *UTF-32*.

Anche per i valori booleani (**TRUE** o **FALSE**) si usa una codifica tabellare, usando il valore 0 per **FALSE** e tutti gli altri valori per **TRUE**.

### Il cambio di base

Siccome tutti i sistemi di numerazione elencati finora sono posizionali, per trasformare un numero da una generica base  $b$  alla base 10, si sfrutta la posizione della cifra. Viceversa, per trasformare un numero naturale dalla base 10 alla base  $b$  si sfrutta l'*Algoritmo di Euclide*, dividendo il numero per la base  $b$ ; il numero convertito si ottiene prendendo i resti delle divisioni a ritroso.

$$\begin{aligned} 27_{(10)} &= ?_{(2)} \\ 27 &= 2 \cdot 13 + 1 \\ 13 &= 2 \cdot 6 + 1 \\ 6 &= 2 \cdot 3 + 0 \\ 3 &= 2 \cdot 1 + 1 \\ 1 &= 2 \cdot 0 + 1 \\ 27_{(10)} &= 11011_{(2)} \end{aligned}$$

La memoria di un calcolatore, come abbiamo visto, è una risorsa a capacità limitata. Per questo, rappresentare i valori numerici, il cui insieme è infinito, è problematico: è possibile rappresentare solo un sottoinsieme dei possibili valori. A seconda dell'insieme numerico che si voglia rappresentare, si è adottato un meccanismo di rappresentazione diverso.

Per quanto riguarda i numeri naturali, avendo in un calcolatore a disposizione  $n$  bit, è possibile rappresentare i numeri che vanno da 0 a  $2^n - 1$ , quindi non è possibile rappresentare numeri arbitrariamente grandi. Avendo una rappresentazione finita, se si prova a sommare il valore 1 al valore  $2^n - 1$  (rappresentato con tutti caratteri 1) si ottiene una rappresentazione con tutti caratteri 0, ovvero la rappresentazione del numero 0. Ciò significa che la rappresentazione a  $n$  bit di numeri naturali è una rappresentazione *circolare*, il che è un fattore da tener

conto durante i calcoli.

Per rappresentare i numeri interi, occorre riservare alcune combinazioni i numeri negativi, restringendo così l'intervallo di rappresentazione dei numeri positivi. Nel riservare queste combinazioni, è possibile adottare due modalità particolari.

- La *rappresentazione modulo-segno* è la più intuitiva. Il bit meno significativo, ovvero quello più a sinistra della rappresentazione, viene usato ad indicare il segno; il valore 0 rappresenta il segno positivo, mentre il valore 1 rappresenta il segno negativo.

Questa modalità di rappresentazione è di facile lettura, ma è di difficile utilizzo durante le operazioni aritmetiche, poiché il bit meno significativo va trattato in maniera difforme dagli altri bit. In aggiunta, questa rappresentazione ha lo svantaggio di permettere due rappresentazioni del numero 0, una con il segno positivo e una con il segno negativo.

- La modalità di rappresentazione più usata è quella del *complemento alla base*.

Data una sequenza di  $k$  cifre rappresentante il numero  $n$  in base  $b$ , si definisce *complemento alla base di n* il valore  $C_b = b^k - n$  e *complemento diminuito di n* il valore  $C_d = C_b - 1$ . Usando questa rappresentazione, si ha un numero positivo quando si usa la rappresentazione binaria tradizionale, mentre si ha un numero negativo effettuando il complemento diminuito della rappresentazione numero (che nel caso binario prende il nome di *complemento a uno*).

A differenza della rappresentazione modulo-segno, usando il complemento alla base si ottengono due intervalli di rappresentazione simmetrici per i numeri positivi e negativi, ma si hanno sempre due rappresentazioni del numero 0. Il vero vantaggio nell'usare questa rappresentazione è la semplicità di effettuare le operazioni aritmetiche basilari.

Rappresentare i numeri razionali, infine, è un processo molto difficoltoso. Ciò è dovuto al fatto qualsiasi intervallo di rappresentazione si scelga se ne possono individuare infiniti al suo interno.

Uno dei metodi più usati per la rappresentazione dei numeri razionali è quello della **virgola mobile** (*floating point*). Secondo questa rappresentazione, un qualunque numero razionale può essere riscritto in modo da evidenziare una *mantissa*, ovvero la sua parte decimale, ed un *esponente*, ad esempio  $0,2345 \cdot 10^3$  o  $-0,235 \cdot 10^{-4}$ .

La rappresentazione in virgola mobile all'interno di un calcolatore consiste proprio nel rappresentare mantissa ed esponente del numero razionale espresso in base 2. In questo caso, un bit viene generalmente usato per il segno del numero, una porzione dei bit viene dedicata alla rappresentazione della mantissa e i rimanenti bit vengono utilizzati per rappresentare l'esponente in complemento a due.

## 2.4 Il sistema operativo

Finora abbiamo visto le varie componenti che formano l'hardware di un calcolatore, fornendo anche cenni su cosa si intende per software.

Come abbiamo detto, il *software* è l'insieme dei programmi, cioè delle sequenze di operazioni da eseguire nel calcolatore, che servono a finalizzare l'hardware alla risoluzione del problema presentato dall'utente. Questo termine comprende sia le operazioni di base, quali quelle fornite dal sistema operativo, sia le operazioni più complesse, definite dall'utente stesso.

Ma cosa si intende per sistema operativo?

**Definizione 2.3** (Sistema operativo). Si definisce **sistema operativo** quel componente del software di base responsabile della gestione delle risorse del calcolatore e della loro allocazione alle applicazioni.

Questi programmi offrono una interfaccia molto semplice all'utente e permettono di gestire le risorse in maniera ottimizzata. In particolare, le principali funzioni del sistema operativo sono:

- gestione della CPU e del processo di elaborazione;
- inizializzazione e terminazione del lavoro del calcolatore;
- gestione dell'accesso alla memoria centrale;
- gestione dei processi e del loro ordine di esecuzione;
- gestione dell'I/O;
- gestione delle informazioni riguardanti i file registrati sulle memorie secondarie;
- gestione delle protezioni delle cartelle di dati;
- supporto all'utente nella preparazione e nella messa a punto dei programmi.

Per gestire tutte queste funzioni, un sistema operativo è strutturato in maniera complessa e gerarchica, come mostrato nella Figura 2.6. Ciascun livello si occupa di gestire una diversa tipologia di risorse, come processori, memoria, periferiche, ecc....

Analizziamo ciascun livello del sistema operativo, partendo dal basso, ovvero dal più vicino all'hardware.

### Nucleo di un sistema operativo

Il **nucleo** di un sistema operativo (detto anche *kernel*) è il componente più complesso, occupandosi della gestione del processore (o dei processori) e dei *processi* (cioè dei programmi in esecuzione), ripartendo il tempo di calcolo se necessario/permesso.



Figura 2.6: Gerarchia di un sistema operativo

I programmi che compongono questo componente sono denominate *primitive* del nucleo e sono scritte nel linguaggio macchina specifico di un determinato processore.

### Gestore della memoria

Il **gestore della memoria** è responsabile dell'allocazione dinamica della memoria centrale ai programmi in esecuzione in un dato momento. A ciascun programma, infatti, il gestore alloca un'area di *memoria virtuale* sufficiente per l'esecuzione del programma e gestisce la corrispondenza tra la memoria virtuale (che può essere più grande della memoria reale) e la memoria reale.

Il gestore della memoria si occupa di suddividerla in blocchi logici, chiamati *pagine* o *segmenti* a seconda del modo in cui vengono costruiti ed usati.

Una *pagina* di memoria è un blocco di dimensione fissa, che viene assegnato tutto insieme ad un programma che ne faccia richiesta. Un *segmento*, viceversa, non ha una dimensione predeterminata, ma variabile a seconda delle richieste del programma.

Il vantaggio della gestione a pagine è soprattutto la semplicità, perché basta avere una tabella di corrispondenza fra pagina e programma assegnatario.

### Gestore delle periferiche

Il **gestore delle periferiche** permette di gestire ed usare le singole periferiche (ad esempio la singola stampante) senza conoscere esattamente la singola periferica, ma riconducendola ad una classe di periferiche virtuali ed usando un adattatore (**driver**) per le caratteristiche specifiche.

### Gestore dei file

Il **gestore dei file**, detto in inglese *file system*, è dedicato alla gestione della memoria secondaria, distribuendole come *file* (sequenze di lunghezza variabile di byte), *cartelle* (contenitori di file e/o di altre cartelle) e *volumi* (unità logiche che corrispondono a porzioni — *partizioni* — della memoria secondaria). La gestione dei volumi, rappresentati internamente come albero, è diversa tra i sistemi operativi Windows e Unix.

Le funzioni principali del gestore dei file sono:

- rendere trasparente all’utente l’utilizzo delle informazioni archiviate su memoria secondaria, nascondendo le modalità di memorizzazione fisica e creando, come abbiamo visto, una struttura logica;
- consentire l’accesso ai file (identificati tramite il *percorso* di cartelle da traversare per il file e tramite il suo *nome*, che indica anche il *formato* di codifica tramite *estensione*) in lettura e scrittura, risolvendo i problemi di concorrenza;
- predisporre funzioni di utilità come la lista, la cancellazione e la duplicazione;
- proteggere le informazioni dal punto di vista dell’integrità e della riservatezza.

In base a queste definizioni, possiamo definire i programmi sono dunque speciali file, riconosciuti dal sistema operativo come eseguibili.

### Interprete dei comandi

L’**interprete dei comandi** (denominato in inglese *shell*) espone una interfaccia agli utenti per interagire con il sistema stesso. Esso definisce le operazioni che possono essere utilizzate direttamente dall’utente finale, ovvero i *comandi*. Questa interfaccia può essere *a caratteri* (ad esempio il prompt dei comandi) o *grafica* e consiste in una serie di comandi che l’utente può eseguire.

### Programmi di utilità

Il sistema operativo è corredata da un insieme di **programmi di utilità**, usati per la configurazione, l’ottimizzazione e la gestione base del sistema stesso.



# Capitolo 3

## Problemi, algoritmi e programmi

Come abbiamo detto, uno degli scopi fondamentali dell'Informatica è la *risoluzione di problemi*, eventualmente in maniera automatica. Ma cosa sia un problema e come far capire all'esecutore come risolverlo è qualcosa che non è stato definito.

In questo capitolo vedremo più in dettaglio di definire cosa sia un problema, ma anche come comprenderlo per poter definire un metodo di risoluzione.

### 3.1 Cos'è un problema?

**Definizione 3.1** (Problema). Un **problema** è una situazione da risolvere per ottenere un certo risultato.

Osservando un fenomeno del mondo reale, dunque, si cerca di definire un *modello* di ciò che si vede, in modo da poter evidenziare subito gli aspetti fondamentali riducendoli a termini trattabili dall'esecutore. Il tipo di modello che viene utilizzato dipende dal problema che si vuole risolvere: si va da un livello di astrazione basso rispetto alla realtà osservata, come nel caso di *modelli iconici*, fino ad un livello di astrazione alta, come nel caso di *modelli matematici*.

Nel definire bene un problema occorre individuare, classificare e selezionare i dati a partire dalle entità che sono caratteristiche. In particolare:

- si parla di **variabili** nel caso di proprietà delle entità che assumono valori diversi nel tempo;
- si parla di **costanti** nel caso di proprietà delle entità il cui valore è stabile nel tempo.

Dal punto di vista fisico, variabili e costanti sono contenitori, ovvero celle di memoria in cui vengono salvati i valori di interesse.

Ad ogni variabile si associa in maniera univoca da un nome e da un *tipo* (l'insieme dei valori che la variabile può assumere nel tempo); nel tempo, ad una variabile può essere assegnato un valore piuttosto che un altro, rispettando il

suo tipo.

**Definizione 3.2** (Identificatore). Un **identificatore** è il nome che viene dato ad una variabile o ad una costante per distinguerla all'interno del modello in esame.

## 3.2 La metodologia di lavoro: dal problema all'algoritmo

Non tutti i problemi sono risolvibili con l'ausilio dell'Informatica; quelli per cui è possibile una precisa formalizzazione, magari in qualche formalismo espresivo, sono i *problemi di interesse* dell'Informatica. L'uso di un formalismo specifico, infatti, permette di definire un insieme di regole che potranno essere seguite da un *esecutore specializzato* per risolvere il problema, portando alla definizione dell'*algoritmo risolutivo*.

Possiamo dunque classificare i problemi in:

- *decidibili*, se per il problema è possibile definire un algoritmo risolutivo;
- *non decidibili*, se per il problema non è possibile definire un algoritmo risolutivo.

Per i problemi decidibili, nella definizione di un algoritmo bisogna valutare anche l'**efficienza temporale e spaziale**, ovvero quanto tempo ci impiegherà l'esecutore per completare l'algoritmo e quanto spazio occuperà per memorizzare i dati; questa valutazione sarà effettuata a livello teorico, cioè numero di operazioni da compiere o valori da memorizzare.

Esistono, però, dei problemi che, pur essendo classificati come decidibili, generano soluzioni non efficientemente, perché il costo temporale e/o spaziale è insostenibile.

Un esempio di problema decidibile è quello della determinazione del *massimo comune divisore* di due interi positivi; a seconda della formulazione si riesce ad individuare una strategia diversa di soluzione.

- Formulando il problema come "Determinare  $z = \text{MCD}(x, y)$  dove  $x, y, z \in \mathbb{N}$  e  $x \geq y$ ", arriviamo alla seguente formulazione matematica:

$$\begin{aligned} \text{MCD}(x, y) &= \max(D_x, D_y) \\ D_x &= \{d \in \mathbb{N} \mid \exists q \in \mathbb{N} \ x = q \cdot d\} \\ D_y &= \{d \in \mathbb{N} \mid \exists q \in \mathbb{N} \ y = q \cdot d\} \end{aligned}$$

Una prima strategia di soluzione consiste nel calcolare i due insiemi  $D_x$  e  $D_y$ , determinare l'intersezione dei due insiemi e poi il valore massimo dell'intersezione. Questa strategia è molto onerosa, perché occorre calcolare tutti i divisori di  $x$  e  $y$ .

- Una seconda strategia di soluzione consiste nell'individuare il valore minimo  $m$  tra  $x$  e  $y$ , nel verificare se  $m$  divide sia  $x$  sia  $y$  e, in caso positivo, fermarsi perché si è trovato il Massimo Comun Divisore; in caso negativo, decremento  $m$  di uno e ripeto il controllo al più fino a  $m = 1$ . Anche questa strategia è molto onerosa, perché occorre effettuare molte divisioni.
- Una terza strategia di soluzione porta ad una sequenza di regole ben nota: calcolo il resto della divisione tra  $x$  e  $y$  e se è zero quel resto è il valore cercato, altrimenti si ripete la procedura cercando il resto della divisione tra  $y$  e il resto precedente.

Questo esempio ci mostra che, anche se inconsciamente, abbiamo seguito una sequenza di passi per poter definire la strategia di risoluzione del problema.

Il primo passo è stato quello della *descrizione del problema*, che ci permette di definire con precisione:

- quali dati abbiamo a disposizione (*i dati di input*), i quali non devono essere né sovrabbondanti né troppo ridotti;
- i risultati che si vogliono ottenere (*i dati di output*);
- le risorse a disposizione sia dal punto di vista logico (come tabelle o criteri di calcolo) sia dal punto di vista fisico (come calcolatori elettronici);
- le soluzioni adottate in precedenza per la risoluzione del problema.

Il secondo passo consiste nell'organizzare la strategia di risoluzione in una serie di operazioni da attuare secondo un ordine definito, in modo tale da poter raggiungere i risultati attesi a partire dai dati iniziali, definendo così l'*algoritmo risolutivo*.

**Definizione 3.3** (Algoritmo). Si definisce **algoritmo** la descrizione di un insieme finito di istruzioni che devono essere eseguiti per portare a termine un dato compito e per raggiungere un risultato definito in precedenza.

Il termine algoritmo è un termine di origine matematica, dove indica le regole per compiere un calcolo o per risolvere un problema; questo termine deriva dal nome dell'algebrista arabo *Al-Khuwarizmi*. Un algoritmo, per essere tale, deve soddisfare le seguenti caratteristiche.

- Deve essere composto da un numero *finito* di istruzioni e deve presentare un punto di inizio e un punto di fine.
- Deve essere *completo ed esaustivo*, nel senso che per tutti i casi che si possono verificare durante l'esecuzione deve essere indicata la soluzione da seguire.
- Deve essere *riproducibile*, cioè ogni successiva esecuzione dello stesso algoritmo con i medesimi dati iniziali deve produrre sempre i medesimi risultati finali.

Anche le istruzioni che fanno parte di un algoritmo devono soddisfare alcune caratteristiche.

- Ogni istruzione deve essere concretamente *realizzabile* dall'esecuzione.
- Le istruzioni devono essere *precise e non ambigue* in modo che non lascino dubbi nell'interpretazione da parte dell'esecutore.
- Ogni istruzione deve avere una *durata limitata nel tempo*.
- Ogni istruzione deve produrre un *risultato osservabile*.
- Ogni istruzione deve avere *carattere deterministico*, cioè deve produrre sempre il medesimo effetto se eseguita a partire dalle stesse condizioni iniziali.
- Ogni istruzione deve essere *elementari*, cioè non ulteriormente scomponibili rispetto alle capacità dell'esecutore.

### 3.3 Dall'algoritmo al programma

Finora abbiamo visto come si passi dalla formulazione di un problema al suo algoritmo risolutivo. Per analizzare come l'algoritmo possa essere eseguito occorre analizzare la struttura astratta di appartenenza degli oggetti da trattare, cioè passare alle loro *rappresentazioni*.

Queste rappresentazioni, note anche come **azioni**, rappresentano eventi che si compiono in un intervallo di tempo finito e che producono un risultato previsto e ben determinato su di un oggetto della realtà. La loro descrizione viene espressa tramite un opportuno formalismo espressivo, detto *linguaggio*. Nel caso di un calcolatore elettronico si fa uso di un adeguato *linguaggio di programmazione*, le cui regole siano comprensibili sia all'utente sia al calcolatore.

Azioni composte, ovvero azioni che possono essere frammentate in azioni più semplici, prendono il nome di **processi**.

L'esecuzione di un algoritmo genera dunque processi ed azioni; la rappresentazione di questi processi ed azioni nel linguaggio di programmazione scelto prende il nome di **programma**.

#### 3.3.1 Diagrammi a blocchi e pseudocodifica

Per trasformare un algoritmo in un programma, occorre descrivere in maniera dettagliata l'algoritmo usando linguaggi utili a questo scopo. I due linguaggi comunemente più usati sono i diagrammi a blocchi e la pseudocodifica.

Un **diagramma a blocchi** consiste in una descrizione grafica dell'algoritmo, fornendo a chi lo legge una visione immediata del procedimento da seguire e dell'ordine delle istruzioni.

I diagrammi a blocchi sono formati da simboli di forma diversa, ciascuno con un proprio significato, collegati da frecce per indicare il flusso delle istruzioni.

- Gli ellissi vengono usati per rappresentare il punto di partenza e di terminazione dell'algoritmo.

- I rettangoli (detti anche *simboli di elaborazione*) vengono usati per rappresentare le istruzioni da eseguire.
- I parallelepipedi vengono usati per rappresentare operazioni di immissione o di emissione dei dati.
- I rombi (detti anche *simboli di decisione*) vengono usati per rappresentare una istruzione di confronto tra due dati.
- I rettangoli con doppio bordo verticale vengono usati per rappresentare il richiamo ad un *sottoprogramma* dell'algoritmo.

Un esempio di un diagramma di flusso è visibile nella Figura 3.1.

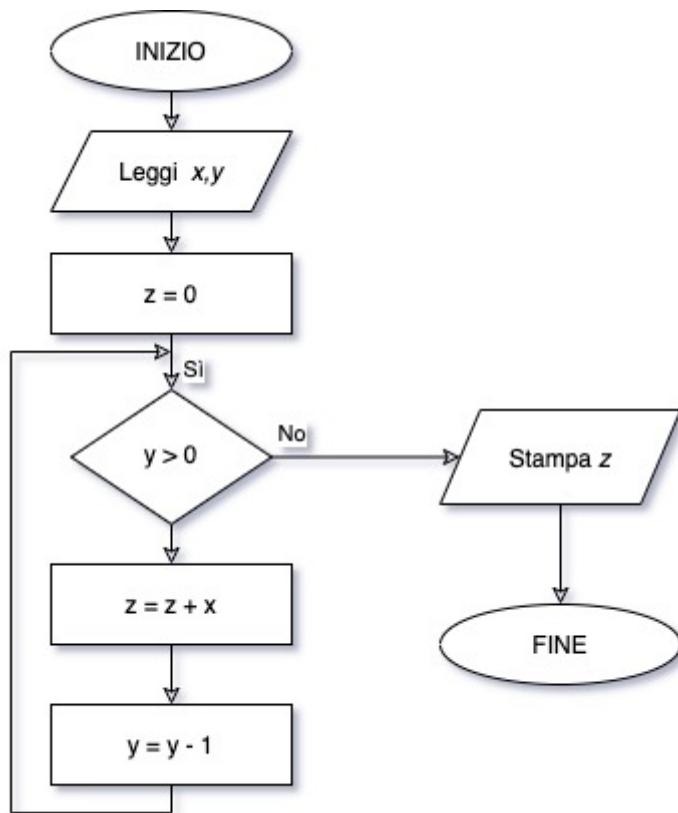


Figura 3.1: Diagramma di flusso per il problema del massimo comun divisore

La **pseudocodifica** permette di descrivere un algoritmo usando termini e parole del linguaggio comune. In questo modo, chi deve tradurre l'algoritmo in un programma si può concentrare solo sulla logica della soluzione, astraendosi dal linguaggio di programmazione che ha deciso di usare.

Sia il diagramma a blocchi sia la pseudocodifica possono essere usati per simulare l'esecuzione dell'algoritmo. Questa simulazione può essere descritta in modo strutturato mediante la sua *traccia*, ovvero tramite una tabella in cui

vengono elencate le istruzioni eseguite nonché il valore di ciascuna variabile dopo l'esecuzione di ciascuna istruzione.

### 3.3.2 La programmazione strutturata e il teorema di Böhm-Iacopini

Abbiamo visto finora come definire un problema e i dati che si possono usare, come organizzare un algoritmo e descriverlo in maniera esaustiva e semplice. Non abbiamo, però, descritto come tradurre l'algoritmo nel linguaggio di programmazione scelto.

Ci concentreremo dunque sulla fase di *stesura del programma*, descrivendo l'insieme di regole e linee guida che devono essere seguite per organizzare bene il proprio lavoro di programmatore. Queste regole permettono di schematizzare le varie fasi, migliorando la qualità del proprio lavoro, e definiscono una metodologia nota come *programmazione strutturata*.

**Definizione 3.4** (Programmazione strutturata). La **programmazione strutturata** è la progettazione, la realizzazione e il collaudo di un programma costituito di parti che dipendono l'una dall'altra secondo un ben definito modello organizzativo.

Un programma può essere descritto come insieme di quattro tipi di istruzioni:

- input/output;
- assegnazione;
- operazioni logico-aritmetiche;
- *operazioni di controllo*, come la *sequenza*, la *selezione* e l'*iterazione*.

Le operazioni di controllo godono di una proprietà fondamentale: rappresentano un flusso con un solo ingresso e una sola uscita. Ciò permette di concatenare e annidare a vari livelli queste strutture, creando programmi complessi a piacere, come affermato anche dal *Teorema di Böhm-Iacopini*.

**Teorema 3.1** (Böhm-Iacopini). *Ogni algoritmo può essere espresso usando esclusivamente le operazioni di sequenza, selezione e iterazione.*

#### Le sequenze

La **sequenza** rappresenta un insieme di istruzioni, semplici come l'assegnazione o complesse come altre operazioni di controllo. A seconda del linguaggio di programmazione scelto, la sequenza viene racchiusa da parentesi graffe, dalle parole chiavi BEGIN e END, da una indentazione diversa, o da altri metodi.

Questa operazione di controllo prende questo nome perché le istruzioni al suo interno vengono eseguite una dopo l'altra, secondo l'ordine con cui sono state scritte. Per questo motivo, in un diagramma a blocchi le sequenze vengono rappresentate tramite i vari blocchi posti uno in fila all'altro.

### Le selezioni

La **selezione** è quella operazione di controllo che consente di effettuare una scelta in base al valore di una condizione. Se il valore  $v_{\text{espressione}}$  della condizione fosse VERO, si eseguirebbe la sequenza di istruzioni parte-true, viceversa si eseguirebbe la sequenza di istruzioni parte-false.

Questo tipo di operazione viene anche detta *alternativa*, perché l'esecutore eseguirà in alternativa l'una o l'altra sequenza di istruzioni. In un diagramma a blocchi, l'operazione di selezione viene identificata tramite il rombo, da cui escono i due rami parte-true e parte-false, le quali si ricongiungono alla fine delle sequenze.

Mettendo in cascata più operazioni di selezione si ottiene una struttura di controllo derivata, la **selezione multipla**. Quando un esecutore incontra una selezione multipla compie le seguenti azioni:

- se il valore  $v_{\text{espressione}}$  è presente in una delle liste di valori, allora viene eseguita la sequenza di istruzioni corrispondente;
- in caso contrario viene eseguita la sequenza parte-false.

### L'iterazione

L'**iterazione** è quella operazione di controllo che consente di eseguire un blocco di istruzioni in modo ripetuto in base alla veridicità di una condizione. In un diagramma a blocchi, l'iterazione viene rappresentata tramite una sequenza di istruzioni, preceduta o seguita da un rombo per la valutazione della condizione.

Esistono in letteratura tre diversi costrutti iterativi.

- L'*iterazione post-condizionale* è quel tipo di iterazione in cui la valutazione della condizione avviene dopo l'esecuzione della sequenza di istruzioni da ripetere (il *corpo*) e termina solo quando la condizione diventa vera.
- L'*iterazione pre-condizionale* è quel tipo di iterazione in cui la valutazione della condizione avviene prima dell'esecuzione del corpo dell'operazione e termina solo quando la condizione è falsa.
- L'*iterazione con contatore* (o *iterazione enumerativa*) è una particolare versione dell'iterazione pre-condizionale, in cui la valutazione della condizione consiste nel enumerare una sequenza numerica, in modo tale da ripetere il corpo dell'operazione un numero predeterminato di volte.

Alcuni linguaggi di programmazione forniscono anche due costrutti per forzare la terminazione dell'esecuzione dell'iterazione o del singolo ciclo di iterazione.



# Capitolo 4

## La programmazione

Nel Capitolo 3 abbiamo visto come riuscire a definire le soluzioni ai nostri problemi, descrivendole come algoritmi, e abbiamo anche visto in cosa consiste un programma. Ciò che non abbiamo visto è come scrivere un programma.

In questo capitolo vedremo di definire lo strumento principale per la scrittura di un programma, ovvero il *linguaggio di programmazione*, per poi definire i diversi stili di programmazione e delle linee guida per scrivere i programmi.

### 4.1 Costruzione analitica di un linguaggio di programmazione

Ogni discorso tra due entità, effettuato tramite un mezzo di comunicazione, richiede che oltre ad un canale di comunicazione si decida una codifica con cui queste informazioni vengano trasmesse. Questo codice prende il nome di *linguaggio*.

Il linguaggio può essere *verbale*, definito da un insieme di segni scritti e espressi vocalmente, e *non verbale*, in cui il messaggio non è veicolato da alcun segno scritto/orale ma da altri atteggiamenti del mittente (come il tono, la frequenza, le pause, i gesti).

In questa nostra analisi, ci concentreremo sui linguaggi verbali.

Un linguaggio verbale si classifica a sua volta in *naturale*, quando si fa riferimento al linguaggio usato dagli esseri umani per comunicare e che si è sviluppato nel corso di millenni, e in *artificiale*, quando si fa riferimento a linguaggi creati dall'uomo appositamente per scopi precisi.

Ogni linguaggio è definito a partire da un *alfabeto*, ovvero un insieme di simboli unitari che il mittente usa per definire sequenze (le *parole*) da esprimere in maniera scritta o orale. La definizione delle parole avviene secondo regole precise che ne determinano la loro validità nel linguaggio; queste regole prendono il nome di *grammatica*.

Comunicare solamente attraverso parole non è una cosa semplice: occorre mettere insieme tutte queste parole in sequenze più lunghe (le *frasi*), in maniera tale che per il destinatario del messaggio, ma anche per il mittente, queste sequenze

siano corrette e sensate. Per fare ciò, ogni linguaggio definisce anche una *sintassi* (le regole per creare frasi valide nel linguaggio, denominate anche *regole di produzione*) e una *semantica* (le regole per definire il significato di ogni frase).

### 4.1.1 Costruzione algebrica di un linguaggio di programmazione

Basandoci su tutto ciò che abbiamo detto finora, un *linguaggio di programmazione* è un linguaggio verbale artificiale, usato da mittenti (i *programmatori*) per comunicare ai destinatari (i calcolatori elettronici) messaggi il cui scopo è istruirli su che azioni eseguire.

Vogliamo dunque ora definire in maniera più formale cosa sia un linguaggio di programmazione, partendo dai concetti di base.

**Definizione 4.1** (Universo linguistico). Dato un insieme finito non vuoto  $V$ , denominato **alfabeto**, si definisce **universo linguistico su  $V$**  (e si indica con  $V^*$ ) l'insieme delle sequenze finite di lunghezza arbitraria di elementi di  $V$ , ciascuno dei quali denominato **simbolo terminale**.

Dal punto di vista matematico possiamo notare che nonostante l'alfabeto sia un insieme finito, l'universo linguistico non lo è. Infatti,  $V^*$  è un insieme numerabile e i suoi sottoinsiemi sono non numerabili e non tutti finiti.

Uno di questi sottoinsiemi forma il linguaggio.

**Definizione 4.2** (Linguaggio). Un **linguaggio  $L$**  sull'alfabeto  $V$  è un sottoinsieme di  $V^*$ .

A partire dall'alfabeto è possibile definire una grammatica, che ci aiuterà poi a definire un particolare linguaggio.

**Definizione 4.3** (Grammatica). Una **grammatica** o **sintassi  $\mathcal{G}$**  è definita da:

- un alfabeto di *simboli terminali*  $V$ ;
- un alfabeto di *simboli non terminali*  $N$  tale che  $V \cap N = \emptyset$ ;
- un simbolo  $s \in N$  detto *assioma*, da cui parte il processo di produzione;
- un insieme finito  $P$  di *regole di produzione* del tipo  $X \Rightarrow \alpha$ , dove  $X \in N$  e  $\alpha \in (N \cup V)^*$ .

Le produzioni dell'insieme  $P$  (le quali possono essere scritte anche nella forma  $X ::= \alpha$ ) possono essere raggruppate secondo questa regola: se esistono più regole aventi la stessa parte sinistra, si possono raggruppare usando la notazione  $X \Rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ .

**Definizione 4.4** (Linguaggio generato da grammatica). Data una grammatica  $\mathcal{G}$ , si dice **linguaggio generato da  $\mathcal{G}$**  (indicandolo con  $L_{\mathcal{G}}$ ) l'insieme di frasi di  $V^*$  derivabili a partire dall'assioma  $s$ .

**Definizione 4.5** (Linguaggio di programmazione). Un **linguaggio di programmazione**  $L$  su un alfabeto  $V$  è un sottoinsieme di  $V^*$  per cui esiste una grammatica  $\mathcal{G}$  tale che  $L = L_{\mathcal{G}}$ .

### 4.1.2 Il formalismo BNF

Nella definizione delle regole di produzione abbiamo introdotto un particolare formalismo espressivo per tali regole. Tramite questo formalismo, insieme alla definizione dei simboli terminali e non, è possibile definire l'intera grammatica.

Questo formalismo è un *metalinguaggio*, ovvero un linguaggio formalmente definito per definire linguaggi artificiali, noto come **forma di Backus-Naur** (in inglese *Backus-Naur Form*, BNF), il cui nome deriva dai due studiosi che per primi l'adottarono negli anni '50, John Backus e Peter Naur.

Questo formalismo, spesso, viene usato con alcune estensioni che permettono una scrittura più concisa, che prende il nome di **Extended BNF** (EBNF); secondo una di queste estensioni, se nella parte destra di una regola di produzione un simbolo appare tra parentesi quadre, quel simbolo è opzionale. Viceversa, se un simbolo appare tra parentesi quadre con un numero naturale ad apice, quel simbolo può apparire zero o più volte (fino a quante ne sono indicate nell'apice). Considerando come insieme dei simboli terminali

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

e come insieme dei simboli non terminali

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

, un esempio di grammatica espressa tramite BNF è il seguente:

$$\begin{aligned} \langle\text{frase}\rangle &::= \langle\text{soggetto}\rangle\langle\text{verbo}\rangle\langle\text{complemento}\rangle \\ \langle\text{soggetto}\rangle &::= \langle\text{articolo}\rangle\langle\text{nome}\rangle \\ \langle\text{articolo}\rangle &::= \text{il} \mid \text{lo} \\ \langle\text{nome}\rangle &::= \text{gatto} \mid \text{topo} \mid \text{monte} \\ \langle\text{verbo}\rangle &::= \text{manga} \mid \text{beve} \\ \langle\text{complemento}\rangle &::= \langle\text{articolo}\rangle\langle\text{nome}\rangle \end{aligned}$$

L'applicazione delle varie regole di produzione si chiama **processo di produzione**; questo processo può essere facilmente rappresentato mediante un albero, denominato **albero di derivazione sintattica** o **albero sintattico**.

## 4.2 I sottoprogrammi

Quando abbiamo parlato di algoritmi, abbiamo detto che ogni istruzione deve essere comprensibile e concretamente realizzabile da un esecutore. Il livello di astrazione di un algoritmo è dunque strettamente dipendente dalle capacità dell'esecutore stesso, ma anche dalle nozioni del programmatore.

Se il programma richiede un algoritmo molto complesso per la sua esecuzione, può essere utile adottare il metodo dei *raffinamenti successivi*, che consiste nel suddividere il problema in sottoproblemi più semplici di complessità minore. Questo metodo è anche detto *sviluppo top-down*, proprio perché si parte dal macro-problema per arrivare a tanti sotto-problemi.

Ad ogni problema, come abbiamo detto, corrisponde un programma. Allo stesso modo, ad ogni sotto-problema corrisponde un *sotto-programma*. In precedenza, quando abbiamo descritto i diagrammi a blocchi, abbiamo parlato di sottoprogrammi. Ma in cosa consiste un sottoprogramma?

**Definizione 4.6** (Sottoprogramma). Un **sottoprogramma** è una istruzione aggiuntiva del linguaggio di programmazione, definita dal programmatore, che agisce sui dati utilizzati dal programma, “nascondendo” la sequenza delle operazioni effettivamente eseguite dalla macchina.

Un sottoprogramma, dunque, permette di mettere a fattor comune operazioni che vengono eseguite spesso all’interno del programma, in modo da ottimizzare la manutenibilità del codice sorgente stesso.

Quando si definisce un sottoprogramma, occorre:

- stabilire un identificatore;
- definire il *corpo* del sottoprogramma, ovvero l’insieme di istruzioni predefinite o altri sottoprogrammi;
- definire le *modalità di comunicazione* tra il *chiamante* (il programma principale o un altro sottoprogramma) e il *chiamato* (il sottoprogramma che si sta scrivendo).

**Definizione 4.7** (Prototipo di un Sottoprogramma). Il **prototipo** di un sottoprogramma rappresenta la descrizione del formato dei messaggi con cui si può chiedere al sottoprogramma stesso di eseguire il suo corpo.

Quando un sottoprogramma viene invocato, l’esecuzione dell’unità di programma *chiamante* viene sospesa e il controllo passa al sottoprogramma chiamato, che eseguirà tutte le istruzioni prima di ritornare il controllo al chiamante. Queste due unità comunicano tra loro mediante **parametri**.

È possibile distinguere due tipologie di parametro.

- Si parla di **parametri formali** per indicare i parametri specificati nel prototipo del sottoprogramma. In genere i parametri formali sono in numero prefissato e ad ognuno di essi viene associato un tipo. L’insieme di identificatore e parametri formali prende il nome di **firma** del sottoprogramma.
  - Si parla di **parametri attuali** per indicare i valori effettivamente forniti al sottoprogramma dall’unità di programma chiamante.
- I parametri attuali devono corrispondere in numero, posizione e tipo a quelli formali per poter essere legati. Questo legame può avvenire:

- per **valore**, ovvero passando una copia del valore della variabile in ingresso al sottoprogramma;
- per **indirizzo** (detto anche per *riferimento*), ovvero passando l'indirizzo della locazione di memoria contenente il valore di interesse, che potrà essere eventualmente modificato.

Possiamo identificare due tipologie di sottoprogrammi:

- le **funzioni**, che sono astrazioni del concetto di *operatore* e che restituisce un valore;
- le **procedure**, che sono astrazioni del concetto di *istruzione* e che non restituisce alcun valore.

## 4.3 Paradigmi di programmazione

Come abbiamo detto precedentemente, un *programma* (detto anche *applicazione*) è usato per risolvere un problema grazie all'aiuto di un calcolatore; in pratica, un programma mostra all'utente, tramite una interfaccia (grafica o a linea di comando), come il calcolatore rappresenta una parte di mondo reale tramite *oggetti*. Col fatto che una applicazione maschera molti dettagli ai suoi utenti, l'utente può pensare che stia usando una macchina dedicata, senza essere a conoscenza dei concetti della programmazione.

Ma cosa si intende con **programmazione**?

- La scrittura di programmi.
- Forzare un calcolatore a fare quello che si vuole (*controllo*), fornendogli istruzioni dettagliate (*concretezza*) che spiegano anche come realizzarle (*insegnamento*).
- La modellazione della realtà di interesse, evitando di descrivere dettagli inutili (*astrazione*).
- Un insieme di frasi, dette **istruzioni**, che descrivono la realtà di interesse in un certo linguaggio di programmazione.

Esistono vari approcci alla programmazione, detti **paradigmi di programmazione**. I più noti sono:

- la **programmazione imperativa**, in cui si specificano le azioni che devono essere eseguite in sequenza per calcolare i risultati a partire dai dati;
- la **programmazione funzionale**, in cui si specificano le azioni che devono essere eseguite sotto forma di funzioni parametriche;
- la **programmazione logica**, in cui si descrivono le proprietà che devono essere verificate dai risultati sulla base delle proprietà verificate dai dati;

- la **programmazione orientata agli oggetti**, secondo cui il mondo reale è fatto di oggetti, per cui la sua rappresentazione nel calcolatore deve consistere di *oggetti software* che cooperano tra loro.

### 4.3.1 Traduzione ed Esecuzione dei Programmi

Come abbiamo detto più volte, un calcolatore è una macchina per eseguire programmi che, generalmente, qualcuno ha scritto per noi. Un calcolatore è anche una macchina programmabile, cioè le cui capacità possono essere aumentate scrivendo noi nuovi programmi usando il linguaggio di programmazione più opportuno per le nostre esigenze.

Esistono vari livelli di linguaggi di programmazione, in base al livello di astrazione. Quello meno astratto è il **linguaggio binario**, in cui le istruzioni e i dati sono rappresentati in codice binario; successivamente abbiamo i **linguaggi assemblativi**, simbolici, che raggruppano poche istruzioni binarie in una singola istruzione. I **linguaggi di programmazione ad alto livello** sono dunque quelli che consentono di scrivere i programmi con istruzioni simboliche, più vicine alla logica umana di risoluzione dei problemi.

Il calcolatore, dunque, cerca di tradurre ciascuna istruzione del programma in un linguaggio macchina, portandole in memoria centrale, dove verranno eseguite una alla volta secondo il *ciclo fetch-decode-execute* che abbiamo visto in precedenza.

Esistono due modalità di traduzione dei linguaggi di programmazione in linguaggio macchina: la compilazione e l'interpretazione.

La **compilazione** consiste nel trasformare il programma sorgente in un programma eseguibile in linguaggio macchina; questa trasformazione viene eseguita da un programma particolare detto **compilatore**. La compilazione è una attività che avviene una sola volta, permettendo di eseguire il suo risultato (denominato anche *oggetto*)

L'**interpretazione** consiste nel trasformare ciascuna istruzione del programma in istruzioni del linguaggio macchina, eseguendole immediatamente; la trasformazione viene eseguita da un programma detto **interprete**.

Volendo fare un paragone, la compilazione corrisponde alla traduzione di un libro, mentre l'interpretazione corrisponde alla traduzione simultanea di un discorso.

Esiste una differenza tra linguaggi di programmazione compilati e interpretati:

- i linguaggi compilati prevedono di distribuire un file eseguibile per ogni piattaforma, mantenendo il sorgente nelle mani del programmatore;
- i linguaggi interpretati prevedono la distribuzione del file sorgente e l'installazione del programma interprete su ogni piattaforma.

## 4.4 Leggibilità del codice

Quando si scrive un programma, una delle qualità che deve soddisfare è la *leggibilità*. Si parla di leggibilità quando il codice sorgente di un programma è facilmente comprensibile ad una prima lettura.

Questa qualità non è importante per l'utente finale, visto che non ha accesso al codice sorgente, ma è importante per il programmatore stesso, poiché migliora la manutenibilità del programma stesso.

Esistono una serie di *convenzioni stilistiche* tali che permettono di migliorare la leggibilità del codice sorgente che si sta scrivendo.

Vedremo ora alcune di queste convenzioni stilistiche.

### 4.4.1 I commenti

Un *commento* è una frase che documenta una porzione di codice, illustrandone lo scopo o il significato. La presenza (o l'assenza) di un commento non ha alcun effetto sull'esecuzione di un programma, ma migliora la leggibilità del codice sorgente.

Un commento può essere classificato in base alla sua finalità; per cui abbiamo:

- *commenti documentazione*, che descrivono lo scopo di un programma, una funzione, una sottoprocedura, ecc...;
- *commenti implementazione*, che descrivono le scelte realizzative adottate dal programmatore;
- *commenti asserzione*, che descrivono proprietà che si verificano sempre durante l'esecuzione del codice sorgente.

Per migliorare la leggibilità del codice usando i commenti, occorre:

- precedere l'inizio del programma con un commento documentazione, in modo da descrivere lo scopo stesso dell'applicazione;
- precedere la definizione di ciascuna funzione/sottoprocedura con un commento documentazione che descriva lo scopo dell'operazione implementata;
- affiancare a ciascuna variabile un commento implementazione.

### 4.4.2 Scelta degli identificatori

La scelta degli identificatori è importante tanto quanto scrivere un buon commento. Un buon identificatore deve essere significativo e auto-documentante, ovvero deve capirsi immediatamente lo scopo della variabile/costante/funzione leggendo il suo identificativo.

In genere:

- gli identificativi delle variabili e delle costanti sono rappresentati tramite sostantivi, cui eventualmente aggiungere degli aggettivi;
- gli identificativi delle funzioni/sottoprocedure sono rappresentati tramite verbi o frasi verbali, ad indicare azioni da compiere.

Può capitare che si usi un acronimo come identificatore. Sebbene ciò sia possibile, è sempre buona pratica fare ciò con cautela, documentando tramite commenti il significato di tale acronimo.

#### 4.4.3 Indentazione

*Indentare* parti di un codice sorgente significa esplicitare la relazione di contenimento di tali parti mediante opportuno incolonnamento.

Alcuni linguaggi di programmazione ignorano il concetto di indentazione, lasciandola libera al programmatore, mentre altri forzano il programmatore ad incolonizzare il codice in maniera specifica per poter avviare la compilazione o l'interpretazione. In entrambi i casi, indentare in modo arbitrario il codice sorgente del proprio programma migliora la leggibilità e, quindi, la manutenibilità del programma stesso.

# Capitolo 5

## Tipi dati ed espressioni

Come abbiamo visto in precedenza, molti elementi di un programma, come ad esempio variabili e costanti, sono caratterizzati da un tipo. I tipi svolgono un ruolo molto importante, perché servono a verificare la semantica stessa delle istruzioni che vengono usate.

In questo capitolo vediamo in cosa consiste un tipo, quali sono i principali tipi che possiamo identificare e come individuare il tipo in espressioni composte.

### 5.1 Tipo di dato

Nel Capitolo refcap:nozioni abbiamo definito un dato come una informazione rappresentata nel calcolatore in una forma tale che il calcolatore possa trattarli. Dal punto di vista pratico sono sequenze di bit memorizzate, ma il modo in cui il calcolatore interpreta queste sequenze è determinato dal tipo.

**Definizione 5.1** (Tipo di dato). Un **tipo di dato** (o semplicemente *tipo*) è costituito da un insieme di possibili *valori* (detto **dominio** del tipo) e da un insieme di **operazioni** che possono essere applicate agli elementi di tale dominio.

La nozione di tipo deriva dall'Algebra Lineare e dalla Logica Matematica, con alcune differenze. In particolare, in un linguaggio di programmazione il dominio associato ad un tipo è finito, mentre per l'Algebra Lineare il dominio può essere anche infinito. Per questo motivo, in Informatica si dice che un tipo è costituito da:

- una componente statica, formata da tutti gli elementi che costituiscono il dominio, opportunamente rappresentati;
- una componente dinamica, formata dalle operazioni sul dominio, limitandosi a tutti gli operatori ammissibili.

In base ai valori rappresentati, si possono identificare due categorie di tipi:

- i *tipi primitivi*, predefiniti nel linguaggio di programmazione stesso, che rappresentano valori numerici o booleani o caratteri;
- i *tipi derivati*, che rappresentano oggetti composti.

### 5.1.1 Tipi primitivi

Un **tipo primitivo** è un tipo semplice, i cui valori non possono essere decomposti ulteriormente. Tutti i linguaggi di programmazione *tipizzati* (ovvero quelli in cui è necessario associare alle variabili, alle espressioni e più in generale ai termini dei programmi delle dichiarazioni di tipo) offrono vari tipi primitivi, che il programmatore può combinare tra loro per ottenere un tipo derivato più adatto alla sua esigenza.

I principali oggetti rappresentati tramite tipi primitivi sono i numeri interi, i numeri reali, i caratteri alfanumerici e i valori booleani.

#### Il tipo booleano

Il tipo *booleano* ha due soli valori: `TRUE` ("vero") e `FALSE` ("falso"). Essi vengono utilizzati in modo speciale nelle espressioni condizionali per controllare il flusso di esecuzione e possono essere manipolati con gli operatori booleani `AND`, `OR`, `NOR` e così via.

Anche se in teoria basterebbe un solo bit per memorizzare un valore booleano, per motivi di efficienza si usa in genere un'intera parola di memoria, come per i numeri interi "piccoli" (una parola di memoria a 8 bit, per esempio, può memorizzare numeri da 0 a 255, ma il tipo booleano utilizza solo i valori 0 e 1).

#### I tipi numerici

I tipi di dati numerici includono i numeri interi e i numeri razionali in virgola mobile (in genere seguendo lo standard IEEE 754-1985), che sono astrazioni dei corrispondenti insiemi di numeri della matematica. Quasi tutti i linguaggi includono tipi di dati numerici come tipi predefiniti e forniscono un certo numero di operatori aritmetici e di confronto su di essi.

A differenza degli insiemi numerici della matematica, i tipi di dati numerici sono spesso limitati (includono cioè un massimo e un minimo numero rappresentabile), dovendo essere contenuti in una singola parola (word) di memoria.

#### I caratteri

Il tipo carattere contiene un carattere, generalmente ASCII, memorizzato in un byte. L'emergere del nuovo standard Unicode ha fatto sì che la memorizzazione di un singolo carattere utilizzi ora 16 o 32 bit.

### 5.1.2 Tipi derivati

Un **tipo derivato** si ottiene per composizione dai tipi primitivi. Si definisce un tipo derivato quando si ha la necessità di aggregare molti dati di tipo primitivo per avere una rappresentazione più semplice e rendere più veloce il loro

ritrovamento all'interno della memoria.

Si possono individuare quattro grandi famiglie di tipi derivati:

- i *puntatori* e i *riferimenti*;
- gli *array*;
- i *record*;
- i *tipi generici*

I tipi derivati devono o possono offrire due operazioni particolari:

- il *costruttore*, che inizializza le nuove istanze del tipo fornendo i valori dei vari elementi;
- il *distruttore*, che contiene le azioni da eseguire quando l'istanza del tipo viene deallocated.

### 5.1.3 Conversione tra tipi

Visto che esistono molteplici tipi di dato, talvolta occorre effettuare delle conversioni tra un tipo e un altro, in modo da trasformare la rappresentazione di un valore.

Sono possibili due modalità di conversione:

- quella *implicita*;
- quella *esplicita*.

#### Conversione implicita

Quando si assegna il valore di una variabile ad un'altra (`var = espressione`), generalmente, si fa attenzione che il dominio dei due tipi sia lo stesso. Quando il dominio del tipo di `var` è più ampio del dominio del tipo di `espressione`, l'assegnazione è comunque valida, ma il compilatore o l'interprete convertono automatica il tipo "più piccolo" nel tipo "più esteso".

Questo tipo di conversione automatica prende il nome di **conversione implicita** o *promozione*.

#### Conversione esplicita

Quando il dominio del tipo di `var` è più ristretto del dominio del tipo di `espressione`, si rischia una possibile perdita di precisione nella rappresentazione. Per questo motivo, se l'assegnazione `texttvar = espressione` è necessaria, occorre effettuare una **conversione esplicita** (o *cast*).

### 5.1.4 Tipo di dato astratto

Finora abbiamo descritto cosa sia un tipo di dato dal punto di vista informatico; abbiamo però detto che l'origine di questo concetto è l'Algebra Lineare, per cui analizziamo il concetto di tipo dal punto di vista matematico.

Come abbiamo già detto, per risolvere in maniera automatica un problema occorre specificare l'algoritmo, implementarlo nel linguaggio di programmazione prescelto, e altre attività complesse, ciascuna delle quali suddividibile in più fasi. In particolare, per specificare l'algoritmo occorre sia specificare i dati necessari con le loro proprietà, sia le operazioni da compiere sui dati per raggiungere la soluzione. Allo stesso modo, l'implementazione implica rappresentare nel linguaggio di programmazione scelto sia i dati sia le operazioni da compiere su di essi, definendo una corrispondenza tra la specifica e ciò che il linguaggio di programmazione permette (una corrispondenza tra tipo di dato *astratto* e tipo di dato *concreto*).

Per i vari tipi primitivi questa corrispondenza è nota e ovvia, ma per quei tipi derivati come è possibile definire questa corrispondenza?

Occorre, dunque, definire in maniera precisa cosa si intende per tipo di dato astratto.

**Definizione 5.2** (Tipo di dato astratto). Per **tipo di dato astratto** si intende un ente matematico definito come:

- un insieme di oggetti (il *dominio*);
- un insieme di possibili *operazioni* (*funzioni* o *predicati*);
- un insieme di costanti.

Le funzioni o i predicati specificati nella definizione del tipo astratto vengono definite anche **operazioni primitive**, perché a partire da esse è possibile definire altre funzioni o predicati.

### Rappresentazione di un tipo astratto

Una volta definito il concetto di tipo di dato astratto, possiamo definire come rappresentarlo nel contesto del linguaggio di programmazione prescelto.

**Definizione 5.3** (Rappresentazione di tipo). Una **rappresentazione** di un tipo astratto  $T$  è definito tramite una tripla  $\langle T', rapp, op \rangle$ , ove:

- $T'$  è il tipo con cui si rappresenta il tipo astratto  $T$ ;
- $rapp$  è una corrispondenza tra i valori di  $T$  e di  $T'$ ;
- $op$  è una funzione che associa ogni operazione primitiva definita su  $T$  ad una operazione definitiva su  $T'$ .

Il tipo  $T'$  non è necessariamente concreto, perché in generale si possono avere più rappresentazioni di un tipo astratto, ciascuna delle quali deve rispettare le caratteristiche di correttezza ed efficacia fissate in precedenza. Ad esempio, una rappresentazione si dice *corretta* se per ogni valore di  $T$  esiste almeno un valore di  $T'$  che la rappresenta e se ogni operazione primitiva di  $T$  è correttamente realizzata dalla corrispondente operazione definita per  $T'$ .

## 5.2 Descrizione dei tipi derivati

Abbiamo precedentemente definito cosa sia un tipo derivato e le varie classificazioni di un tipo derivato. Vediamo ora di descrivere questi tipi.

### 5.2.1 I puntatori e i riferimenti

In molti casi, specie quando l'allocazione della memoria è dinamica come vedremo più avanti, è più comodo e utile conoscere l'indirizzo di memoria di una variabile piuttosto che memorizzare la variabile stessa. Per questi casi, alcuni linguaggi di programmazione offrono i **puntatori**, ovvero variabili il cui valore è un indirizzo di memoria di un'altra variabile (Figura 5.1).

L'operatore con cui, dato un puntatore, si accede alla variabile puntata viene

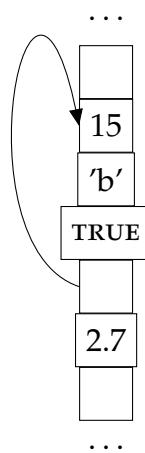


Figura 5.1: Puntatore

detto *operatore di dereferenziazione* (*dereferencing*). Molti linguaggi offrono anche un operatore "inverso", spesso detto operatore *indirizzo-di*, che data una variabile consente di ricavarne l'indirizzo. Un insieme esteso di operazioni sui puntatori viene fornito dai linguaggi dotati di aritmetica dei puntatori.

L'uso di puntatori è spesso necessario per costruire strutture dati complesse e dalla forma non prevedibile a priori e/o variabile nel tempo come grafi, alberi, liste e così via. Inoltre, i puntatori possono essere usati per realizzare il passaggio di parametri per riferimento nei linguaggi che non lo offrono come meccanismo nativo.

Se usati in modo indiscriminato, tuttavia, i puntatori possono portare allo sviluppo di software molto complesso e, soprattutto, condurre a errori di programmazione difficili da individuare. Per questo motivo alcuni linguaggi, tra cui Java, tentano di limitarne l'uso.

Alcuni linguaggi forniscono un meccanismo simile ai puntatori, ma caratterizzato da dereferenziazione implicita; le variabili di questo tipo sono dette **riferimenti**.

Le caratteristiche dei riferimenti sono diverse nei diversi linguaggi, ma in genere hanno una caratteristica comune: sintatticamente, i tipi riferimento non richiedono l'uso di un operatore di dereferenziazione, e non prevedono un operatore "indirizzo di"; di conseguenza, sui riferimenti non è possibile l'aritmetica dei puntatori.

Sia per i puntatori sia per i riferimenti, esiste un valore speciale, detto **NULL** o **NIL**, che indica un puntatore che non punta a nessuna locazione di memoria.

### 5.2.2 Gli array

Pensiamo ad un elenco alfabetico di persone, nel quale ad ogni cognome e nome viene associato un numero intero che corrisponde alla posizione che la persona occupa nell'elenco. Questo elenco è possibile rappresentarlo tramite una tabella a due colonne, in cui la prima colonna è data dalla posizione e la seconda colonna contiene il nome e cognome della persona.

Questa rappresentazione posizionale prende il nome di *array*.

**Definizione 5.4 (Array).** Si definisce **array** (o *vettore*) un insieme di variabili omogenee, ovvero dello stesso tipo dati, primitivo o derivato, a cui è possibile accedere in modo posizionale.

Le diverse variabili che compongono un array sono chiamate **elementi** (o *componenti*) e vengono individuate tramite l'identificatore e l'*indice posizionale*.

Il numero delle componenti di un array, ovvero la sua **dimensione**, è prefissato. I valori dell'indice posizionale, invece, cambiano a seconda del linguaggio di programmazione: si hanno indici posizionali compresi tra  $O$  e  $N - 1$  (dove  $N$  è la dimensione dell'array) e indici posizionali compresi tra 1 e  $N$  (Figura 5.2).

Dal punto di vista tecnico, possiamo considerare l'array come un puntatore ad un'area di memoria in cui i dati sono collocati sequenzialmente.

L'accesso alle componenti dell'array avviene tramite l'operatore `[]` (chiamato operatore di indicizzazione o *subscripting*).

a	b	c	f	r	t
0	1	2	3	4	5

Figura 5.2: Array

### La ricerca di elementi in un array

Uno dei problemi più comuni che si possano avere usando un array è quello della *ricerca di un elemento*, in cui si restituisce l'indice posizionale in cui si trova l'elemento oppure  $-1$ .

In questo tipo di problemi, in ogni momento si hanno due insiemi:

- un insieme composto dagli elementi già esaminati;
- un insieme composto dagli elementi da esaminare ancora (*spazio di ricerca*).

È possibile individuare due principali metodologie di ricerca:

- la **ricerca sequenziale**, in cui si analizzano gli elementi uno dopo l'altro fino a trovare l'elemento ricercato;
- la **ricerca binaria**, in cui l'array è ordinato e si parte dal centro dividendo lo spazio di ricerca in due fin quando non si trova l'elemento.

### La fusione di sequenze ordinate

Un altro problema comune nell'uso di array è quello della *fusione di sequenze ordinate*, in cui la nuova sequenza è ordinata a sua volta e la sua lunghezza è pari alla somma delle lunghezze delle singole sequenze.

### Array multidimensionali

In molte applicazioni le informazioni devono essere rappresentate da collezioni di variabili omogenee rappresentate come *strutture multidimensionali*; nei linguaggi di programmazione, queste strutture multidimensionali sono gli *array di array*, cioè array in cui gli elementi sono a loro volta degli array.

Per un array di array, l'accesso avviene usando come indici una tupla di numeri naturali.

Il più piccolo array di array è l'*array bidimensionale* (o *matrice*). Essendo l'array più esterno quello che rappresenta le righe della matrice, la scansione degli elementi deve avvenire sempre a partire dalle righe.

In generale, per array multidimensionali la scansione degli elementi deve avvenire sempre dall'array più esterno.

### 5.2.3 I record

Un oggetto del mondo reale necessita di molti dati per essere descritto in modo adeguato. Si pensi ad un libro: per descriverlo occorre avere il titolo, il nome degli autori, l'anno e il luogo di edizione, la casa editrice, ecc. Per questo motivo, è comodo aggregare tutte queste informazioni in un'unica struttura che le possa contenere.

Questa struttura è diversa dall'array, perché i dati al suo interno sono eterogenei (ovvero sono di diversi tipi), e prende il nome di **record** (o *oggetto*). Ciascuna di queste informazioni (i *campi* del record) può essere acceduta in modo indipendente specificandone il nome: in molti linguaggi, specialmente quelli derivati dal C, questa operazione è specificata mediante l'operatore . (punto).

### 5.2.4 Tipi generici

In alcuni linguaggi è possibile specificare **tipi generici** (detti anche *parametrici* o *template*), che sono in effetti costruttori di tipo definiti dal programmatore. Essi non possono essere direttamente usati nei programmi, ma devono essere applicati ad uno o più tipi esistenti (che sono quindi i parametri del costruttore) per generare nuovi tipi di dato.

La relazione tra costruttore di tipo e tipo concreto può quindi essere vista in analogia a quella tra classe ed oggetto nella programmazione orientata agli oggetti.

## 5.3 Strutture collegate

Sia l'array sia il record rappresentano una *struttura dati statica*, in cui la dimensione si stabilisce al momento della creazione e i cui elementi sono memorizzati sequenzialmente nella memoria centrale.

In particolare, se non tutte le celle dell'array sono valorizzate, si ha un notevole spreco di memoria; inoltre, se si vogliono aggiungere elementi oltre la dimensione stabilita occorre creare un nuovo array con dimensioni maggiori.

Per evitare sovradimensionamenti o sottostime, con gestioni complesse delle locazioni di memoria, è possibile definire strutture la cui gestione della memoria è *dinamica*, cioè le risorse di memoria vengono allocate o rilasciate su richiesta. Occorre, dunque, svincolare la sequenzialità fisica degli elementi, tipica degli array, da quella logica, necessaria per mantenere la sequenza degli elementi. Queste strutture sono note come **strutture collegate**.

### 5.3.1 La lista

Il primo tipo di struttura collegata che analizziamo è la **lista**, una struttura collegata lineare in cui ogni elemento ha un solo successore, non necessariamente allocato consecutivamente in memoria.

L'elemento base di una lista è il *nodo*, una struttura composta dall'informazione di interesse (*info*) e dal riferimento al successivo elemento (*next*). Le operazioni più comuni effettuabili sulle liste sono:

- modifica, inserimento, eliminazione di un nodo;
- accesso ad un nodo;
- ricerca di una informazione;
- verifica se la lista sia vuota;
- calcolo della lunghezza.

Le differenze principali tra gli array e le liste sono le seguenti:

- gli array sono un unico oggetto, i cui elementi sono consecutivamente allocati in memoria, mentre le liste, essendo formate da più oggetti, sono allocate in maniera discontinua;
- essendo allocate in maniera discontinua, le operazioni di inserimento e cancellazione di elementi nelle liste sono più semplici;
- l'accesso agli elementi degli array avviene in maniera diretta, grazie agli indici, mentre l'accesso agli elementi di una lista avviene in maniera sequenziale.

Volendolo definire in maniera formale, il tipo astratto *Lista* è una tripla  $\langle S, F, C \rangle$ , dove:

- $S = \{\text{Lista}, V, \text{Boolean}\}$ , con *Lista* il *dominio di interesse* e  $V$  il dominio di sostegno;
- $F = \{\text{listaVuota}, \text{vuota}, \text{cons}, \text{car}, \text{cdr}\}$ ;
- $C = \{\text{Lista\_Vuota}\}$ .

Le operazioni definite in  $F$  sono:

- $\text{listaVuota}: () \rightarrow \text{Lista}$ , che crea una lista vuota;
- $\text{vuota}: \text{Lista} \rightarrow \text{Boolean}$ , che verifica se una data lista sia vuota;
- $\text{cons}: V \times \text{Lista} \rightarrow \text{Lista}$ , che crea una lista mettendo un elemento di  $V$  in testa alla lista data in input;
- $\text{car}: \text{Lista} \rightarrow V$ , che restituisce il primo elemento di una lista data;
- $\text{cdr}: \text{Lista} \rightarrow \text{Lista}$ , che restituisce la lista iniziale privata del primo elemento.

### 5.3.2 La pila

Un secondo tipo di struttura collegata che analizziamo è la **pila**, detta in inglese *stack*. La pila rappresenta un elenco di dati avente la caratteristica peculiare di permettere l'inserimento di nuovi elementi e l'estrazione degli elementi introdotti da un'unica estremità; ciò si definisce *politica LIFO* (*Last In First Out*).

Anche per la pila, l'elemento base è il *nodo*.

La sua definizione formale è una tripla  $\langle S, F, C \rangle$ , dove:

- $S = \{\text{Pila}, V, \text{Boolean}\}$ , con Pila il *dominio di interesse* e  $V$  il dominio di sostegno;
- $F = \{\text{pilaVuota}, \text{vuota}, \text{push}, \text{pop}, \text{top}\}$ ;
- $C = \{\text{Pila_Vuota}\}$ .

Le operazioni definite in  $F$  sono:

- $\text{pilaVuota}: () \rightarrow \text{Pila}$ , che crea una pila vuota;
- $\text{vuota}: \text{Pila} \rightarrow \text{Boolean}$ , che verifica se una data pila sia vuota;
- $\text{push}: V \times \text{Pila} \rightarrow \text{Pila}$ , che crea una pila mettendo un elemento di  $V$  in cima alla pila data in input;
- $\text{top}: \text{Pila} \rightarrow V$ , che restituisce il primo elemento di una pila data;
- $\text{pop}: \text{Pila} \rightarrow \text{Pila}$ , che restituisce la pila iniziale privata dell'elemento in cima.

### 5.3.3 La coda

Il terzo tipo di struttura collegata che analizziamo è la **coda**, detta in inglese *queue*. La coda rappresenta un elenco di dati avente la caratteristica peculiare di permettere l'inserimento da una estremità e l'estrazione dall'altra estremità degli elementi, rappresentati tramite *nodi*; ciò si definisce *politica FIFO* (*First In First Out*).

La sua definizione formale è una tripla  $\langle S, F, C \rangle$ , dove:

- $S = \{\text{Coda}, V, \text{Boolean}\}$ , con Coda il *dominio di interesse* e  $V$  il dominio di sostegno;
- $F = \{\text{codaVuota}, \text{vuota}, \text{inCoda}, \text{outCoda}, \text{primo}\}$ ;
- $C = \{\text{Coda_Vuota}\}$ .

Le operazioni definite in  $F$  sono:

- $\text{codaVuota}: () \rightarrow \text{Coda}$ , che crea una coda vuota;
- $\text{vuota}: \text{Coda} \rightarrow \text{Boolean}$ , che verifica se una coda data in input sia vuota;

- $\text{inCoda}: V \times \text{Coda} \rightarrow \text{Coda}$ , che crea una coda mettendo un elemento di  $V$  in coda (cioè alla fine) alla coda data in input;
- $\text{primo}: \text{Coda} \rightarrow V$ , che restituisce il primo elemento di una coda data;
- $\text{outCoda}: \text{Coda} \rightarrow \text{Coda}$ , che restituisce la coda iniziale privata dell'elemento in testa.

## 5.4 Espressioni

Finora abbiamo parlato di variabili e costanti, ma lavorare con singole variabili e singole costanti non è efficiente. Per questo motivo parliamo di *espressioni*.

**Definizione 5.5** (Espressione). Una **espressione** viene formata mediante la composizione di *operatori* (i simboli usati per denotare operazioni) e *operandi* (gli argomenti delle operazioni) e viene caratterizzata da un tipo dati.

Le espressioni più semplici sono i letterali, le variaibl e le costanti, ma sono espressioni anche le invocazioni di sottoprogrammi e l'accesso agli elementi di un record. Queste espressioni semplici sono componibili usando i vari operatori disponibili per quella tipologia di dati.

Le espressioni vengono valutate calcolando il calore di ciascun operando e applicando man mano i vari operatori seguendo il seguente ordine:

1. precedenza degli operatori;
2. associatività degli operatori;
3. parentesi.



# Capitolo 6

## Correttezza e complessità

Finora abbiamo visto come risolvere un problema tramite un programma e come scrivere un programma. Rimangono però ancora molte domande per cui è necessario ottenere una risposta.

Come possiamo essere sicuri che il programma che abbiamo scritto fornisca il comportamento che abbiamo pensato?

Come possiamo verificare se un algoritmo è migliore di un altro? E cosa si intende per "migliore"?

In questo capitolo analizzeremo il concetto di *correttezza* di un programma e quello di *complessità* di un algoritmo, rispondendo a tutte queste domande.

### 6.1 Cosa si intende per correttezza?

La **correttezza** è una delle qualità più importanti di un programma. Possiamo dire, in maniera informale, che un programma è *corretto* se si comporta come è stato pensato.

Il fatto che sia una delle qualità più importanti è abbastanza ovvio: un programma non corretto può causare danni più o meno gravi a seconda del malfunzionamento che si manifesta (danni economici, materiali, alle persone, catastrofici, ecc.).

Quando si parla di correttezza, ciò che viene subito in mente è la ricerca, l'identificazione e la correzione degli *errori* di programmazione. Possiamo formalizzare questi concetti nelle seguenti attività:

- la *verifica di correttezza*, che ha lo scopo di verificare se una parte di un programma contiene o meno degli errori;
- l'*identificazione degli errori*, che permette di individuare quanti e quali sono gli errori presenti in un programma;
- la *correzione degli errori*, che ha lo scopo di sostituire la parte di codice errata con del codice privo di errori.

Abbiamo accennato precedentemente al concetto di malfunzionamento e di errore, senza mai definire questi concetti in maniera precisa. Forniamo ora la definizione di questi concetti.

**Definizione 6.1** (Malfunzionamento). Un **malfunzionamento** di un programma è una discrepanza tra il funzionamento effettivo del programma e quello corretto.

**Definizione 6.2** (Programma non corretto). Un programma si dice **non corretto** (o *errato*) se durante la sua esecuzione possono avvenire dei malfunzionamenti.

**Definizione 6.3** (Errore). Un **errore** (o *difetto*) è la causa di un malfunzionamento.

In base a queste definizioni, possiamo dire che un malfunzionamento è relativo al comportamento *esterno* del programma, mentre un errore è relativo al comportamento *interno*. Inoltre, la definizione di non correttezza è una definizione probabilistica, ovvero si basa sulla probabilità che un malfunzionamento accada, ma non sul fatto che accada realmente.

Per formalizzare tutti questi concetti è necessario ricorrere alla nozione di *specifica* di un programma.

**Definizione 6.4** (Specificità). Sia dato un programma  $P$ , che riceva in ingresso alcuni dati che devono soddisfare specifiche proprietà e condizioni, note come **pre-condizioni**, e fornisca in uscita altri dati i quali devono soddisfare alcune proprietà, note come **post-condizioni**.

Se l'esecuzione delle operazioni previste dal programma  $P$  avviene in modo non ambiguo e in un tempo finito, allora l'insieme di pre-condizioni, programma  $P$  e post-condizioni viene definito come la **specificità** del programma.

La specifica di un programma fa parte della sua documentazione, quindi può essere descritta tramite commenti nel codice sorgente stesso.

Sfruttando il concetto di specifica, possiamo dire che un programma è *parzialmente corretto* se la sua specifica è soddisfatta per ogni insieme di dati che soddisfino la pre-condizione e se il programma termina producendo dati che soddisfino la post-condizione.

Possiamo dunque affermare che chi utilizza un programma ha la responsabilità di fornire in ingresso al programma dati che soddisfino la precondizione, mentre è responsabilità del programmatore far sì che quando vengono forniti in ingresso dati che soddisfino la precondizione, il programma termini e produca un risultato che soddisfi la post-condizione.

Il concetto di specifica è facilmente estendibile anche ai sottoprogrammi, con minime variazioni. In questo caso:

- la pre-consizione di un sottoprogramma è l'insieme delle proprietà che si assumono verificate quando il sottoprogramma viene chiamato;

- la post-condizione di un sottoprogramma è l'insieme delle proprietà che devono risultare verificate al termine dell'esecuzione del sottoprogramma stesso.

### 6.1.1 Errori nella programmazione

Come abbiamo detto, durante la scrittura di un programma, qualsiasi sia il paradigma usato, è possibile commettere numerosi errori in vari modi.

Una prima classificazione di questi errori è:

- errori riconosciuti dal compilatore/interprete;
- errori non riconosciuti dal compilatore/interprete.

Alternativamente, una seconda classificazione dei possibili errori di programmazione è:

- frasi non corrette nel linguaggio di programmazione (noti come *errori grammaticali*);
- frasi corrette nel linguaggio di programmazione ma con un significato diverso da quello prefissato (*errori semanticici*);
- frasi corrette nel linguaggio di programmazione, anche dal punto di vista semantico, ma che portano ad un risultato non corretto (*errori logici*).

Gli errori grammaticali sono sempre riconosciuti dal compilatore/interprete, insieme ad alcuni errori semanticici (quelli di *semantica statica*); altri tipi di errori semanticici (quelli di *semantica dinamica*) e quelli logici non sono riconosciuti dal compilatore/interprete.

Analizziamo nel dettaglio questi errori.

#### Errori grammaticali

Gli **errori grammaticali** (noti anche come *errori semanticici*) sono legati all'uso non corretto delle regole grammaticali del linguaggio. Alcuni esempi di questo tipo di errori sono l'uso errato di separatori, la scrittura errata di parole chiave del linguaggio e la scrittura di espressioni mal formate.

Come abbiamo detto, questo tipo di errori viene sempre riconosciuto dal compilatore/interprete, che li segnala mediante opportuni messaggi di errore. Il riconoscimento di questi errori da parte del compilatore/interprete è purtroppo imperfetta, perché la correzione suggerita può provocare altri errori. Sta al programmatore interpretare correttamente le segnalazioni di errori.

#### Errori di semantica statica

Gli **errori di semantica statica** sono quel tipo di errori relativi a istruzioni ben formate grammaticalmente ma errate perché prive di significato, i quali vengono riconosciuti dal compilatore/interprete. Alcuni esempi di questo tipo di errori

sono l'uso di una variabile non dichiarata, l'accesso ad una variabile non inizializzata o errori di tipo.

Non tutti i linguaggi di programmazione riconoscono questo tipo di errori durante la compilazione o l'interpretazione.

### Errori di semantica dinamica

Gli **errori di semantica dinamica** sono quegli errori relativi ad istruzioni che hanno significato in alcuni casi, ma non sempre. Questo tipo di errori sono anche detti *errori a tempo di esecuzione* (o *runtime error*) proprio perché si verificano durante l'esecuzione del programma, ma anche *difetti* (*bug*).

A seconda del linguaggio di programmazione, questi errori vengono segnalati sotto forma di *eccezioni* o *errori*, strutture dati particolari che possono causare la terminazione anormale del programma.

### Errori logici

Gli **errori logici** non sono riconosciuti dal compilatore e possono provocare la produzione di risultati non corretti, oppure la non terminazione del programma stesso. Visto che non si manifestano in modo evidente, questo tipo di errori sono peggiori degli errori di semantica dinamica.

Solamente applicando tecniche di verifica è possibile mitigare l'incidenza di questi errori.

#### 6.1.2 Verifica di correttezza

Per verificare la correttezza di un programma o di un suo sottoprogramma è possibile seguire due approcci.

Nell'**approccio formale** si tenta di dimostrare la correttezza o la non correttezza di un programma (di un sottoprogramma) usando tecniche formali, simili alla dimostrazione dei teoremi in Matematica. Questo approccio consiste nei seguenti passi:

- determinare la pre-condizione e la post-condizione;
- verificare se la specifica è soddisfatta;
- stabilire se il programma termina per ogni possibile insieme di dati di ingresso che soddisfi la pre-condizione.

L'approccio formale può essere molto complesso e lungo.

Per questo motivo esiste anche un secondo approccio, noto anche come **metodo sperimentale di verifica della correttezza** o (**test di correttezza**), orientato alla verifica sperimentale del funzionalmento del programma (o del sottoprogramma).

Questo metodo consiste nei seguenti passi:

1. si determina un insieme di dati di input  $I$ ;

2. si esegue il programma  $P$  con i dati  $I$ ;
3. si verificano i risultati ottenuti:
  - a) se l'esecuzione non termina in un lasso di tempo ragionevole o se i risultati ottenuti non sono quelli attesi, allora il programma non è corretto;
  - b) se non sono stati rilevati errori, bisogna stabilire se è necessario effettuare nuove prove (e determinare un nuovo insieme  $I$ ) oppure fermarsi

Come si può vedere, le possibili combinazioni di dati in ingresso sono infinite, o comunque non è possibile testarle tutte; usare un sottoinsieme di dati di ingresso, dunque, non esclude l'esistenza di una combinazione erronea, ma ne riduce la probabilità.

Le varie tecniche di verifica di correttezza si classificano in base al tipo di dati usati per stimolare il metodo. Abbiamo dunque:

- **test a scatola nera**, in cui l'insieme dei dati in ingresso viene scelto solamente basandosi sulla specifica del metodo;
- **test a scatola trasparente**, in cui l'insieme dei dati in ingresso viene scelto anche in base ai dettagli implementativi del metodo.

In entrambi i casi, la scelta dell'insieme dei dati di ingresso è una attività che deve essere condotta e pianificata durante la fase di progettazione del programma.

I test a scatola nera partizionano l'insieme dei possibili dati di ingresso in vari sottoinsiemi, detti *insiemi di equivalenza*, basandosi sulla seguente ipotesi: se il metodo si comporta correttamente per una combinazione di dati di ingresso  $X$  scelta nell'insieme di equivalenza  $C_X$ , allora il metodo si comporta correttamente anche per ogni altra combinazione scelta nello stesso insieme.

Questa ipotesi condiziona la scelta dei dati da usare in ingresso: si cercheranno insiemi di dati *normali* per il metodo e insiemi di dati *particolari* (casi limite, come una sequenza vuota o che contiene un solo elemento).

I test a scatola trasparente, invece, sono basati sull'ipotesi che, per verificare la correttezza di un metodo, bisogna far in modo di eseguire ciascuna istruzione del metodo stesso almeno una volta.

## 6.2 Complessità di un programma

Un'altra qualità importante di un programma è la sua *efficienza*. In maniera intuitiva, un programma tanto più è efficiente quanto minori sono le risorse di calcolo necessarie per la sua esecuzione.

Ma quali sono le risorse di calcolo importanti per decretare l'efficienza di un programma?

Generalmente, le risorse principali da valutare sono il *tempo di esecuzione* e la *quantità di memoria utilizzata*, ma taluni casi richiedono di analizzare anche la *quantità di dati traseriti da/su una memoria secondaria*, o la *quantità di traffico generato su una rete di calcolatori*.

Per stabilire se un programma è più efficiente di un altro, ovvero per stabilire quale sia l'algoritmo migliore, occorre dunque stabilire il *costo computazionale* dell'algoritmo stesso, analizzandolo sia dal punto di vista del tempo di esecuzione (si parla di **complessità temporale**) sia dal punto di vista della memoria occupata (si parla di **complessità spaziale**). Questa attività richiede l'analisi di diversi fattori di un calcolatore reale, come l'hardware di cui è composto, il sistema operativo usato, il carico di lavoro del processore in quel momento, la configurazione dei dati di ingresso per la singola esecuzione e la loro dimensione, se si volesse effettuare una *misura sperimentale* della complessità del calcolatore.

La sperimentazione, però, non può per sua natura essere oggettiva, poiché in ogni momento possono cambiare vari fattori. Per questo motivo, è utile avere un *modello di costo approssimato*, che permetta di astrarre i dettagli implementativi concentrando sul costo intrinseco del programma.

Il modello di costo si basa su un *modello di calcolo astratto*, in modo da rendere indipendente il costo dal calcolatore, svincolandosi dalle possibili configurazioni dei dati in ingresso ma basandosi su quelle sfavorevoli. Questo modello, inoltre, deve essere rappresentato come una funzione asintotica della dimensione dell'input.

Secondo questo modello:

- le istruzioni semplici hanno costo unitario;
- il costo delle istruzioni condizionali è dato dal costo della condizione più il costo del blocco relativo al valore della condizione;
- il costo delle istruzioni iterative è dato dal costo della condizione e dal costo del blocco di istruzioni che formano il corpo dell'iterazione, moltiplicato per il numero di ripetizioni;
- il blocco di istruzioni ha costo pari alla somma dei costi delle istruzioni che lo compongono.

L'*analisi del caso peggiore* dei dati in input permette di essere sicuri del comportamento dell'algoritmo in tutte le situazioni, poiché se l'algoritmo è sufficientemente efficiente nel caso peggiore allora lo sarà anche in tutti gli altri casi.

Non è l'unica analisi possibile per la definizione del modello di costo: è possibile analizzare anche il *caso migliore* (usando la configurazione dell'input più favorevole) oppure il *caso medio*.

### 6.2.1 Analisi asintotica della complessità

Come abbiamo detto, nel modello di costo, espresso in termini di funzioni della dimensione dell'input, è importante valutare il valore della funzione al crescere della dimensione dell'input. Per questo motivo è importante analizzare l'andamento asintotico della funzione.

Una modalità per esprimere l'andamento asintotico è la cosiddetta **notazione  $O$ -grande**; secondo questa notazione, una funzione  $f(n)$  si dice  $O(g(n))$  se e sono se esistono due costanti positive  $c$  e  $n_0$  tali che  $f(n) \leq c g(n) \forall n \geq n_0$ .

In pratica, la notazione  $O$ -grande definisce verso quale funzione la nostra funzione di costo tende asintoticamente, per cui si cerca di definire una approssimazione ottima dell'andamento.

Usando questa notazione, possiamo dire che un algoritmo (o un programma) ha costo (o *complessità*)  $O(g(n))$  se la quantità di tempo per eseguire un algoritmo su un input di dimensione  $n$  è nel caso peggiore  $O(g(n))$ .



# Capitolo 7

## Gestione della memoria e ricorsione

Nel Capitolo 6 abbiamo dichiarato che la quantità di memoria utilizzata da un programma è una delle misurazioni effettuate per determinarne l'efficienza. In effetti, la memoria centrale è sempre stata una risorsa limitata, per cui la sua gestione per un programma è di vitale importanza.

L'Informatica deriva dalla Matematica, come abbiamo detto. Applicando alcuni principi matematici ai nostri algoritmi, i programmi che ne derivano possono incrementare la quantità di memoria centrale usata in maniera rapida, fino a consumarla tutta.

In questo capitolo vedremo come i linguaggi di programmazione vedono e gestiscono la memoria centrale, per poi parlare della *ricorsione*, tecnica di programmazione basata sul principio di induzione.

### 7.1 Gestione della memoria a runtime

La gestione della memoria centrale viene effettuata dal sistema operativo, che la partiziona e che assegna ciascuna parte ad un processo diverso. In ogni parte, il sistema operativo definisce tre aree e le comunica al processo stesso: l'*area codice*, l'*heap* e lo *stack* (Figura 7.1).

L'**area codice** è di solito posizionata all'inizio della parte di memoria centrale allocata (dall'indirizzo più basso). Al suo interno viene memorizzato il codice sorgente del programma o la sua versione compilata (nel caso si usi un linguaggio di programmazione interpretato o compilato), insieme a variabili globali e variabili statiche.

In aggiunta, l'area codice contiene anche le variabili staticamente allocate che ancora non sono state valorizzate; per questo tipo di variabili è possibile identificare una porzione dell'area codice denominata *simbolo di inizio blocco* (in inglese *Block Starting Symbol*, BSS).

L'**heap** è l'area di memoria collocata successivamente all'area codice. La sua dimensione è dinamica, ovvero cresce e decresce durante l'esecuzione del processo.

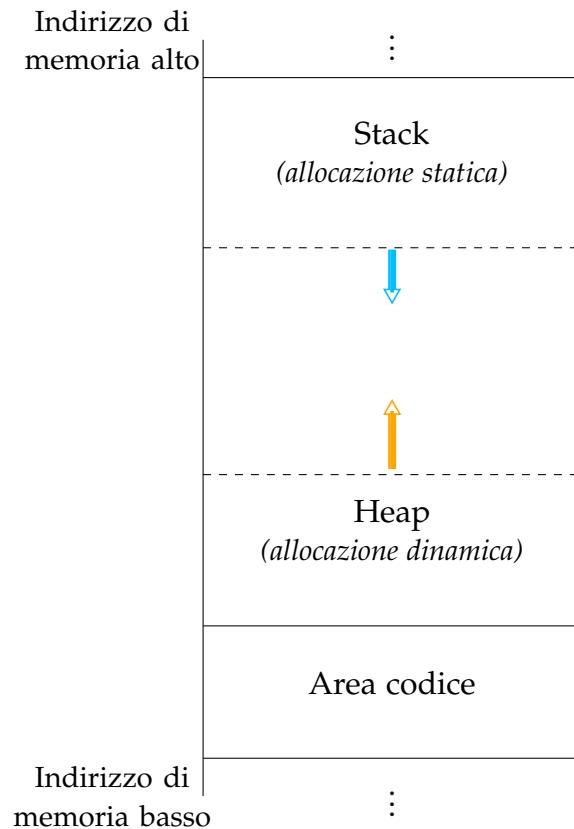


Figura 7.1: La memoria centrale per un programma

La gestione di questa area è completamente a carico del programma, in maniera diretta o indiretta.

In alcuni linguaggi di programmazione, la gestione è manuale e il programmatore deve allocare e deallocare personalmente le variabili; una cattiva gestione manuale provoca la cosiddetta *perdita di memoria* (*memory leak*), ovvero provoca che parti di memoria risultino occupate anche se non sono più utilizzate e raggiungibili dal programma stesso.

In altri linguaggi di programmazione, come ad esempio i linguaggi orientati agli oggetti (i quali usano l'heap per memorizzare i vari oggetti del programma), la gestione è effettuata direttamente dal programma stesso, attraverso un meccanismo noto come **garbage collector**. L'uso del garbage collector permette di ridurre la probabilità di perdita di memoria, ma non la elimina del tutto; inoltre, inoltre, il garbage collector utilizza risorse di calcolo del calcolatore, levandole al programma.

Lo **stack**, detta anche *pila dei record di attivazione*, è collocata all'altro opposto dell'area codice (dall'indirizzo più alto). La sua gestione è effettuata direttamente dal processore tramite un meccanismo LIFO.

Lo stack contiene strutture particolari denominate **record di attivazione** (in inglese *frame*), che collezionano i dati del programma e di ciascun sottoprogramma. In particolare, questi dati sono:

- le locazioni di memoria per i parametri formali;
- le locazioni di memoria per le eventuali variabili locali;
- il valore di ritorno dell'invocazione della funzione;
- una locazione di memoria per l'*indirizzo di ritorno*, cioè l'indirizzo della successiva istruzione da eseguire nella funzione chiamante.

## 7.2 La ricorsione

Finora abbiamo visto tecniche di programmazione sequenziali. Un'altra tecnica di programmazione usata normalmente è la *ricorsione*.

La **ricorsione** è quella tecnica di programmazione in cui un sottoprogramma, direttamente o indirettamente, può invocare se stesso. Questa tecnica si può applicare anche alla definizione di tipi, permettendo di definirli in termini di se stesso.

La ricorsione è una tecnica basata su definizioni matematiche. Vediamo ora queste basi.

### 7.2.1 Definizioni induttive in Matematica

La base della ricorsione è il *principio di induzione matematica*, il quale afferma che se  $P$  è una proprietà che vale per il numero 0, e se  $P(n) \Rightarrow P(n+1)$  per ogni  $n \in \mathbb{N}$ , allora  $P(n)$  vale per ogni  $n \in \mathbb{N}$ .

Volendo applicare questo principio ai concetti di insieme e funzione, otteniamo due nuovi strumenti matematici.

**Definizione 7.1** (Insieme definito in modo induttivo). Un insieme  $A$  si dice **definito in modo induttivo** se è definito in termini di se stesso.

L'esempio pratico di insieme definito in modo induttivo è quello dei numeri naturali, in cui:

- 0 è un numero naturale;
- se  $n$  è un numero naturale, allora anche  $n + 1$  lo è.

Questo modo di definire gli insiemi è detto *intensionale*, poiché si forniscano le regole per calcolare tutti gli elementi dell'insieme. In particolare, nella definizione induttiva di un insieme si possono sempre identificare:

- uno o più *casi base*, ciascuno dei quali descrive l'appartenenza di alcuni elementi all'insieme in modo diretto;
- uno o più *casi induttivi*, ciascuno dei quali descrive l'appartenenza di alcuni elementi all'insieme in modo indiretto, ovvero in termini dell'appartenenza di altri elementi all'insieme stesso.

Allo stesso modo, è possibile definire una funzione in modo induttivo.

**Definizione 7.2** (Funzione definita in modo induttivo). Una funzione  $f$  si dice **definita in modo induttivo** se è definita in termini di se stessa.

Il dominio di una funzione definita in modo induttivo (detta anche *funzione definita per casi*) è di solito un insieme definito in modo induttivo.

Come si può intuire, anche per le funzioni definite in modo induttivo è possibile identificare uno o più casi base e uno o più casi induttivi.

Queste definizioni intensionali, per essere ben formulate, devono godere delle seguenti proprietà:

- non devono essere "circolari", ovvero che non devono mai definire il calcolo di una funzione su un argomento al calcolo della stessa funzione sullo stesso argomento;
- devono essere complete, cioè i ragionamenti devono sempre portare ad un caso base;
- devono essere non ambigue, cioè ad ogni passo del ragionamento si deve poter scegliere uno ed un solo caso, che sia base o induttivo.

### 7.2.2 Tipi di dato ricorsivi

Come abbiamo detto nel Capitolo 5, un tipo è descritto dal suo dominio e dalle operazioni che si possono compiere su di esso. Se il dominio è un insieme definito in modo induttivo, allora il tipo descritto da quel dominio è un **tipo ricorsivo**.

I tipi ricorsivi sono principalmente tipi derivati, per cui è possibile descrivere uno o più costruttori che accettano come parametro una istanza del tipo stesso. Per evitare una ricorsione infinita, occorre definire in maniera chiara uno o più casi base, ovvero uno o più costruttori che non hanno come parametri una istanza del tipo stesso.

Esempi di tipi ricorsivi (o più correttamente di tipi che possono essere definiti anche in maniera ricorsiva) sono la lista, la pila e la coda. In particolare, nel tipo Lista l'operazione `listaVuota` rappresenta il costruttore base, mentre l'operazione `cons` rappresenta il costruttore induttivo.

I tipi ricorsivi sono molto usati nei linguaggi di programmazione funzionali.

### 7.2.3 Sottoprogrammi ricorsivi

Ora che abbiamo definito in cosa consiste la ricorsione, possiamo iniziare a definire in cosa consiste e come scrivere un sottoprogramma ricorsivo.

**Definizione 7.3** (Sottoprogramma ricorsivo). Un **sottoprogramma ricorsivo** è un sottoprogramma che, direttamente o indirettamente, può invocare se stesso.

Il classico esempio di sottoprogramma ricorsivo è quello del calcolo del fattoriale. Come sappiamo, la funzione fattoriale si può definire come

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

Questa definizione è facilmente trasponibile in qualsiasi linguaggio di programmazione, definendo così il corpo del sottoprogramma.

Ogni sottoprogramma ricorsivo segue una struttura.

1. Il corpo deve contenere la dichiarazione di una variabile che rappresenta ciò che deve essere calcolato ed ritornato al chiamante.
2. Il corpo contiene una istruzione condizionale per identificare il caso (base o induttivo) in cui ci si trova.
3. Ciascuna sequenza dei vari rami dell'istruzione condizionale assegna un valore alla variabile di uscita.
4. Il sottoprogramma termina restituendo il valore al chiamante.

Alcune funzioni implementate in un programma in maniera ricorsiva è possibile implementarle anche in maniera iterativa, evitando che ci siano chiamate alla funzione stessa all'interno del suo corpo.

L'esecuzione di un sottoprogramma ricorsivo viene gestita allo stesso modo dei sottoprogrammi normali, ovvero usando record di attivazione.

La particolarità sta nel fatto che per ogni invocazione ricorsiva del sottoprogramma viene creato un nuovo record di attivazione, che viene impilato nello stack. All'interno di ciascun record di attivazione, il sottoprogramma riesce ad accedere solamente alle variabili allocate nel record stesso, permettendo di isolare le varie chiamate.

Quando la ricorsione è molto ampia, purtroppo, lo stack può essere consumato facilmente, provocando la terminazione erronea del programma stesso.

### Progettazione di sottoprogrammi ricorsivi

Per progettare e scrivere un sottoprogramma ricorsivo è possibile seguire le seguenti linee guida.

1. Determinare i casi base.
2. Determinare i casi induttivi.
3. Verificare che sia i casi base sia quelli induttivi partizionano l'insieme dei possibili dati di ingresso del metodo.
4. Usare una istruzione condizionale per selezionare il caso corrente ed eseguire le azioni corrispondenti.

5. Dichiarare una variabile per memorizzare il risultato del metodo.

Talvolta può essere necessario scrivere un sottoprogramma aggiuntivo, per avviare la ricorsione.

# Capitolo 8

## Ordinamento

Analizziamo ora alcuni algoritmi usati per risolvere una classe di problemi ben nota: i *problemi di ordinamento*.

Come sappiamo dalla Matematica Discreta, data una sequenza  $\langle a_1, a_2, \dots, a_n \rangle$  di  $n$  elementi su cui insiste una *relazione d'ordine*, si dice che la sequenza è *totalmente ordinata* rispetto alla relazione d'ordine se per ogni  $i \in [1, n]$  vale la condizione  $a_i \leq a_j$  con  $j \geq i$ ; se la condizione non è sempre verificata, si dice che la sequenza è *parzialmente ordinata*.

I problemi di ordinamento consistono dunque nel generare una permutazione della sequenza tale che la sequenza sia totalmente ordinata.

### 8.1 Ordinamento per selezione

Un primo algoritmo per l'ordinamento è l'**ordinamento per selezione** (*selection sort*), che si basa sulla seguente strategia di risoluzione: finché la sequenza non è ordinata si seleziona l'elemento di valore minimo tra quelli ancora non ordinati e si dispone l'elemento nella posizione definitiva.

La strategia di risoluzione definisce, dunque, un meccanismo a fasi (chiamate *passate*) che separa la sequenza di due partizioni, quella degli elementi ordinati e quella degli elementi non ordinati. Questa strategia definisce un algoritmo con complessità temporale *quadratica* ( $O(n^2)$ ).

### 8.2 Ordinamento per inserzione

La strategia di risoluzione dell'algoritmo di **ordinamento per inserzione** (*insertion sort*) suddivide la sequenza in due insiemi:

- elementi relativamente ordinati (ordinati tra loro ma che non sono necessariamente collocati nelle loro posizioni definitive):
- elementi non relativamente ordinati.

Inizialmente si considera relativamente ordinato il primo elemento e si inizia con le passate, collocando il primo elemento non ordinato tra quelli ordinati. Anche questa strategia ha complessità *quadratica* ( $O(n^2)$ ).

### 8.3 Ordinamento a bolle

Un secondo algoritmo per i problemi di ordinamento è il cosiddetto **ordinamento a bolle** (*bubble sort*).

La strategia di risoluzione è simile a quella del *selection sort*, quindi è basata su passate successive, confronti e scambi. A differenza del *selection sort*, il confronto avviene tra tutte le coppie di elementi adiacenti non ordinati.

Anche questa strategia ha complessità temporale *quadratica* ( $O(n^2)$ ).

### 8.4 Ordinamento per fusione

La strategia di risoluzione dell'algoritmo di **ordinamento per fusione** (*merge sort*) è nota come *divide et impera*: si divide la sequenza in due sotto-sequenze e si ordinano le due sotto-sequenze, per poi fonderle in una sequenza ordinata. Questa è una strategia ricorsiva, in cui il passo base è la sequenza con un solo elemento, ordinata per definizione.

La complessità temporale di questo algoritmo è *quasi-lineare* ( $O(n \cdot \log n)$ ).

### 8.5 Ordinamento veloce

L'algoritmo di ordinamento più usato è noto come **ordinamento veloce** (*quick sort*). La strategia di risoluzione sfrutta il *pivot*, un elemento della sequenza da usare come discriminante nella separazione della sequenza in sotto-sequenze. La complessità di questo algoritmo è *quadratica* ( $O(n^2)$ ) nel caso peggiore, ma è *quasi-lineare* nel caso medio, più probabile.

# Bibliografia

- [1] Cabibbo, Luca: *Fondamenti di informatica — Oggetti e Java.* McGraw-Hill, 2004.
- [2] Lorenzi, Agostino e Daniele Rossi: *Le basi dell'Informatica — I fondamenti della programmazione.* Atlas, 2002.
- [3] Wikipedia: *Code segment* — Wikipedia, L'enciclopedia libera, 2019. [https://en.wikipedia.org/w/index.php?title=Code\\_segment&oldid=907667282](https://en.wikipedia.org/w/index.php?title=Code_segment&oldid=907667282), Online in data 01 gennaio 2021.
- [4] Wikipedia: *.bss* — Wikipedia, L'enciclopedia libera, 2020. <https://en.wikipedia.org/w/index.php?title=.bss&oldid=980737468>, Online in data 01 gennaio 2021.
- [5] Wikipedia: *Costruttore (informatica)* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Costruttore\\_\(informatica\)&oldid=116820696](http://it.wikipedia.org/w/index.php?title=Costruttore_(informatica)&oldid=116820696), Online in data 02 gennaio 2021.
- [6] Wikipedia: *Data segment* — Wikipedia, L'enciclopedia libera, 2020. [https://en.wikipedia.org/w/index.php?title=Data\\_segment&oldid=977894548](https://en.wikipedia.org/w/index.php?title=Data_segment&oldid=977894548), Online in data 01 gennaio 2021.
- [7] Wikipedia: *Distruttore (informatica)* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Distruttore\\_\(informatica\)&oldid=116820924](http://it.wikipedia.org/w/index.php?title=Distruttore_(informatica)&oldid=116820924), Online in data 02 gennaio 2021.
- [8] Wikipedia: *Garbage collection* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Garbage\\_collection&oldid=116821226](http://it.wikipedia.org/w/index.php?title=Garbage_collection&oldid=116821226), Online in data 01 gennaio 2021.
- [9] Wikipedia: *Informatica* — Wikipedia, L'enciclopedia libera, 2020. <http://it.wikipedia.org/w/index.php?title=Informatica&oldid=116211014>, Online; in data 11 novembre 2020].
- [10] Wikipedia: *Memory leak* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Memory\\_leak&oldid=117019763](http://it.wikipedia.org/w/index.php?title=Memory_leak&oldid=117019763), Online in data 01 gennaio 2021.
- [11] Wikipedia: *Storia dei sistemi operativi* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Storia\\_dei\\_sistemi\\_operativi&oldid=111696616](http://it.wikipedia.org/w/index.php?title=Storia_dei_sistemi_operativi&oldid=111696616), Online in data 14 novembre 2020.

- [12] Wikipedia: *Storia dell'informatica* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Storia\\_dell%27informatica&oldid=116381397](http://it.wikipedia.org/w/index.php?title=Storia_dell%27informatica&oldid=116381397), Online; in data 13 novembre 2020.
- [13] Wikipedia: *Tipo di dato* — Wikipedia, L'enciclopedia libera, 2020. [http://it.wikipedia.org/w/index.php?title=Tipo\\_di\\_dato&oldid=116821783](http://it.wikipedia.org/w/index.php?title=Tipo_di_dato&oldid=116821783), Online in data 24 dicembre 2020].

# Elenco delle figure

1.1	L'abaco nell'Antica Roma . . . . .	5
1.2	I bastoncini di Nepero (versione del XVIII secolo) . . . . .	6
1.3	La pascalina . . . . .	7
1.4	Un prototipo della calcolatrice di Leibniz . . . . .	7
1.5	Parte dell'Analytical Engine di Babbage . . . . .	8
1.6	Riproduzione dello Z1 . . . . .	11
1.7	L'ENIAC . . . . .	12
1.8	Varie tipologie di transistor . . . . .	13
1.9	IL PDP-1 . . . . .	14
1.10	Un modello di System/360 . . . . .	15
1.11	L'APPLE II . . . . .	16
1.12	L'IBM 5150, un esempio di PC-IBM . . . . .	17
2.1	Macchina di Von Neumann . . . . .	21
2.2	RAM e ROM . . . . .	24
2.3	Un esempio di disco magnetico . . . . .	25
2.4	Struttura delle superfici di una faccia . . . . .	25
2.5	Il ciclo <i>fetch-decode-execute</i> . . . . .	28
2.6	Gerarchia di un sistema operativo . . . . .	34
3.1	Diagramma di flusso per il problema del massimo comun divisore . .	41
5.1	Puntatore . . . . .	57
5.2	Array . . . . .	59
7.1	La memoria centrale per un programma . . . . .	74



# Indice analitico

- O-grande, Notazione, 71
- Abaco, 5
- Alfabeto, 45, 46
- ALGOL, 15
- Algoritmo, 39
  - Traccia, 41
- ALU, 26
- Applicazione, *vedi* Programma
- Area codice, 73
- Array, 58
  - Dimensione, 58
  - Elemento, 58
- ASCII, 30
- Assioma, 46
- Azione, 40
- Böhm-Iacopini, Teorema, 42
- Babbage, Charles, 8
- Backus, John, 14
- Bit, 22
- BNF, 47
- Boole, George, 9
- Bus, 21, 28
  - di I/O, 29
  - Locale, 29
- Byte, 22
- Calcolatore, 19
- Calcolatore elettronico, *vedi* Calcolatore
- CD-ROM, 25
- Central Processing Unit, *vedi* CPU
- COBOL, 15
- Codice, 3
- Colmar, Thomas de, 8
- Commento, 51
- Compilazione, 50
- Complessità
  - Spaziale, 70
  - Temporale, 70
- Correttezza, 65
  - Approccio formale, 68
  - Metodo sperimentale, 68
  - Verifica, 65
- Costante, 37
  - Identificatore, 38
- CPU, 21, 26
  - Registro, 27
    - IR, 27
    - MAR, 27
    - MDR, 27
    - PC, 27
    - PSW, 27
  - CU, 26
- da Vinci, Leonardo, 6
- Dato, 4, 53
  - di input, 39
  - di output, 39
  - Tipo, 53
    - Astratto, 56
    - Derivato, 54
    - Dominio, 53
    - Operazione, 53
    - Primitivo, 54
    - Ricorsivo, 76
- DEC, 14, 15
  - Diagramma a blocchi, 40
  - Digital Equipment Corporation, *vedi* DEC
- DVD, 26
- EBNF, 47
- Elaborazione, 4
  - Unità centrale, *vedi* CPU

- Errore, 66
  - Correzione, 65
  - Grammaticale, 67
  - Identificazione, 65
  - Logico, 68
  - Semantica
    - Dinamica, 68
    - Statica, 67
  - Semantico, 67
- Espressione, 63
- Forma di Backus-Naur, *vedi* BNF
- FORTRAN, 14
- Gödel, Kurt, 11
- Grammatica, 46
- Hardware, 20
- Heap, 73
  - Garbage collector, 74
- Hilbert, David, 10
- Hollerith, Herman, 10
- Informatica, 2
- Informazione, 3
- Interfaccia
  - di I/O, 21, 28
- Interpretazione, 50
- Jacquard, Joseph Marie, 8
- Jobs, Steve, 16
- Leibniz
  - calcolatrice, 7
  - Gottfried Wilhelm, 7
- Linguaggio, 4, 40, 45, 46
  - di comandi, 4
  - di programmazione, 4, 40, 46
  - Grammatica, 45
  - naturale, 45
  - non verbale, 45
  - Regole di produzione, 46
  - verbale, 45
- Lovelace, Ada, 9
- Malfunzionamento, 66
- Memoria, 21
  - centrale, 21, 22
- Locazione, 22
- Parola, 22
  - secondaria, 22, 23
  - spazio di indirizzamento, 22
- Microprocessore, 26
- Modello, 37
- Neumann
  - John, 12
  - Macchina di, 12, 21
- Numerazione, Sistema di, 29
  - Base, 29
  - Binario, 8, 30
  - Decimale, 29
  - Esadecimale, 30
  - Ottale, 30
- Ordinamento, 79
  - a bolle, 80
  - per fusione, 80
  - per inserzione, 79
  - per selezione, 79
  - veloce, 80
- Parametro, 48
  - Attuale, 48
  - Formale, 48
- PASCAL, 15
- Pascal
  - Blaise, 6
  - Pascalina, 6
- Problema, 37
  - Descrizione, 39
- processo, 40
- Processore, *vedi* CPU
- Programma, 20, 40
  - Leggibilità, 51
  - Post-condizione, 66
  - Pre-condizione, 66
  - Specificità, 66
- Programmazione, 49
- Paradigma, 49
  - Funzionale, 49
  - Imperativo, 49
  - Logica, 49
  - Orientato agli oggetti, 50
  - strutturata, 42
- Pseudocodifica, 41

- Puntatore, 57
- RAM, 13, 23
- Random Access Memory, *vedi* RAM
- Read Only Memory, *vedi* ROM
- Record, 60
- Regola di produzione, 46
- Ricorsione, 75
  - Funzione induttiva, 76
  - Insieme induttivo, 75
  - Sottoprogramma ricorsivo, 76
- Riferimento, 58
- ROM, 23
- Scheda perforata, 8
- Schickard, Wilhelm, 6
- Simbolo terminale, 46
- Sintassi, *vedi* Grammatica
- Sistema, 20
  - di elaborazione, 20
  - Operativo, 20, 33
    - Gestore dei file, 35
    - Gestore della memoria, 34
    - Gestore delle periferiche, 35
    - Interprete dei comandi, 35
    - Nucleo, 33
    - Programmi di utilità, 35
    - operativo, 15
  - Software, 20, 33
  - Sottoprogramma, 48
    - Firma, 48
    - Funzione, 49
    - Procedura, 49
    - Prototipo, 48
  - Stack, 74
    - Record di attivazione, 74
  - Struttura collegata, 60
    - Coda, 62
    - Lista, 60
    - Pila, 62
  - Tipo
    - Generico, 60
  - Turing, Alan, 11
- Unicode, 31
- Unità
  - di controllo, *vedi* CU
- logico-aritmetica, *vedi* ALU
- Universo linguistico, 46
- Variabile, 37
- Identificatore, 38
- Zuse, Konrad, 10