1. Linguaggi di Programmazione – Introduzione

1.1 La programmazione

Il motivo fondamentale che da sempre spinge l'uomo alla progettazione e alla creazione di macchine di qualsiasi tipo è quello di facilitare e alleggerire il proprio lavoro. Il calcolatore elettronico non si sottrae a questa legge: esso è stato creato in origine perché si sostituisse all'uomo nella soluzione di tediosi problemi numerici, come la compilazione delle tabelle dei logaritmi. I primi calcolatori elettronici erano progettati per risolvere problemi particolari, ma con l'introduzione di calcolatori ad uso generico *(general-purpose)*, capaci cioè di risolvere un'ampia classe di problemi se opportunamente guidati, nasce la programmazione.

La programmazione può essere definita come la disciplina che demanda al calcolatore elettronico la soluzione di un determinato problema. Un linguaggio di programmazione è, di conseguenza, lo strumento utilizzato per scrivere programmi che realizzano algoritmi. Da un certo punto di vista, un linguaggio di programmazione può essere immaginato come il linguaggio che consente la comunicazione tra l'uomo (in particolare, il programmatore) e il calcolatore. Il programmatore, infatti, ha davanti a sé un problema da risolvere e vuole che il calcolatore lo faccia per lui. Naturalmente, il calcolatore non è in grado di farlo da solo, ma ha bisogno che il programmatore lo istruisca su come risolvere il problema in questione. Il programma, quindi, non è altro che la traduzione di un algoritmo di soluzione per il problema nel linguaggio comprensibile al calcolatore. Il programmatore deve inoltre comunicare al calcolatore i dati del problema da risolvere, mentre quest'ultimo si preoccuperà di calcolare la soluzione e restituirla come risultato finale.

1.2 Il linguaggio macchina

Ogni calcolatore è in grado di comprendere un particolare linguaggio di programmazione di basso livello detto *linguaggio macchina*, il cui testo è una sequenza di bit che il processore interpreta, secondo un funzionamento dettato dalla sua struttura fisica, eseguendo una sequenza di azioni. Per comprendere meglio la struttura e le caratteristiche del linguaggio macchina, richiamiamo nella figura sottostante l'architettura della macchina di von Neumann che costituisce lo schema fondamentale su cui si basano i calcolatori elettronici.

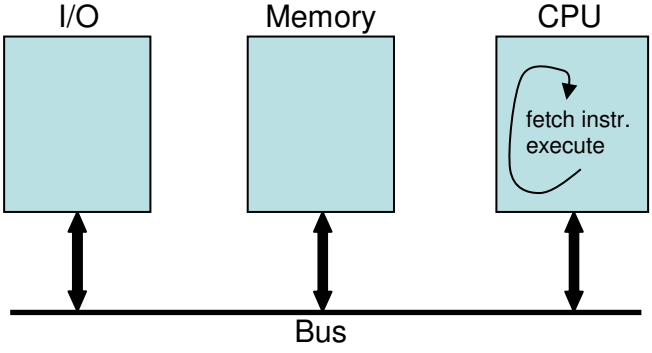


Figura 1. Macchina di von Neumann

Il linguaggio macchina deve essere facilmente rappresentabile sotto forma di segnali elettrici ed è quindi formato soltanto dalle cifre binarie 0 ed 1. Ogni modello di CPU ha il suo particolare linguaggio macchina anche se vi sono molte similitudini fra un idioma e un altro. Le "parole" di un linguaggio macchina sono chiamate *istruzioni*; ognuna di esse ordina alla CPU di eseguire un'azione elementare, come la lettura di una locazione di memoria oppure il calcolo della somma dei valori contenuti in due registri. Un programma è semplicemente una lunga lista di istruzioni che vengono eseguite da una CPU. L'esecuzione di un programma avviene processando un'istruzione per volta secondo uno schema che viene chiamato *ciclo di fetch-execute*. Esso consiste nel decodificare l'istruzione da eseguire (fetch) e quindi eseguirla (execute).

Abbiamo dunque da una parte l'uomo, capace di comunicare attraverso un linguaggio molto evoluto, e dall'altra il calcolatore, capace di comprendere soltanto il proprio linguaggio macchina. E' ovvio che i primi programmatori si sono trovati costretti ad imparare il linguaggio macchina dell'elaboratore da loro utilizzato. Programmare in linguaggio macchina risulta molto difficile, noioso e frustrante perché bisogna scrivere e leggere attraverso simboli binari e bisogna guidare il calcolatore attraverso tutti i suoi passi elementari verso l'ottenimento del risultato finale della computazione. Sulla spinta del loro disagio i primi programmatori operarono una semplificazione del linguaggio macchina introducendo il linguaggio *ASSEMBLER*. In questo linguaggio il codice binario delle varie istruzioni è sostituito da un codice mnemonico, vengono introdotte le variabili per rappresentare aree di memoria e i primi tipi di dato fondamentali come gli interi e i reali.

La differenza tra un programma scritto in linguaggio macchina e l'equivalente scritto in ASSEMBLER può essere apprezzata nell'esempio riportato di seguito.

ESEMPIO: calcolare z = x + y, dove x = 8 e y = 38.

Nel linguaggio macchina bisogna stabilire quali locazioni di memoria associare alle variabili x, y e z e codificare in binario i valori numerici 8 e 38, mentre utilizzando l'ASSEMBLER questo non si rende più necessario.

Linguaggio Macchina:

```
00000000000000000000000001000000
00000000000100000000000001000100
00000010000000010000000000000000
00000001000000000000000000111100
```

Assembler:

```
z : INT;
x : INT 8;
y : INT 38;
LOAD R0,x;
LOAD R1,y;
ADD R0,R1;
STORE R0,z;
```

1.3 La traduzione

L'unico e solo linguaggio che il calcolatore è in grado di comprendere ed eseguire è il linguaggio macchina: la CPU non è in grado di eseguire programmi scritti in linguaggio ASSEMBLER. Com'è che dunque si è affermata la programmazione in ASSEMBLER? Ciò è stato possibile grazie alla realizzazione di un apposito programma (scritto dunque in linguaggio macchina), detto Assemblatore, che si preoccupa di tradurre un qualsiasi programma scritto in ASSEMBLER nel programma equivalente codificato in linguaggio macchina.

Indubbiamente, la parte più delicata di questo processo di traduzione risulta essere la determinazione dell'effettiva area di memoria associata a ogni variabile utilizzata all'interno del programma. L'Assemblatore non è in grado di determinare in maniera assoluta gli indirizzi di memoria poiché non conosce quella che sarà la situazione della memoria all'atto dell'esecuzione del programma. Il codice generato dall'Assemblatore risulta essere quindi un codice intermedio, detto *codice rilocabile*, in cui i riferimenti agli indirizzi di memoria sono specificati in maniera relativa e non assoluta. Sarà compito di un altro programma, il *loader*, quello di "risolvere gli indirizzi" una volta che il programma viene caricato in memoria per essere eseguito.

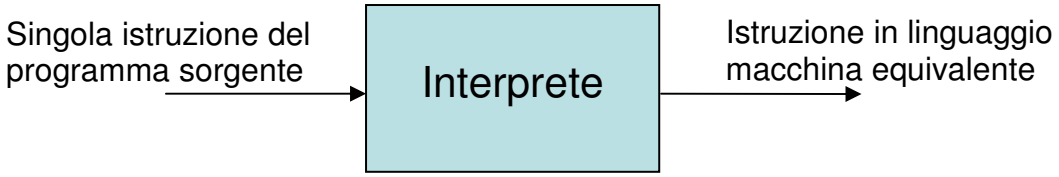
1.4 Linguaggi di alto livello

Una volta innescato il meccanismo di traduzione, i linguaggi di programmazione si sono evoluti fino ad arrivare ai cosiddetti *linguaggi di alto livello* la cui sintassi è molto più vicina al linguaggio naturale. Ciò è stato possibile grazie allo sviluppo di traduttori sempre più potenti e complessi capaci di ricondurre in maniera non ambigua qualsiasi programma all'equivalente codificato in linguaggio macchina.

Programmare in un dato linguaggio di programmazione significa generalmente scrivere uno o più semplici file di testo, chiamati *codice sorgente*. Il codice sorgente, contenente le istruzioni da eseguire e (spesso) alcuni dati noti e costanti, può essere poi eseguito passandolo ad un traduttore. Esistono due tipi diversi di traduttori: gli *interpreti* e i *compilatori*.

1.4.1 L'interpretazione

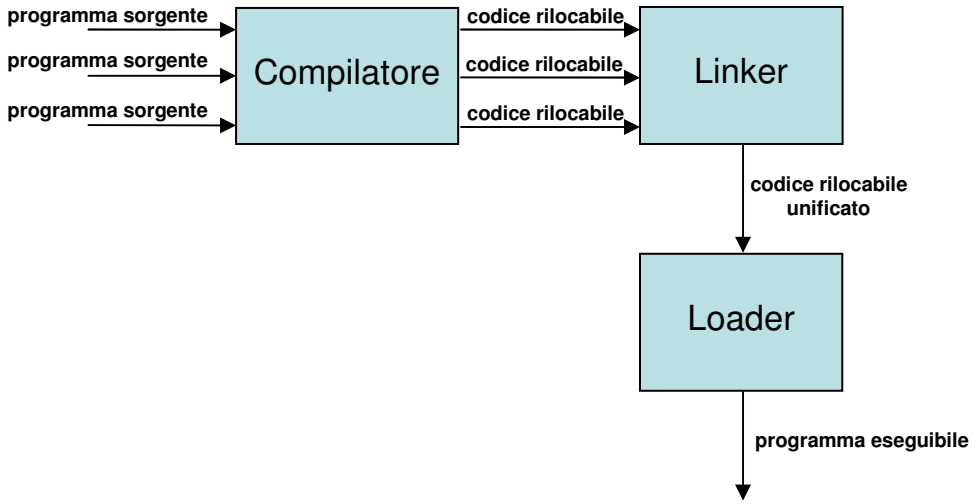
L'interpretazione può essere vista come la riproposizione ad alto livello del ciclo di fetch-execute della macchina di von Neumann. Essa consiste nel tradurre una singola istruzione del programma sorgente nell'istruzione in linguaggio macchina equivalente, la quale viene subito eseguita, per poi passare al processamento dell'istruzione successiva.



1.4.2 La compilazione

La compilazione consiste nel tradurre l'intero programma sorgente prima di eseguirlo. Attraverso la compilazione diviene possibile anche tradurre diversi programmi sorgenti in modo da ottenere un

unico *codice oggetto*. Ciò si ottiene compilando separatamente i vari programmi e poi collegando tra loro le traduzioni ottenute per mezzo di un apposito programma detto *linker*.



1.4.3 Interpretazione vs compilazione

L'approccio interpretato comporta una minore efficienza a *tempo di esecuzione (run-time)*; un programma interpretato, in esecuzione, richiede più memoria ed è meno veloce, a causa dell'*overhead* introdotto dall'interprete stesso. Durante l'esecuzione, l'interprete deve infatti analizzare le istruzioni a partire dal livello sintattico, identificare le azioni da eseguire (eventualmente trasformando i nomi simbolici delle variabili coinvolte nei corrispondenti indirizzo di memoria), ed eseguirle; mentre le istruzioni del codice compilato, già in linguaggio macchina, vengono caricate e istantaneamente eseguite dal processore. In compenso, l'interprete risparmia memoria a tempo di esecuzione dato che il programma compilato, ovvero il *codice oggetto*, può essere molto più grande del codice sorgente. D'altra parte, però, il compilatore, conoscendo l'intero codice sorgente, è in grado di effettuare ottimizzazioni sul codice oggetto ottenuto in modo da migliorarne le prestazioni. Infine, l'interpretazione risulta più utile nella determinazione di errori a tempo di esecuzione dato che spesso con la compilazione si perdono tutte le relazioni esistenti tra codice sorgente e codice oggetto. Di solito un linguaggio di programmazione è di tipo interpretato o compilato, ma esistono le eccezioni, come il linguaggio Java, che applica un ibrido fra le due soluzioni, utilizzando un compilatore per produrre del codice intermedio che viene successivamente interpretato.

1.5 Paradigmi di programmazione

Esistono numerose famiglie di linguaggi di programmazione, riconducibili a diversi *paradigmi di programmazione*. Ciascun paradigma fornisce al programmatore strumenti concettuali di diversa natura per descrivere gli algoritmi da far eseguire al calcolatore. Si parla così di un paradigma logico per quei linguaggi che consentono all'utente di esprimersi usando notazioni e concetti derivati dalla logica matematica e dal calcolo dei predicati; di paradigma funzionale se il linguaggio ha una struttura che ricorda da vicino il calcolo funzionale della matematica, e così via. Il paradigma più diffuso è quello procedurale o imperativo, ma al giorno d'oggi, il paradigma dominante è indubbiamente quello orientato agli oggetti, che deriva storicamente dal paradigma procedurale.

Linguaggi imperativi. La programmazione imperativa è un paradigma di programmazione secondo cui un programma viene inteso come un insieme di istruzioni (dette anche direttive, comandi), ciascuna delle quali può essere pensata come un "ordine" che viene impartito alla *macchina virtuale* definita dal linguaggio di programmazione utilizzato. Da un punto di vista sintattico, i costrutti di un linguaggio imperativo sono spesso identificati da verbi all'imperativo, per esempio: print (stampa), read (leggi), do (fà).

Programmazione funzionale. A livello puramente astratto un programma (o, equivalentemente, un algoritmo) è una funzione che, dato un certo input, restituisce un certo output (dove per output si può anche intendere la situazione in cui il programma non termini la propria esecuzione o restituisca un errore). La programmazione funzionale consiste esattamente nell'applicazione di questo punto di vista. Ogni operazione sui dati in input viene effettuata attraverso particolari funzioni elementari, appropriatamente definite del programmatore, che opportunamente combinate, attraverso il concetto matematico di composizione di funzioni, danno vita al programma. Il programma seguente, ad esempio, è un esempio di programma che calcola la somma di due numeri:

successore(x) = x + 1;

predecessore(x) = x – 1;

somma(x,0) = x;

somma(x,y) = somma(successore(x),predecessore(y));[1]

Programmazione logica. Il Prolog è stato il primo rappresentante di questa classe. Nati a partire da un progetto per un dimostratore automatico di teoremi, i linguaggi logici, o dichiarativi, rappresentano un modo completamente nuovo di concepire l'elaborazione dei dati: invece di una lista di comandi, un programma in un linguaggio dichiarativo è una lista di regole che descrivono le proprietà dei dati e i modi in cui questi possono trasformarsi. Le variabili non vengono mai assegnate (l'assegnamento non esiste), ma istanziate al momento dell'applicazione in una determinata regola; l'esecuzione del programma consiste nell'applicazione ripetuta delle regole disponibili fino a trovare una catena di regole e trasformazioni che consente di stabilire se il risultato desiderato è vero o meno. Non esistono né cicli, né salti, né un ordine rigoroso di esecuzione; affinché sia possibile usarli in un programma dichiarativo, tutti i normali algoritmi devono essere riformulati in termini ricorsivi e di backtracking; questo rende la programmazione con questi linguaggi un'esperienza del tutto nuova e richiede di assumere un modo di pensare radicalmente diverso, perché più che calcolare un risultato si richiede di dimostrarne il valore esatto. A fronte di queste richieste, i linguaggi dichiarativi consentono di raggiungere risultati eccezionali quando si tratta di manipolare gruppi di enti in relazione fra loro. A livello puramente esemplificativo, supponiamo di voler scrivere un programma che calcoli la radice quadrata di 64. In un ipotetico linguaggio logico esso potrebbe essere il seguente:

radice_quadrata(x) = y AND y * y = 64;

x = 64;

y = ?

[1] Naturalmente, è possibile definire semplicemente somma(x,y) = x + y; ma l'esempio riportato vuole semplicemente essere un'illustrazione del diverso tipo di approccio alla programmazione fornito dal paradigma funzionale.

Naturalmente, l'unico valore per y che rende veri i predicati costituenti il programma è 8.

Programmazione concorrente. I moderni supercomputer e ormai tutti i calcolatori di fascia alta e media sono equipaggiati con più di una CPU. Come ovvia conseguenza, questo richiede la capacità di sfruttarle; per questo sono stati sviluppati dapprima il multithreading, cioè la capacità di lanciare più parti dello stesso programma contemporaneamente su CPU diverse, e in seguito alcuni linguaggi studiati in modo tale da poter individuare da soli, in fase di compilazione, le parti di codice da lanciare in parallelo.

Linguaggi di scripting. I linguaggi di questo tipo nacquero come linguaggi batch: vale a dire liste di comandi di programmi interattivi che invece di venire digitati uno ad uno su una linea di comando, potevano essere salvati in un file, che diventava così una specie di comando composto che si poteva eseguire in modalità batch per automatizzare compiti lunghi e ripetitivi. I primi linguaggi di scripting sono stati quelli delle shell Unix; successivamente, vista l'utilità del concetto molti altri programmi interattivi iniziarono a permettere il salvataggio e l'esecuzione di file contenenti liste di comandi, oppure il salvataggio di registrazioni di comandi visuali (le cosiddette Macro dei programmi di videoscrittura, per esempio). Il passo successivo fu quello di far accettare a questi programmi anche dei comandi di salto condizionato e delle istruzioni di ciclo, regolati da simboli associati ad un certo valore: in pratica implementare cioè l'uso di variabili. Ormai molti programmi nati per tutt'altro scopo offrono agli utenti la possibilità di programmarli in modo autonomo tramite linguaggi di scripting più o meno proprietari. Molti di questi linguaggi hanno finito per adottare una sintassi molto simile a quella del C: altri invece, come il Perl e il Python, sono stati sviluppati ex novo allo scopo. Visto che nascono tutti come feature di altri programmi, tutti i linguaggi di scripting hanno in comune il fatto di essere linguaggi interpretati, cioè eseguiti da un altro programma (il programma madre o un suo modulo).

Programmazione orientata agli oggetti. La programmazione orientata agli oggetti (OOP, Object Oriented Programming) è un paradigma di programmazione, che prevede di raggruppare in un'unica entità (la classe) sia le strutture dati che le procedure che operano su di esse, creando per l'appunto un "oggetto" software dotato di proprietà (dati) e metodi (procedure) che operano sui dati dell'oggetto stesso. La modularizzazione di un programma viene realizzata progettando e realizzando il codice sotto forma di classi che interagiscono tra di loro. Un programma ideale, realizzato applicando i criteri dell'OOP, sarebbe completamente costituito da oggetti software (istanze di classi) che interagiscono gli uni con gli altri. La programmazione orientata agli oggetti è particolarmente adatta a realizzare interfacce grafiche.

1.6 Proprietà dei linguaggi

Non ha senso, in generale, parlare di linguaggi migliori o peggiori, o di linguaggi migliori in assoluto: ogni linguaggio nasce per affrontare una classe di problemi più o meno ampia, in un certo modo e in un certo ambito. Però, dovendo dire se un dato linguaggio sia adatto o no per un certo uso, è necessario valutare le caratteristiche dei vari linguaggi.

Caratteristiche intrinseche

Sono le qualità del linguaggio in sé, determinate dalla sua sintassi e dalla sua architettura interna. Influenzano direttamente il lavoro del programmatore, condizionandolo. Non dipendono né dagli strumenti usati (compilatore/interprete, linker) né dal sistema operativo o dal tipo di macchina.

- **Espressività**: la facilità e la semplicità con cui si può scrivere un dato algoritmo in un dato linguaggio; può dipendere dal tipo di algoritmo, se il linguaggio in questione è nato per affrontare certe particolari classi di problemi. In generale se un certo linguaggio consente di scrivere algoritmi con poche istruzioni, in modo chiaro e leggibile, la sua espressività è buona.

- **Didattica**: la semplicità del linguaggio e la rapidità con cui lo si può imparare. Il BASIC, per esempio, è un linguaggio facile da imparare: poche regole, una sintassi molto chiara e limiti ben definiti fra quello che è permesso e quello che non lo è. Il Pascal non solo ha i pregi del BASIC ma educa anche il neo-programmatore ad adottare uno stile corretto che evita molti errori e porta a scrivere codice migliore. Al contrario, il C non è un linguaggio didattico perché pur avendo poche regole ha una semantica molto complessa, a volte oscura, che lo rende molto efficiente ed espressivo ma richiede tempo per essere padroneggiata.

- **Leggibilità**: la facilità con cui, leggendo un codice sorgente, si può capire cosa fa e come funziona. La leggibilità dipende non solo dal linguaggio ma anche dallo stile di programmazione di chi ha creato il programma: tuttavia la sintassi di un linguaggio può facilitare o meno il compito. Non è detto che un linguaggio leggibile per un profano lo sia anche per un esperto: in generale le abbreviazioni e la concisione consentono a chi già conosce un linguaggio di concentrarsi meglio sulla logica del codice senza perdere tempo a leggere, mentre per un profano è più leggibile un linguaggio molto prolisso.

A volte, un programma molto complesso e poco leggibile in un dato linguaggio può diventare assolutamente semplice e lineare se riscritto in un linguaggio di classe differente, più adatta.

- **Robustezza**: è la capacità del linguaggio di prevenire, nei limiti del possibile, gli errori di programmazione. Di solito un linguaggio robusto si ottiene adottando un controllo molto stretto sui tipi di dati e una sintassi chiara e molto rigida; altri sistemi sono l'implementare un garbage collector, limitando (a prezzo di una certa perdita di efficienza) la creazione autonoma di nuove entità di dati e quindi l'uso dei puntatori, che possono introdurre bug molto difficili da scoprire.

L'esempio più comune di linguaggio robusto è il Pascal, che essendo nato a scopo didattico presuppone sempre che una irregolarità nel codice sia frutto di un errore del programmatore; mentre l'ASSEMBLER è l'esempio per antonomasia di linguaggio totalmente libero, in cui niente vincola il programmatore (e se scrive codice pericoloso o errato, non c'è modo di essere avvertiti).

- **Modularità**: quando un linguaggio facilita la scrittura di parti di programma indipendenti (moduli) viene definito modulare. I moduli semplificano la ricerca e la correzione degli errori, permettendo di isolare rapidamente la parte di programma che mostra il comportamento errato e modificarla senza timore di introdurre conseguenze in altre parti del programma stesso. Questo si ripercuote positivamente sulla manutenibilità del codice; inoltre permette di riutilizzare il codice scritto in passato per nuovi programmi, apportando poche modifiche. In genere la modularità si ottiene con l'uso di sottoprogrammi (subroutine, procedure, funzioni) e con la programmazione ad oggetti.

- **Flessibilità**: la possibilità di adattare il linguaggio, estendendolo con la definizione di nuovi comandi e nuovi operatori. I linguaggi classici come il BASIC, il Pascal e il Fortran non hanno questa capacità, che invece è presente nei linguaggi dichiarativi, in quelli funzionali e nei linguaggi imperativi ad oggetti più recenti come il C++ e Java.

- **Generalità**: la facilità con cui il linguaggio si presta a codificare algoritmi e soluzioni di problemi in campi diversi. Di solito un linguaggio molto generale, per esempio il C, risulta meno espressivo e meno potente in una certa classe di problemi di quanto non sia un linguaggio specializzato in quella particolare nicchia, che in genere è perciò una scelta migliore finché il problema da risolvere non esce da quei confini.

- **Efficienza**: la velocità di esecuzione e l'uso oculato delle risorse del sistema su cui il programma finito gira. In genere i programmi scritti in linguaggi molto astratti tendono ad essere lenti e voraci di risorse, perché lavorano entro un modello che non riflette la reale struttura dell'hardware ma è una cornice concettuale, che deve essere ricreata artificialmente; in compenso facilitano molto la vita del programmatore poiché lo sollevano dalla gestione di numerosi dettagli, accelerando lo sviluppo di nuovi programmi ed eliminando intere classi di errori di programmazione possibili. Viceversa un linguaggio meno astratto ma più vicino alla reale struttura di un computer genererà programmi molto piccoli e veloci ma a costo di uno sviluppo più lungo e difficoltoso.

- **Coerenza**: l'applicazione dei principi base di un linguaggio in modo uniforme in tutte le sue parti. Un linguaggio coerente è un linguaggio facile da prevedere e da imparare, perché una volta appresi i principi base questi sono validi sempre e senza (o con poche) eccezioni.

Caratteristiche esterne

Viste le qualità dei linguaggi, vediamo quelle degli ambienti in cui operano. Un programmatore lavora con strumenti software, la cui qualità e produttività dipende da un insieme di fattori che vanno pesati anch'essi in funzione del tipo di programmi che si intende scrivere.

- **Diffusione**: il numero di programmatori nel mondo che usa il tale linguaggio. Ovviamente più è numerosa la comunità dei programmatori tanto più è facile trovare materiale, aiuto, librerie di funzioni, documentazione, consigli. Inoltre ci sono un maggior numero di software house che producono strumenti di sviluppo per quel linguaggio, e di qualità migliore.

- **Standardizzazione**: un produttore di strumenti di sviluppo sente sempre la tentazione di introdurre delle variazioni sintattiche o delle migliorie più o meno grandi ad un linguaggio, originando un dialetto del linguaggio in questione e fidelizzando così i programmatori al suo prodotto: ma più dialetti esistono, più la comunità di programmatori si frammenta in sottocomunità più piccole e quindi meno utili. Per questo è importante l'esistenza di uno standard per un dato linguaggio che ne garantisca certe caratteristiche, in modo da evitarne la dispersione. Quando si parla di Fortran 77, Fortran 90, C 99 ecc. si intende lo standard sintattico e semantico del tale linguaggio approvato nel tale anno, in genere dall'ANSI o dall'ISO.

- **Integrabilità**: dovendo scrivere programmi di una certa dimensione, è molto facile trovarsi a dover integrare parti di codice precedente scritte in altri linguaggi: se un dato linguaggio di programmazione consente di farlo facilmente, magari attraverso delle procedure standard, questo è decisamente un punto a suo favore. In genere tutti i linguaggi "storici" sono bene integrabili, con l'eccezione di alcuni, come lo Smalltalk, creati più per studio teorico che per il lavoro reale di programmazione.

- **Portabilità**: la possibilità che portando il codice scritto su una certa piattaforma (CPU + architettura + sistema operativo) su un'altra, questo funzioni subito, senza doverlo

modificare. A questo scopo è molto importante l'esistenza di uno standard del linguaggio, anche se a volte si può contare su degli standard de facto come il Delphi.

1.7 Cenni storici

Il primo linguaggio di programmazione della storia è a rigor di termini il Plankalkül di Konrad Zuse, sviluppato da lui nella svizzera neutrale durante la II guerra mondiale e pubblicato nel 1946; ma non venne mai realmente usato per programmare.

La programmazione dei primi elaboratori veniva fatta invece in Shortcode, da cui poi si è evoluto l'assembly o ASSEMBLER, che costituisce una rappresentazione simbolica del linguaggio macchina. La sola forma di controllo di flusso è l'istruzione di salto condizionato, che porta a scrivere programmi molto difficili da seguire logicamente per via dei continui salti da un punto all'altro del codice.

La maggior parte dei linguaggi di programmazione successivi cercarono di astrarsi da tale livello basilare, dando la possibilità di rappresentare strutture dati e strutture di controllo più generali e più vicine alla maniera (umana) di rappresentare i termini dei problemi per i quali ci si prefigge di scrivere programmi. Tra i primi linguaggi ad alto livello a raggiungere una certa popolarità ci fu il Fortran, creato nel 1957 da John Backus, da cui derivò successivamente il BASIC (1964): oltre al salto condizionato, reso con l'istruzione IF, questa nuova generazione di linguaggi introduce nuove strutture di controllo di flusso come i cicli WHILE e FOR e le istruzioni CASE e SWITCH: in questo modo diminuisce molto il ricorso alle istruzioni di salto (GOTO), cosa che rende il codice più chiaro ed elegante, e quindi di più facile manutenzione.

Dopo la comparsa del Fortran nacquero una serie di altri linguaggi di programmazione storici, che implementarono una serie di idee e paradigmi innovativi: i più importanti sono l'ALGOL (1960) e il Lisp (1959). Tutti i linguaggi di programmazione oggi esistenti possono essere considerati discendenti da uno o più di questi primi linguaggi, di cui mutuano molti concetti di base; l'ultimo grande progenitore dei linguaggi moderni fu il Simula (1967), che introdusse per primo il concetto (allora appena abbozzato) di oggetto software.

Nel 1970 Niklaus Wirth pubblica il Pascal, il primo linguaggio strutturato, a scopo didattico; nel 1972 dal BCPL nascono prima il B (rapidamente dimenticato) e poi il C, che invece fu fin dall'inizio un grande successo. Nello stesso anno compare anche il Prolog, finora il principale esempio di linguaggio logico, che pur non essendo di norma utilizzato per lo sviluppo industriale del software (a causa della sua inefficienza) rappresenta una possibilità teorica estremamente affascinante.

Con i primi mini e microcomputer e le ricerche a Palo Alto, nel 1983 vede la luce Smalltalk, il primo linguaggio realmente e completamente ad oggetti, che si ispira al Simula e al Lisp: oltre a essere in uso tutt'oggi in determinati settori, Smalltalk viene ricordato per l'influenza enorme che ha esercitato sulla storia dei linguaggi di programmazione, introducendo il paradigma orientato agli oggetti nella sua prima incarnazione matura. Esempi di linguaggi orientati agli oggetti odierni sono Eiffel (1986), C++ (che esce nello stesso anno di Eiffel) e successivamente Java, classe 1995.