Linguaggio di programmazione

```
4780 GOTO 5000
4790 :
4800 REM   --------------------
4801 REM   --- DARSTELLUNG ---
4802 REM   --- DES MANUALS ---
4803 REM   --------------------
4810 :
4820 PRINT"■■";
4825 W=V+1:IF W<0 THEN W=W+14
4830 FOR X=1 TO 2:PRINT"■■■■■■■■■■■■■■■";
4835 FOR I=0 TO 23
4840 PRINT MD$(I+W);
4850 NEXT:PRINT:NEXT
4860 PRINT"■■■■■■■■■■■■■■■";
4870 FOR I=0 TO 23
4880 IF MD$(I+W)=CHR$(32) THEN PRINT MB$
(I+1);:GOTO 4900
4890 PRINT MD$(I+W);
4900 NEXT
4910 PRINT:PRINT"■■■■■■■■■■■■■■■■";
4920 FOR I=2 TO 24 STEP 2
4925 PRINT"|";
4930 IF MD$(I+W-1)="■■ ■■" THEN PRINT"■
■";:GOTO 4940
4935 PRINT" ";
4940 NEXT:PRINT"■"
4950 PRINT"■■■■■■■■■■■■■■■■";
4960 FOR I=2 TO 24 STEP 2
4965 PRINT"|";
4970 IF MD$(I+W-1)="■■ ■■" THEN PRINT"■"
MB$(I)"■";:GOTO 4980
4975 PRINT MB$(I);
4980 NEXT:PRINT"■"
```

Codice sorgente di un programma scritto in linguaggio BASIC.

Un **linguaggio di programmazione**, in informatica, è un linguaggio formale, dotato (al pari di un qualsiasi linguaggio naturale) di un lessico, di una sintassi e di una semantica ben definiti. È utilizzabile per il controllo del comportamento di una macchina formale o di una implementazione di essa (tipicamente, un computer) ovvero in fase di programmazione di questa attraverso la scrittura del codice sorgente di un programma ad opera di un programmatore. Condizione sufficiente per un linguaggio per essere considerato un linguaggio di programmazione è l'essere Turing completo.

Storia

Il primo linguaggio di programmazione della storia, se si esclude il linguaggio meccanico adoperato da Ada Lovelace per la programmazione della macchina di Charles Babbage, è a rigor di termini il Plankalkül di Konrad Zuse, sviluppato da lui nella Svizzera neutrale durante la seconda guerra mondiale e pubblicato nel 1946. Plankalkül non venne mai realmente usato per programmare.

La programmazione dei primi elaboratori veniva fatta invece in short code, da cui poi si è evoluto l'assembly, 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 il Lisp (1959) e l'ALGOL (1960) . 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 object-oriented nella sua prima incarnazione *matura*. Esempi di linguaggi object-oriented odierni sono Eiffel (1986), C++ (che esce nello stesso anno di Eiffel) e successivamente Java, classe 1995.

Concetti fondamentali

Tutti i linguaggi di programmazione esistenti sono definiti da un lessico, una sintassi ed una semantica e possiedono (almeno) questi due concetti chiave:

- Variabile: un dato o un insieme di dati, noti o ignoti, già memorizzati o da memorizzare; ad una variabile corrisponde sempre, da qualche parte, un certo numero (fisso o variabile) di locazioni di memoria che vengono *allocate*, cioè riservate, per contenere i dati stessi. Molti linguaggi inoltre attribuiscono alle variabili un tipo, con differenti proprietà (stringhe di testo, numeri, liste, *atomi* ecc.).
- Istruzione: un comando oppure una regola descrittiva: anche il concetto di istruzione è molto variabile fra i vari linguaggi. A prescindere dal particolare linguaggio però, ogni volta che un'istruzione viene eseguita, lo stato interno del calcolatore (che sia lo stato reale della macchina oppure un ambiente virtuale, teorico, creato dal linguaggio) cambia.

Alcuni concetti sono poi presenti nella gran parte dei linguaggi:

- Espressione: una combinazione di variabili e costanti, unite da operatori; le espressioni sono state introdotte inizialmente per rappresentare le espressioni matematiche, ma in seguito la

loro funzionalità si è estesa. Una espressione viene **valutata** per produrre un valore, e la sua valutazione può produrre "effetti collaterali" sul sistema e/o sugli oggetti che vi partecipano.
- Strutture dati, meccanismi che permettono di organizzare e gestire dati complessi.
- Strutture di controllo, che permettono di governare il flusso di esecuzione del programma, alterandolo in base al risultato o valutazione di una espressione (che può ridursi al contenuto di una variabile, o essere anche molto complessa) (cicli iterativi quali ad esempio *for*, *do*, *while* e strutture condizionali quali ad esempio *if*, *switch-case*).
- Sottoprogramma: un blocco di codice che può essere richiamato da qualsiasi altro punto del programma. In tale ambito quasi tutti linguaggi offrono funzionalità di riuso di codice accorpando cioè sequenze di istruzioni all'interno di funzioni richiamabili secondo necessità all'interno di programmi o all'interno di librerie richiamabili in ogni programma.
- Funzionalità di input dati da tastiera e visualizzazione dati in output (stampa a video) attraverso i cosiddetti canali standard (standard input, standard output).
- Possibilità di inserire dei commenti sul codice scritto, sintatticamente identificati e delimitati, che ne esplichino le funzionalità a beneficio della leggibilità o intelligibilità.

Il codice sorgente

Programmare in un dato linguaggio di programmazione significa generalmente scrivere uno o più semplici file di testo ASCII, chiamato **codice sorgente** che esprime l'algoritmo del programma tradotto nel linguaggio di programmazione. I font, i colori e in generale l'aspetto grafico sono irrilevanti ai fini della programmazione in sé: per questo i programmatori non usano programmi di videoscrittura, ma degli *editor* di testo (come *emacs* e *brief*) che invece offrono funzioni avanzate di trattamento testi (*espressioni regolari*, sostituzioni condizionali e ricerche su file multipli, possibilità di richiamare strumenti esterni ecc).

Se un dato editor è in grado di lavorare a stretto contatto con gli altri strumenti di lavoro (compilatore, linker, interprete ecc.: vedi più avanti) allora più che di semplice editor si parla di **IDE** o ambiente di sviluppo integrato. Va notato che alcuni linguaggi di programmazione recenti consentono anche una forma mista di programmazione, in cui alla stesura di codice sorgente ASCII si associano anche operazioni di programmazione visuale, attraverso le quali il programmatore descrive alcuni aspetti del programma *disegnando* a video attraverso il mouse; un'applicazione tipica di quest'ultima forma di programmazione è il disegno interattivo della GUI del programma (finestre, menù, e così via).
Per essere eseguito dal processore il codice sorgente deve essere tradotto in linguaggio macchina che è il linguaggio in cui opera la macchina a livello fisico, e questo è possibile attraverso due possibile tecniche: la compilazione e l'interpretazione.

Il codice sorgente, contenente le istruzioni da eseguire e (spesso) alcuni dati noti e costanti, può essere poi eseguito passandolo ad un interprete che eseguirà le istruzioni in esso contenute, il che è la prassi normale per i linguaggi di scripting; oppure può venire compilato, cioè tradotto in istruzioni di linguaggio macchina da un programma compilatore: il risultato è un file binario 'eseguibile' (codice eseguibile) che non ha bisogno di altri programmi per andare in esecuzione, ed è anche molto più veloce di un programma interpretato.

In passato, la compilazione è stata la norma per tutti i linguaggi di programmazione di uso generale; attualmente vi sono numerosi linguaggi interpretati e di uso generale, come il linguaggio Java o quelli della piattaforma .NET, che applicano un approccio ibrido fra le due soluzioni, utilizzando un compilatore per produrre del codice in un linguaggio *intermedio* (detto bytecode) che viene

successivamente interpretato. La differenza di prestazioni tra i linguaggi interpretati e quelli compilati è stata ridotta con tecniche di compilazione just-in-time, sebbene si continui ad utilizzare i linguaggi compilati (se non addirittura l'assembly) per le applicazioni che richiedono le massime prestazioni possibili.

La compilazione

La compilazione è il processo per cui il programma, scritto in un linguaggio di programmazione ad alto livello, viene tradotto in un codice eseguibile per mezzo di un altro programma detto appunto compilatore.

La compilazione offre numerosi vantaggi, primo fra tutti il fatto di ottenere eseguibili velocissimi nella fase di run (esecuzione) adattando vari parametri di questa fase all'hardware a disposizione; ma ha lo svantaggio principale nel fatto che è necessario compilare un eseguibile diverso per ogni sistema operativo o hardware (piattaforma) sul quale si desidera rendere disponibile l'esecuzione ovvero viene a mancare la cosiddetta portabilità.

Il collegamento (linking)

Se il programma, come spesso accade, usa delle librerie, o è composto da più moduli software, questi devono essere 'collegati' tra loro. Lo strumento che effettua questa operazione è detto appunto linker ("collegatore"), e si occupa principalmente di risolvere le interconnessioni tra i diversi moduli.

Esistono principalmente due tipi differenti di collegamento: **dinamico** e **statico**.

Collegamento statico

Tutti i moduli del programma e le librerie utilizzate vengono incluse nell'eseguibile, che risulta grande, ma contiene tutto quanto necessario per la sua esecuzione. Se si rende necessaria una modifica ad una delle librerie, per correggere un errore o un problema di sicurezza, tutti i programmi che le usano con collegamento statico devono essere ricollegati con le nuove versioni delle librerie.

Collegamento dinamico

Le librerie utilizzate sono caricate dal sistema operativo quando necessario (*linking dinamico*; le librerie esterne sono chiamate "DLL", *Dynamic-link libraries* nei sistemi Microsoft Windows, mentre "SO" *Shared Object* nei sistemi Unix-like). L'eseguibile risultante è più compatto, ma dipende dalla presenza delle librerie utilizzate nel sistema operativo per poter essere eseguito.

In questo modo, le librerie possono essere aggiornate una sola volta a livello di sistema operativo, senza necessità di ricollegare i programmi. Diventa anche possibile usare diverse versioni della stessa libreria, o usare librerie personalizzate con caratteristiche specifiche per il particolare host.

Nella realizzazione di un progetto software complesso, può succedere che alcune parti del programma vengano realizzate come librerie, per comodità di manutenzione o per poterle usare in diversi programmi che fanno parte dello stesso progetto.

La complicazione aggiunta è che quando si installa un programma con collegamento dinamico è necessario verificare la presenza delle librerie che utilizza, ed eventualmente installare anche queste. I sistemi di package management, che si occupano di installare i programmi su un sistema operativo, di solito tengono traccia automaticamente di queste dipendenze.

In genere si preferisce il collegamento dinamico, in modo da creare programmi piccoli e in generale ridurre la memoria RAM occupata, assumendo che le librerie necessarie siano già presenti nel sistema, o talvolta distribuendole insieme al programma.

L'interpretazione

```
#A is height B is radius

def cone (a, b):
    formula = (3.14 * .33 *a)*(b * b)
    return formula
```

Un codice python

Per cercare di eliminare il problema della portabilità (la dipendenza o meno del linguaggio dalla piattaforma) si è tentato di creare altri linguaggi che si potessero basare soltanto su librerie compilate (componenti) ad hoc per ogni piattaforma, mentre il loro codice viene interpretato, e quindi non c'è la necessità di una compilazione su ogni tipologia di macchina su cui viene eseguito.[*senza fonte*] Il grosso difetto di questi linguaggi è la lentezza dell'esecuzione; però hanno il grosso pregio di permettere di usare lo stesso programma senza modifica su più piattaforme. Si dice in questo caso che il programma è portabile.

La perdita di prestazioni che è alla base dei linguaggi interpretati è il doppio lavoro che è affidato alla macchina che si accinge ad elaborare tale programma. Al contrario di un programma compilato, infatti, ogni istruzione viene controllata e interpretata ad ogni esecuzione da un interprete.

Si usano linguaggi interpretati nella fase di messa a punto di un programma per evitare di effettuare numerose compilazioni o invece quando si vuole creare software che svolgono operazioni non critiche che non necessitano di ottimizzazioni riguardanti velocità o dimensioni, ma che traggono più vantaggio dalla portabilità. I linguaggi di scripting e tutti quelli orientati al Web sono quasi sempre interpretati. PHP, Perl, Tcl/Tk e JavaScript e molti altri sono esempi concreti di interazione non vincolata alla piattaforma.

Ci sono vari tentativi per rendere i compilatori multipiattaforma creando un livello intermedio, una sorta di semi-interpretazione, come nel caso sopra menzionato di Java; d'altro canto per i linguaggi interpretati ci sono tentativi per generare delle compilazioni (o semi-compilazioni) automatiche specifiche per la macchina su cui sono eseguiti.

Esistono anche strumenti per automatizzare per quanto possibile la compilazione di uno stesso programma su diverse piattaforme, ad esempio GNU autoconf/automake, che permette di realizzare una distribuzione del codice sorgente che può essere configurata e compilata automaticamente su diverse piattaforme, in genere almeno tutti gli Unix.

Confronto tra compilazione e interpretazione

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
   else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Un esempio di codice sorgente in Python. L'evidenziazione di alcune parti di codice è uno strumento comune fra i programmatori per orientarsi fra il codice.

Questi due metodi di creazione ed esecuzione di un programma presentano entrambi vantaggi e svantaggi: il maggior vantaggio della compilazione è senz'altro l'efficienza nettamente superiore in termini di prestazioni, al prezzo del restare vincolati ad una **piattaforma** (combinazione di architettura hardware e sistema operativo) particolare; un linguaggio interpretato invece non ha, in linea di massima, questa dipendenza ma è più lento e richiede più memoria in fase di esecuzione.

Bytecode e P-code

Una soluzione intermedia fra compilazione e interpretazione è stata introdotta nelle prime versioni di Pascal (compresa quella realizzata nel 1975 dal suo inventore, Niklaus Wirth) e successivamente adottata nei linguaggi Java e Python, con il bytecode, e nei linguaggi Visual Basic e .NET di Microsoft con il P-code.

In tutti e due questi casi il codice sorgente dei programmi non viene compilato in linguaggio macchina, ma in un codice intermedio "ibrido" destinato a venire interpretato al momento dell'esecuzione del programma: il motivo di questo doppio passaggio è di avere la portabilità dei linguaggi interpretati ma anche, grazie alla pre-compilazione, una fase di interpretazione più semplice e quindi più veloce. Nel caso del bytecode di Java siamo di fronte a un vero linguaggio assembly, che in origine doveva essere implementato in un modello di processore reale, poi mai realizzato; alcuni microprocessori moderni, come gli ARM con Jazelle implementano nativamente molte istruzioni bytecode e sono quindi in grado di eseguire bytecode Java come fosse assembly.

Tuttavia il codice intermedio è più facile sia da interpretare che da compilare: per questo motivo sia per Java che per i linguaggi .NET sono stati sviluppati i compilatori JIT (Just In Time), che al momento del lancio di un programma Java o .NET compilano al volo il codice intermedio e mandano in esecuzione un codice macchina nativo, eliminando completamente la necessità dell'interprete e rendendo i programmi scritti in questi linguaggi veloci quasi quanto i corrispondenti programmi compilati.

Classi di linguaggi

In generale esistono circa 2500 linguaggi di programmazione più o meno noti e diffusi. Questi in primis vengono classificati in linguaggi di programmazione ad alto livello e linguaggi di programmazione a basso livello a seconda del livello di astrazione a partire dal linguaggio macchina fin verso il linguaggio logico umano. A loro volta i linguaggi possono essere classificati in linguaggi *compilati* e *interpretati* come visto sopra. Normalmente i linguaggi vengono poi distinti in tre grandi famiglie basate sul paradigma di programmazione di riferimento: i linguaggi **imperativi**, quelli **funzionali** e quelli **logici**.

Imperativi

Nei linguaggi imperativi l'istruzione è un comando esplicito, che opera su una o più variabili oppure sullo stato interno della macchina, e le istruzioni vengono eseguite in un ordine prestabilito. Scrivere un programma in un linguaggio imperativo significa essenzialmente occuparsi di cosa la macchina deve fare per ottenere il risultato che si vuole, e il programmatore è impegnato nel mettere a punto gli algoritmi necessari a manipolare i dati.

Le strutture di controllo assumono la forma di *istruzioni di flusso* (GOTO, FOR, IF/THEN/ELSE ecc.) e il calcolo procede per iterazione piuttosto che per ricorsione. I valori delle variabili sono spesso assegnati a partire da costanti o da altre variabili (assegnamento) e raramente per passaggio di parametri (istanziazione).

Tipici linguaggi imperativi:

- APL
- Assembly
- ALGOL
- B
- BASIC
- BCPL
- COBOL
- FORTRAN
- Forth
- Hot soup processor
- PL/I
- POP

Strutturati

La programmazione strutturata è una tecnica il cui scopo è di limitare la complessità della struttura del controllo dei programmi. Il programmatore è vincolato ad usare solo le strutture di controllo canoniche definite dal Teorema di Böhm-Jacopini, ovvero la *sequenza*, la *selezione* e il *ciclo*, evitando le istruzioni di salto incondizionato.

- Ada
- C
- Fortran 90/95
- Modula-2
- Oberon
- Pascal

Orientati ad oggetti

La programmazione a oggetti è basata su una evoluzione del concetto di tipo di dato astratto caratterizzata da incapsulamento, ereditarietà, polimorfismo. Oltre a linguaggi specializzati che implementano completamente i principi di tale metodologia (come Smalltalk o Java), molti linguaggi moderni incorporano alcuni concetti della programmazione a oggetti.

- Ada95
- Attack
- BETA
- Clarion
- CLOS
- C++
- C#
- D
- DataFlex
- Delphi
- Eiffel
- Fortran 2003
- Java
- Linden Scripting Language
- Modula-3
- mShell
- Objective C
- OCaml
- OpenGenera
- PHP
- Python
- PowerBuilder
- REALbasic
- REBOL
- Ruby
- Scala
- Scriptol
- Simula

- Smalltalk
- Visual Basic
- Visual Basic .NET

Funzionali

I linguaggi funzionali sono basati sul concetto matematico di funzione. In un linguaggio funzionale puro l'assegnazione esplicita risulta addirittura completamente assente e si utilizza soltanto il passaggio dei parametri. Tipicamente in tale modello il controllo del calcolo è gestito dalla ricorsione e dal pattern matching, mentre la struttura dati più diffusa è la lista, una sequenza di elementi. Il più importante esponente di questa categoria è senz'altro il Lisp (LISt Processing).

- Clarion
- Clean
- Curry
- Haskell
- Lisp
- Logo
- PHP
- Scala
- Scheme
- Standard ML
- Caml
- OCaml

Dichiarativi (o logici)

Nei linguaggi logici l'istruzione è una *clausola* che descrive una relazione fra i dati: programmare in un linguaggio logico significa descrivere l'insieme delle relazioni esistenti fra i dati e il risultato voluto, e il programmatore è impegnato nello stabilire in che modo i dati devono evolvere durante il calcolo. Non c'è un ordine prestabilito di esecuzione delle varie clausole, ma è compito dell'interprete trovare l'ordine giusto.

La struttura di controllo principale è rappresentata dal **cut**, che è detto *rosso* se modifica il comportamento del programma o *verde* se rende solo più efficiente il calcolo, che procede per ricorsione e non per iterazione. Le variabili ricevono il loro valore per istanziazione o da altre variabili già assegnate nella clausola (*unificazione*) e quasi mai per assegnamento, che è usato solo in caso di calcolo diretto di espressioni numeriche.

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 una 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.

- Curry
- Mercury
- Prolog

Altre classificazioni

Dal punto di vista dei tipo di dato espresso un linguaggio può essere a tipizzazione forte o a tipizzazione debole.

Linguaggi esoterici

- Befunge
- Brainfuck
- COW
- FALSE
- HQ9+
- HQ9++
- INTERCAL
- Malbolge
- Whitespace
- LOLCODE

Linguaggi paralleli

I moderni supercomputer e - ormai - tutti i calcolatori di fascia alta e media sono equipaggiati con più 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.

- Occam
- Linda
- Axum

Linguaggi di scripting

I linguaggi di scripting sono nati come *linguaggi batch*, per automatizzare compiti lunghi e ripetitivi da eseguire, appunto, in modalità batch. Invece di digitare uno ad uno i comandi per realizzare un certo compito, essi sono salvati in sequenza in un file, utilizzabile a sua volta come comando composto. I primi linguaggi di scripting sono stati quelli delle shell Unix; successivamente, vista l'utilità del concetto, molti altri programmi interattivi hanno cominciato 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, è stato in molti casi l'estensione dei linguaggi con l'associazione di simboli a valori, cioè l'uso di variabili, con i comandi di gestione del flusso, ovvero i costrutti di salto condizionato, le istruzioni di ciclo o di ricorsione, rendendoli così linguaggi completi. Recentemente molti programmi nati per scopi ben diversi dalla programmazione offrono agli utenti la possibilità di programmarli in modo autonomo tramite linguaggi di scripting.

La sintassi di molti linguaggi di scripting, come PHP o i dialetti di ECMAScript, è simile a quella del C, mentre altri, come Perl o Python, ne adottano invece una progettata ex novo. Visto che molto spesso i linguaggi di scripting nascono per l'invocazione di comandi o procedure esterne, altrettanto spesso essi sono interpretati, cioè eseguiti da un altro programma, come il programma madre, del quale il linguaggio di scripting è una estensione, o un apposito interprete.

- AutoIt
- Applescript
- ActionScript
- Game Maker Language (vedi Game Maker)
- Hybris
- HyperTalk
- JavaScript
- JScript (Implementazione Microsoft di Javascript)
- mIRC scripting
- Lingo
- Lua
- Perl
- PHP
- Python
- QBasic
- Rexx
- Ruby
- Tcl
- thinBasic
- Visual Basic for Applications (VBA)
- VBScript

Valutare un linguaggio di programmazione

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, IDE, 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; la segnalazione e gestione di errori comuni a runtime dovuti a dati che assumono valori imprevisti (overflow, underflow) o eccedono i limiti definiti (indici illegali per vettori o matrici) controllo dei limiti; 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'assembly 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 niente che lo avverta).

- **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

Oltre alle accennate qualità dei linguaggi, possono essere esaminate 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 C *K&R* o il Delphi.