DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Dopo aver affrontato negli scorsi anni due argomenti fondanti dell'Informatica – il modo di codificare l'informazione digitale e la concreta possibilità di risolvere problemi mediante gli elaboratori elettronici – con questa terza serie andiamo ad esplorare "*Come parlano i calcolatori*". La teoria dei linguaggi e la creazione di linguaggi di programmazione hanno accompagnato di pari passo l'evolversi delle architetture di calcolo e di gestione dei dati, permettendo lo sviluppo di applicazioni sempre più complesse, svincolando il programmatore dall'architettura dei sistemi e consentendogli quindi di concentrarsi sull'essenza del problema da risolvere.
Lo sviluppo dell'Informatica distribuita ha comportato la nascita, accanto ai linguaggi per l'interazione tra programmatore e calcolatore, anche di linguaggi per far parlare i calcolatori tra di loro – i protocolli di comunicazione. Inoltre, la necessità di garantire la sicurezza e la privatezza delle comunicazioni, ha spinto allo sviluppo di tecniche per "non farsi capire" da terzi, di quì l'applicazione diffusa della crittografia.
Di questo e di altro parleranno le monografie quest'anno, come sempre affidate alla penna (dovrei dire tastiera!) di autori che uniscono una grande autorevolezza scientifica e professionale ad una notevole capacità divulgativa.

I linguaggi per la programmazione

Carlo Ghezzi

1. Introduzione

Il linguaggio costituisce il mezzo primario per la comunicazione tra le persone. In assenza di un linguaggio col quale comunicare in forma orale e scritta non esisterebbe una società organizzata: non sarebbe possibile l'interazione e la cooperazione tra gli individui né sarebbe possibile l'accumulo e la trasmissione di conoscenze. Lo studio del linguaggio è pertanto sempre stato strettamente connesso con gli studi sociali, antropologici e filosofici.
A partire dalla seconda metà del secolo scorso, la nascita dell'informatica ha fatto nascere una nuova dimensione di sviluppo dei linguaggi: i linguaggi per la comunicazione tra l'uomo e il computer. In realtà la comunicazione tra uomo e macchina è un fenomeno più generale, nato prima e indipendentemente dall'informatica; ma è solo attraverso le "macchine informatiche" che la comunicazione raggiunge livelli di sofisticazione che richiedono l'uso di linguaggi di complessità paragonabile a quella dei linguaggi per la comunicazione umana. In questo articolo ci concentreremo sui linguaggi per la programmazione, lasciando ad altri articoli un approfondimento di altre forme linguistiche tipiche dell'informatica (ad esempio, i linguaggi per il trattamento di dati, i linguaggi per la specifica dei requisiti ecc.).

2. Perché i linguaggi di programmazione?

Il computer è una macchina in grado di eseguire programmi scritti nel proprio linguaggio macchina: è un interprete del linguaggio macchina. I programmi consentono di specializzare il computer all'esecuzione di compiti specifici; una volta rappresentati nella forma binaria richiesta dalla tecnologia digitale del computer e inseriti nella memoria centrale, questi vengono interpretati dall'unità centrale di elaborazione (CPU).
Fin dalla nascita dei primi computer, ci si rese conto che programmare direttamente in linguaggio macchina non sarebbe stato agevole, se non per piccoli e semplici programmi. La rappresentazione del programma in linguaggio macchina avrebbe comportato la scomposizione del problema da risolvere in un algoritmo espresso in istruzioni da formulare in un lin-

guaggio binario: istruzioni elementari per il trasferimento di dati dalla e nella memoria, per l'esecuzione di semplici operazioni aritmetico-logiche, per il trasferimento esplicito del controllo mediante la modifica del "program counter". In sintesi, il livello di astrazione al quale il programmatore doveva esprimersi era troppo lontano dal problema che voleva risolvere.

Una prima soluzione fu quella di rendere simbolico il linguaggio macchina, definendo così i linguaggi di tipo assembler. In tal modo, anziché usare un ostico alfabeto binario, il programmatore poteva dare nomi simbolici mnemonici alle operazioni elementari del computer e alle celle di memoria contenenti dati o istruzioni. Di fatto, si manteneva una corrispondenza biunivoca tra istruzioni in linguaggio macchina e istruzioni assembler: semplicemente, si rendevano queste ultime più facilmente usabili, in quanto più facilmente comprensibili alle persone umane.

Ma la storia dei linguaggi di programmazione era solo agli inizi. A partire dagli anni 50 del secolo scorso iniziò infatti un flusso continuo di proposte e realizzazioni di nuovi linguaggi di programmazione, definiti allo scopo specifico di facilitare la programmazione dei computer. Si tratta di linguaggi che si possono complessivamente definire di *alto livello*. Con ciò si intende che il livello espressivo fornito dal linguaggio si allontana dal livello di dettagli minuti imposti dalle operazioni svolte dai circuiti digitali che costituiscono l'elaboratore per avvicinarsi alle capacità espressive umane. Senza questi linguaggi possiamo senz'altro affermare che l'informatica non si sarebbe diffusa fino a diventare la tecnologia fondante della vita contemporanea: non si sarebbero sviluppate le applicazioni che rendono i computer utili o addirittura indispensabili per il funzionamento quotidiano della nostra società.

La motivazione principale alla base della continua evoluzione dei linguaggi è proprio la necessità di rispondere meglio ai requisiti posti dalle applicazioni e dal loro processo di ingegnerizzazione. Le applicazioni hanno invaso settori applicativi che vanno dalla gestione aziendale al calcolo scientifico, dai giochi interattivi all'elaborazione di documenti, dal controllo di impianti in tempo reale alle gestione distribuita di servizi in rete. Le esigenze poste al linguaggio nei diversi casi sono spesso assai diverse. Allo stesso tempo sono diventati sempre più critici i requisiti di affidabilità delle applicazioni, di una loro facile adattabilità al mutare dei requisiti con il passare del tempo, di un loro sviluppo rapido da parte di team di programmatori che lavorano in parallelo. Anche i computer sono andati evolvendo nel tempo in termini di prestazioni e di memoria. Tutto ciò ha portato a una continua evoluzione dei linguaggi e fa presumere che questa evoluzione sia destinata a continuare nel prossimo futuro.

La tabella 1 sintetizza gli sviluppi storici dei linguaggi di programmazione, limitando la visione della Babele linguistica esistente a una piccola porzione che vuole mettere in luce i linguaggi che riteniamo più influenti. La tabella indica il periodo storico in cui i linguaggi sono stati definiti, evitando di mostrare le versioni successive che ne hanno accompagnato l'evoluzione. Nel seguito di questo articolo ci concentreremo invece non tanto sullo sviluppo storico dei linguaggio—compito impossibile per una trattazione sintetica come questa—quanto su alcuni concetti fondamentali che consentono una loro classificazione.

3. Classi di linguaggi e paradigmi di programmazione

I linguaggi di programmazione possono essere caratterizzati in base allo stile di programmazione che essi supportano, detto anche *paradigma di programmazione*. Un paradigma caratterizza il modo in cui il programma si presenta, come vengono espresse le computazioni e come il programma viene organizzato in parti.

1950-1960	1960-1970	1970-1980	1980-1990	1990-oggi
FORTRAN	BASIC	Smalltalk	Ada	Java
ALGOL 60	SIMULA 67	Modula-2	C++	C#
COBOL	ALGOL 68	PROLOG	Eiffel	PERL
APL	Pascal	Scheme	Common LISP	Python
LISP			CLOS	

TABELLA 1
Tappe fondamentali dell'evoluzione storica dei linguaggi

L'esistenza di diversi stili linguistici caratterizza anche altre forme espressive: dal linguaggio poetico a quello architettonico o musicale. In architettura si parla infatti di stile romanico o gotico, di "art nouveau" o di stile "bauhaus". Ciascuno stile è nato in risposta ad esigenze costruttive (e, ovviamente, estetiche). L'analogia vale anche per i linguaggi di programmazione, anche se la motivazione "estetica", pur presente, non è certo prevalente. Fondamentale è invece la capacità dei linguaggi di favorire lo sviluppo di software di buona qualità.

I paradigmi dei linguaggi di programmazione possono essere distinti in due classi fondamentali: paradigmi computazionali e strutturali. Una loro comprensione e la classificazione dei diversi linguaggi secondo i diversi paradigmi possono aiutare a meglio capire sia i linguaggi che le applicazioni che con essi vengono sviluppate.

Un *paradigma computazionale* caratterizza il modello di calcolo definito dal linguaggio. Il paradigma computazionale di gran lunga prevalente è quello imperativo, il cui modello di calcolo è il cosiddetto "modello di Von Neumann". Con ciò si intende che i programmi scritti utilizzando il linguaggio riflettono il modo di operare del computer soggiacente, il quale a sua volta riflette il modo di operare del computer primigenio, ideato da John Von Neumann. In altre parole, i linguaggi imperativi non sono altro che astrazioni della macchina di Von Neumann, il cui modello di calcolo consiste nell'esecuzione sequenziale, passo-passo, di istruzioni che modificano lo stato della memoria della macchina. Sono astrazioni, in quanto mascherano i dettagli di basso livello mentre mantengono i tratti fondamentali del funzionamento da cui astraggono. I linguaggi imperativi operano su variabili, intese come astrazioni di celle di memoria il cui contenuto viene modificato nel corso dell'esecuzione, e forniscono istruzioni per l'esecuzione condizionale (tipo *if...then...else*) e per le iterazioni (tipo *while...do...*). A questa classe appartengono linguaggi ormai obsoleti, ma tuttora in uso, come FORTRAN e COBOL, ma anche linguaggi più moderni come C, C++, Java e C#.

Val la pena osservare che con il passare del tempo, in realtà, la "macchina" in grado di eseguire i programmi è andata evolvendo rispetto al primitivo computer ideato da Von Neumann. In particolare, si è passati dal modello di calcolo della singola macchina di Von Neumann a quello di un insieme di macchine di Von Neumann operanti in parallelo. Ciò è stato reso possibile, da un lato, dalla definizione di meccanismi per poter "multi-programmare" i computer, fornendo l'astrazione del parallelismo logico dei programmi, e, dall'altro, dalla possibilità di interconnettere e coordinare più computer, realizzando sistemi fisicamente paralleli (sistemi multi-processore o sistemi distribuiti, interconnessi da una rete di comunicazione). Linguaggi moderni, come Java o C#, forniscono un supporto linguistico per la programmazione concorrente e distribuita, permettendo di definire sia più unità concorrenti che operano all'interno di un singolo nodo di elaborazione, sia unità che possono essere allocate su nodi diversi di una rete.

Anche se la concorrenza e la distribuzione delle computazioni ha reso più complesso il modello computazionale complessivo di linguaggi tipo Java o C#, i singoli nodi dell'architettura soggiacente il linguaggio continuano a seguire il paradigma di Von Neumann. Paradigmi computazionali diversi sono invece offerti dai linguaggi funzionali e dai linguaggi logici (o dichiarativi). In entrambi i casi si raggiunge un più elevato livello di astrazione rispetto al paradigma imperativo, che maschera i dettagli esecutivi del computer soggiacente, a scapito però di una minore efficienza di esecuzione di programmi.

Un *paradigma funzionale* esprime il calcolo effettuato da un programma come una funzione matematica. Le sue basi concettuali risiedono dunque nella matematica, e in particolar modo nella teoria delle funzioni ricorsive. I linguaggi funzionali consentono di esprimere in maniera sintetica ed elegante il procedimento di risoluzione di problemi anche complessi. Il capostipite dei linguaggi funzionali è il LISP, usato in passato prevalentemente per lo sviluppo di applicazioni nell'ambito dell'intelligenza artificiale. Altri linguaggi funzionali sono ML, Scheme[1], e Haskell.

Il modello computazionale dei *linguaggi logici* (*o dichiarativi*) invece si basa sulla logica (più precisamente, su sottoinsiemi del calcolo dei predicati del I ordine) per la descrizione dei programmi e

[1] Molte università hanno adottato Scheme come primo linguaggio di programmazione dei curricula di Informatica per la valenza formativa delle astrazioni fornite dai linguaggi funzionali (in particolar modo, l'uso della ricorsione). Esistono inoltre ottimi testi didattici basati sull'uso di Scheme.

sulle regole logiche di inferenza per la loro valutazione. Anche in questo caso, si tratta dunque di un paradigma fortemente radicato nella matematica. Il più noto linguaggio logico è il Prolog.
Passiamo ora ad analizzare i *paradigmi strutturali*. Questi classificano i linguaggi di programmazione in funzione delle modalità che essi offrono per dare una struttura ai programmi, scomponendoli in unità logiche. I due principali paradigmi strutturali sono quello procedurale e quello orientato agli oggetti (object-oriented). Entrambi i paradigmi prevedono una scomposizione di un programma complesso in parti, possibilmente caratterizzate da elevati livelli di autonomia e sviluppabili in maniera indipendente le une dalle altre. Entrambi i paradigmi, pertanto, rispondono al classico principio "divide et impera"; entrambi costituiscono degli strumenti per la modularizzazione di un programma. La scomposizione in moduli fa sì che il programma complessivo risulti più facile da leggere e da scrivere: risulta dunque più facile sia lo sviluppo iniziale del programma che la sua successiva manutenzione. La possibilità di sviluppo separato e indipendente delle diverse parti consente poi di ridurre i tempi complessivi di sviluppo delle applicazioni, rispetto ai tempi necessari per lo sviluppo di un'applicazione monolitica. I due paradigmi si differenziano per le astrazioni rese possibili dai meccanismi di modularizzazione. Come illustreremo tra breve, il paradigma procedurale consente una scomposizione per funzioni, mentre il paradigma object-oriented consente una scomposizione per oggetti.
Secondo il *paradigma procedurale*, un programma viene scomposto in sottoprogrammi. Tipici esempi sono le SUBROUTINE e FUNCTION del Fortran, le procedure del Pascal, le function del C. Un sottoprogramma consente di definire e dare nome a un'operazione complessa: per esempio, una funzione *sort* per ordinare un insieme di valori, una funzione *solve* per risolvere un sistema di equazioni, o una funzione *draw_map* per disegnare una mappa. Una volta definita, l'operazione complessa può essere utilizzata tutte le volte che ciò risulti necessario attraverso una semplice operazione di chiamata del sottoprogramma. Dal punto di vista della modularizzazione che così viene ottenuta, si può osservare dunque che risultano ben isolate le parti di programma che utilizzano le operazioni complesse da quelle che le definiscono. Eventuali cambiamenti interni al sottoprogramma, per esempio la codifica di un diverso algoritmo di ordinamento, all'interno del sottoprogramma *sort*, non hanno riflessi imprevisti sulle altre parti del programma e possono dunque essere da queste ignorate.
Il *paradigma object-oriented* scompone invece un programma in classi di oggetti. Una classe di oggetti è definita da un insieme di operazioni attraverso le quali gli oggetti della classe possono essere manipolati. Come semplice esempio, si supponga di voler rappresentare in un programma la classe di oggetti "poligoni regolari". Le operazioni che si vogliono definire per i poligoni regolari potrebbero essere il calcolo del perimetro, dell'area e il disegno sullo schermo del computer. Deve essere poi possibile creare diversi poligoni regolari, definendone la dimensione del lato. Un altro esempio è la classe fax, che definisce oggetti (i diversi dispositivi fisici fax), mediante i quali è possibile trasmettere e ricevere documenti. Infine, un altro esempio è costituito dai conti correnti bancari, per i quali le operazioni possibili sono depositi, prelievi e richieste di saldo, e dai correntisti di una banca, le cui operazioni possibili sono l'apertura e chiusura di un conto corrente o la richiesta di un mutuo.
Dagli esempi forniti, si può osservare che un oggetto è caratterizzato da uno stato (i dati nei quali vengono memorizzati valori che possono essere modificati) e da operazioni. In un linguaggio object-oriented pertanto gli oggetti sono definiti da un costrutto speciale, detto *classe*, di cui possono essere generati esemplari. Una classe raggruppa la definizione della struttura di dati e dei sottoprogrammi che implementano le operazioni. Inoltre una classe si comporta come un tipo dei convenzionali linguaggi procedurali. Così come è possibile generare, per esempio in FORTRAN o C, diverse variabili di tipo *integer*, allo stesso modo si possono generare diversi conti correnti e correntisti, esemplari delle rispettive classi. Inoltre, così come è possibile manipolare gli *integer* mediante apposite operazioni definite per essi dal linguaggio, allo stesso modo si possono manipolare conti correnti e correntisti con le operazioni per essi definite dalle rispettive classi. Si dice pertanto comunemente che il costrutto di classe consente di definire *tipi di dati astratti*.
I vantaggi in termini di strutturazione e modularità dei linguaggi object-oriented rispetto ai linguaggi procedurali sono notevoli. La possibilità

non solo di associare una struttura di dati alle operazioni che ne consentono la manipolazione, ma anche di rendere invisibile tale struttura di dati all'esterno del costrutto di classe (e cioè ai moduli di programma clienti) consente di ottenere sia una più razionale suddivisione in moduli che una più facile modificabilità dei programmi. Eventuali modifiche della struttura di dati che realizza lo stato degli oggetti non hanno effetto sui moduli clienti. La modifica della struttura di dati resta nascosta (incapsulata) all'interno del costrutto di classe.

Un'ulteriore caratteristica fondamentale dei linguaggi object-oriented è il costrutto di *sottoclasse*. Mediante questo costrutto è possibile definire oggetti che specializzano il comportamento della classe originaria. Per esempio, avendo definito la classe "fax", è possibile definire la sottoclasse "fax con telefono", che permette di comunicare non solo documenti, ma anche messaggi vocali. La sottoclasse definisce solo le nuove funzionalità che gli oggetti della sottoclasse forniscono, in quanto tutte le altre risultano *ereditate* dalla classe. La sottoclasse può inoltre ridefinire alcune delle funzionalità fornite dalla classe.

Il meccanismo di sottoclasse è un potente strumento linguistico a supporto della evoluzione del software. Per esempio, la modifica che aggiunge a un fax la funzionalità di trasmissione dei messaggi vocali viene ottenuta senza dover apportare modifiche globali al software esistente: basta definire le nuove funzionalità in una sottoclasse. Inoltre i programmi esistenti continuano ad operare correttamente anche se gli oggetti "fax" su cui operano non sono dei semplici fax, ma dei "fax con telefono", in quanto le operazioni di trasmissione e ricezione di testi continuano ad essere da questi supportate[2].

Il paradigma object-oriented è stato introdotto per la prima volta dal linguaggio SIMULA 67; esso è stato riscoperto più tardi da Smalltalk e infine diffuso, in una forma ibrida, dal C++. C++ è riuscito a diffondere i concetti della progettazione del software object-oriented in quanto essi sono stati aggiunti ai concetti della tradizionale programmazione procedurale offerta dal linguaggio C, consentendo in tal modo un'adozione graduale del nuovo paradigma. I linguaggi Java e C#, che stanno avendo una larga diffusione, adottano in maniera più coerente il paradigma object-oriented.

4. Ulteriori classificazioni e conclusioni

I paradigmi computazionali e strutturali consentono di classificare i linguaggi di programmazione, e quindi di meglio coglierne le caratteristiche distintive principali. Altre forme di classificazione sono pure possibili, e spesso molto utili. Si potrebbe distinguere i linguaggi a seconda dello stile mediante il quale si definiscono i programmi. In tal caso, si potrebbe distinguere tra linguaggi testuali e linguaggi grafici, o visuali. Altre distinzioni potrebbero essere fatte tra i linguaggi compilati e linguaggi interpretati, o tra i linguaggi con tipizzazione statica e quelli con tipizzazione dinamica. Nel primo caso, si vuole distinguere tra i linguaggi per i quali sono prevalenti le implementazioni basate sulla traduzione verso un linguaggio macchina (o un linguaggio vicino ad essa) e quelli che vengono implementati mediante una diretta interpretazione dei programmi nella forma sorgente[3]. Tipici linguaggi interpretati sono LISP e molti cosiddetti "linguaggi di script" che oggi sono molto diffusi (per esempio, Python o più recentemente Ruby). Normalmente invece i linguaggi sono compilati, anche se sono comuni situazioni ibride, come quella adottata da Java, che non viene di solito compilato in linguaggio macchina, ma in un linguaggio intermedio, detto Bytecode, che viene poi interpretato. La seconda classificazione invece distingue tra linguaggi in cui ogni variabile ha associato in maniera statica un tipo, che caratterizza gli oggetti a cui può fare riferimento, rispetto a quelli in cui questo tipo può cambiare dinamicamente. Nel primo caso, il linguaggio consente di verificare, prima dell'esecuzione, che i dati vengano manipolati in maniera corretta; nel secondo, questo

[2] Tecnicamente, questo importante risultato è dovuto a due caratteristiche dei linguaggi object-oriented: il binding dinamico e il polimorfismo.

[3] Si potrebbe obiettare che stiamo in realtà distinguendo due modi per implementare i linguaggi piuttosto che diverse tipologie di linguaggi. In realtà, approfondendo questa questione si arriverebbe a osservare che proprio alcune caratterstiche dei linguaggi favoriscono l'una o l'altra soluzione implementativa.

non risulta possibile (se non imponendo vincoli specifici). Intuitivamente, i linguaggi che ricadono all'interno della prima categoria supportano una maggiore affidabilità dei programmi, in quanto possono garantire a priori l'assenza di certi errori durante l'esecuzione.
Per concludere questa breve trattazione sui linguaggi di programmazione, sottolineiamo come sia fondamentale che uno specialista di informatica padroneggi i concetti che ne stanno alla base. I linguaggi di programmazione sono gli strumenti principali per la produzione di software, e questo sta alla base delle applicazioni che si usano quotidianamente in ogni settore della vita odierna. La qualità delle applicazioni e, in primis, la loro affidabilità, è fortemente dipendente dal linguaggio di programmazione che è stato utilizzato per lo sviluppo. Capire a fondo i linguaggi significa dunque non solo poterli usare al meglio nello sviluppo delle applicazioni ma, in ultima analisi, sviluppare prodotti di qualità.

Bibliografia

Elencare i riferimenti ai diversi linguaggi di programmazione porterebbe a una bibliografia sterminata. Si riporta dunque soltanto il riferimento a un testo di carattere generale che contiene un'analisi dei concetti dei linguaggi, oltre ai riferimenti bibliografici ai diversi linguaggi analizzati.

[1] Ghezzi C., Jazayeri M.: *Programming Language Concepts*. (III Edition), J. Wiley and Sons, 1997 (la traduzione della II edizione di questo testo è disponibile in lingua Italiana col titolo Concetti dei Linguaggi di Programmazione, Editore Franco Angeli, 1989).

Carlo Ghezzi è Professore Ordinario di Ingegneria del Software, membro del Senato Accademico del Politecnico di Milano e responsabile scientifico dell'area di ricerca sull'Ingegneria del Software presso il CEFRIEL. È Fellow dell'ACM (*Association for Computing Machinery*) e dell'IEEE e Editor in Chief della rivista ACM *Transactions on Software Engineering and Methodology*. È autore di numerosi articoli scientifici e libri. Ha interessi di ricerca nell'ambito dei linguaggi di programmazione e dell'ingegneria del software, con particolare riguardo ai fondamenti teorici, metodologici e tecnologici dello sviluppo delle applicazioni distribuite su rete.
E-mail: carlo.ghezzi@polimi.it