Acc. Sc. Torino - Quaderni
xx (2002), 1-..., x ff, x t, ,x tabb

Comunicazione uomo-uomo e comunicazione uomo-macchina (per tacer delle donne): linguaggi naturali e artificiali.

Simona Ronchi Della Rocca

Credo di dover innanzitutto spiegare la polemica contenuta nel titolo. Il linguaggio della scienza, così come quello quotidiano, è ancora ben lontano dall'essere politicamente corretto, dal punto di vista del genere; nell'informatica, scienza in cui io lavoro, *interazione uomo-macchina* è il nome ufficiale di una materia di insegnamento. Del resto, sull'invito alla conferenza da cui questo articolo è tratto, io sono stata presentata come *ordinario* di Fondamenti dell'Informatica e *socio* dell'Accademia delle Scienze. Insomma, nella scienza, o le donne non esistono, o quando si accetta la loro esistenza, vengono trasformate in uomini. Quando le donne avranno diritto di cittadinanza, nella scienza?

Ora posso introdurre il tema di questo articolo.

I linguaggi artificiali, o linguaggi di programmazione, sono gli strumenti con cui si comunica con i calcolatori, ed in particolare li si *istruisce* sui compiti che si desidera vengano svolti. Un generico calcolatore è in grado di svolgere un insieme vastissimo di compiti, che vanno dal calcolo numerico alla video-grafica, dalla simulazione di realtà virtuali, al controllo di processi meccanici.
Un calcolatore è una struttura molto complessa, che possiamo immaginare fatta a strati come una cipolla: lo strato più interno è quello dove effettivamente vengono compiute le operazioni, che sono pochissime e molto elementari: essenzialmente operazioni aritmetiche, spostamento di informazioni da un posto all'altro, confronto tra diversi valori. Queste operazioni sono codificate in *linguaggio-macchina*, le cui parole sono stringhe binarie (cioè composte solo di due caratteri, 0 e 1): un linguaggio contemporaneamente elementare e difficilissimo da capire per un essere umano. In effetti chi vuole programmare un calcolatore non usa questo linguaggio, ma usa un linguaggio cosiddetto *di programmazione*, che ha una struttura più vicina ai linguaggi naturali ed è quindi più facilmente comprensibile per una persona. Il *programma*, cioè la sequenza di istruzioni scritta in questo linguaggio, viene poi automaticamente analizzato e, tramite diversi passi di traduzione, riscritto nel linguaggio-macchina: a questo punto la sequenza di comandi da esso codificata può essere eseguita. Quindi i linguaggi di programmazione devono essere comprensibili sia per le persone umane che per le macchine. Esiste un gran numero di linguaggi di programmazione, alcuni disegnati espressamente per agire in ambienti specifici, altri (la maggior parte) di carattere generale, che si prestano a programmare praticamente qualunque tipo di applicazione. Ma che cosa hanno in comune tutti questi linguaggi? Quali caratteristiche li distinguono dai linguaggi naturali, quelli che usiamo per comunicare tra esseri umani? Perché ne esistono tanti?

Un linguaggio, sia esso naturale o artificiale, presenta due aspetti: la sintassi, cioè l'insieme di regole che ci permette di costruire frasi corrette, e la semantica, che assegna un significato alle frasi. Per i linguaggi di programmazione, bisogna che tutti e due questi aspetti siano automatizzabili. Infatti i procedimenti di traduzione, dal linguaggio originale in linguaggio-macchina, possono essere effettivi solo se è possibile in modo automatico:

1. verificare se il programma è sintatticamente corretto (*analisi sintattica*),
2. in caso positivo, assegnargli un significato (*analisi semantica*).

La fattibilità dell'analisi sintattica si raggiunge tramite le *grammatiche formali*, che definiscono in modo esaustivo le regole sintattiche. Un esempio molto semplice è la grammatica per generare i numerali in notazione decimale corrispondenti a numeri interi non negativi. La grammatica è formata dalle seguenti due regole:

1. <cifra>::= 0|1|2|….|9
2. <numero>::= <cifra>|<numero><cifra>

Le parole tra parentesi acute definiscono classi di oggetti del linguaggio. Una regola si intepreta nel modo seguente: il simbolo a sinistra del segno ::= può essere sostituito da una delle sequenze di simboli che occorrono alla sua destra, separate dal segno |. Un simbolo non parentetizzato rappresenta un oggetto del linguaggio, e quindi non può più essere sostituito (viene anche detto *oggetto terminale* della grammatica). Nel caso particolare, la regola 1 definisce cos'è una cifra, dando l'insieme dei valori che essa può assumere, la regola 2 descrive le modalità di costruzione di un numero, che o coincide con una

cifra o si può ottenere da un numero precedentemente costruito semplicemente postponendogli una cifra.
In generale, per un linguaggio di programmazione esistono varie grammatiche, dipendenti una dall'altra, che lo definiscono: una grammatica per le parole, una per le frasi, e una per i discorsi compiuti, o programmi.

La differenza essenziale tra linguaggi artificiali e quelli naturali è nella semantica. I linguaggi naturali servono per comunicare con esseri (presumibilmente) intelligenti, mentre l'interlocutore dei linguaggi di programmazione è un'entità assolutamente priva di intelligenza: una macchina. Tutte le ricchezze del linguaggio naturale sono perciò bandite dai linguaggi di programmazione, in particolare l'ambiguità semantica. L'ambiguità semantica permette che una frase abbia più di un significato, a seconda del contesto in cui si viene a trovare, o non ne abbia nessuno se non in un contesto simbolico o metaforico. Una frase semanticamente ambigua è la seguente: "La vecchia porta la sbarra". L'ambiguità è evidente se la considera inserita in due diversi contesti, nel modo seguente: " La vecchia porta la sbarra: lo sforzo piega il suo corpo stanco" e "La donna fugge correndo dal suo assalitore verso la cantina, ma la vecchia porta la sbarra". La frase è ambigua in quanto alcune parole che la compongono possono avere doppio significato (porta e sbarra), ma si possono costruire frasi semanticamente ambigue anche a partire da parole che ambigue non sono. Consideriamo la frase: "L'uomo guardava la ragazza col cannocchiale": il cannocchiale può essere il mezzo con cui l'uomo guarda la ragazza, e questa è l'interpretazione più naturale, ma potrebbe anche essere tenuto in mano dalla ragazza. Noi non facciamo normalmente caso all'ambiguità del nostro linguaggio, in quanto automaticamente capaci (quasi sempre!) di discernere il significato corretto, ma essa permea la comunicazione tra gli esseri umani e offre loro possibilità di esprimere costrutti raffinati come l'ironia, le metafore, la poesia.
Però è esperienza comune la necessità di usare un linguaggio tanto più semplice e preciso quanto più è stupido il nostro interlocutore (con tutto il rispetto per gli stupidi!). Quindi, poiché la macchina è l'essere più stupido in assoluto, i linguaggi di programmazione devono essere tali che ogni frase abbia *uno e un solo significato*. La non ambiguità semantica si ottiene in modo incrementale: si associa un significato ad ogni parola, e poi si definisce il significato di una frase in funzione del significato delle singole parole, e così via…

Inoltre, i linguaggi di programmazione devono possedere un'altra proprietà: la *sostituibilità semantica*. Cioè sostituendo una parte di un discorso (o programma) con un'altra di uguale significato, il significato dell'intero discorso non deve cambiare. Nei linguaggi naturali la sostituibilità semantica non vale, e questo è conseguenza non solo dell'ambiguità semantica, ma di due altri fattori: l'uso di frasi definitorie, e la possibilità di usare la stessa parola in veste di *significante* e di *significato.* Consideriamo la frase definitoria:" Roma è la capitale d'Italia": in essa non possiamo sostituire la parola Roma con il suo significato, altrimenti otteniamo la frase: "La capitale d'Italia è la capitale d'Italia", che è una tautologia e ha un significato diverso rispetto alla definizione di partenza. Nella frase: "otto è palindroma", la parola "otto" è usata come significante, non come significato, quindi la sua sostituzione con "quattro più quattro", che pure ha lo stesso significato, rende la stessa frase un non senso. Nei linguaggi di programmazione non è che queste due costruzioni non si possano esprimere, ma si esplicita nella sintassi il diverso uso delle parole. Quindi la definizione si introduce con una costruzione sintattica particolare, e quando una parola è usata come significante si esplicita questo uso con un'apposita funzione.
La sostituibilità semantica permette di riutilizzare programmi come pezzi di altri programmi, e quindi di risparmiare lavoro.

I linguaggi di programmazione, inoltre, devono possedere una potenza espressiva apparentemente enorme, adeguata per programmare ogni possibile applicazione.

In conclusione, i linguaggi di programmazione sono linguaggi che devono possedere le seguenti caratteristiche: una sintassi precisa, una semantica non ambigua, la sostituibilità semantica e un potere espressivo che ci permetta di dare a un calcolatore qualunque ordine esso sia in grado di svolgere.
Sono caratteristiche molto chiare, ma non suggeriscono nulla sulla struttura di tali linguaggi: come si fa a inventare un linguaggio artificiale? A cosa ci si può ispirare?

Per quanto apparentemente molto diversi l'uno dall'altro, i linguaggi di programmazione si possono raggruppare in classi, o paradigmi, ciascuno dei quali nasce da un'idea astratta di calcolo. E' interessante notare come alcuni dei paradigmi più usati nascano da un'idea di programmazione nata prima dell'invenzione del calcolatore stesso, e precisamente da un problema che i matematici si sono

posti intorno agli anni ’30: quali funzioni matematiche possono essere *effettivamente calcolate*? Una funzione è una relazione che lega in modo deterministico una sequenza di valori dati (detti *argomenti*) a uno specifico *risultato*. Un esempio è la funzione *somma* di numeri interi, che ad ogni coppia di numeri interi associa il valore della loro somma (e noi sappiamo per esperienza che questa funzione può essere effettivamente calcolata).
A questa domanda hanno risposto diversi grandi matematici, come Turing, Kleene, Church, Post e altri. Per rispondere, ciascuno di essi ha innanzitutto dovuto definire cosa significa *calcolare effettivamente*. Tutti hanno concordato sul fatto che calcolare effettivamente significa *calcolare in modo automatico*, ma, poiché allora non esistevano i calcolatori, ciascuno di essi ha definito una propria nozione di *calcolo automatico*, con risultati tra di loro molto diversi. Il risultato interessante è stato che, se pure partendo da definizioni di computazione diverse, tutti hanno trovato la stessa classe di funzioni, che appunto viene definita *classe delle funzioni calcolabili*.
Poiché tra i compiti che il calcolatore deve svolgere c’è quello di calcolare, ogni linguaggio di programmazione deve essere in grado di esprimere *almeno* tutte le funzioni calcolabili.

Vediamo ora due delle nozioni di calcolo automatico allora definite, quella di Church e quella di Turing, perché sono quelle che hanno dato origine ai due primi paradigmi di programmazione.

Per Church il calcolo automatico corrisponde ad un procedimento iterativo di semplificazione di espressioni ottenuto applicando regole predefinite: la macchina che calcola è quindi simile ad un essere umano che applica regole che non necessariamente capisce. Un esempio è il calcolo di espressioni aritmetiche. Ammettiamo di saper calcolare la somma e il prodotto di due numeri (tramite gli algoritmi che ci hanno insegnato a scuola). Per calcolare il valore di una espressione aritmetica, basterà seguire questa semplice procedura: leggere l’espressione da sinistra a destra, fino ad incontrare la prima espressione immediatamente calcolabile (una somma o prodotto di due numeri), quindi sostituire ad essa il suo valore, e ricominciare la lettura da sinistra a destra dell’espressione nuova così ottenuta. Quando non si incontrano più sotto-espressioni da calcolare, abbiamo trovato il valore cercato. La procedura è completamente meccanizzabile. Church non solo utilizzò una procedura simile per definire le funzioni calcolabili, ma trovò anche che per calcolarle tutte è sufficiente una sola operazione: la sostituzione. Naturalmente il linguaggio che egli ha definito, il lambda-calcolo, non è di facile lettura, in quanto così minimale. Ma da esso è nato un insieme di linguaggi di programmazione, i cosiddetti *linguaggi funzionali* il cui padre è il LISP, che è stato molto usato per il trattamento di dati non numerici.

Turing invece ha definito una nozione di macchina basata su componenti meccaniche, a cui si è poi effettivamente ispirato il progetto del calcolatore di Von Neumann. La macchina di Turing è una macchina astratta composta di un nastro infinito, diviso in celle, ognuna delle quali contiene un simbolo, e una unità di controllo, che possiede una testina mobile che punta ad ogni istante ad una cella del nastro. L’unità di controllo è in grado di eseguire operazioni del tipo:
1. leggere il simbolo puntato dalla testina
2. sostituirlo con un altro simbolo
3. spostarsi sulla cella precedente, o su quella successiva, del nastro.

Calcolare con una macchina di Turing significa eseguire una sequenza di operazioni del tipo precedentemente descritto, su di un nastro contenente all’inizio dell’esecuzione i dati iniziali del calcolo e alla fine il risultato.
Le due componenti la macchina di Turing, il nastro e l’unità di controllo, corrispondono alla memoria e all’unità di controllo dei calcolatori reali. Dalla macchina di Turing sono nati sia il calcolatore reale che i linguaggi di programmazione cosidetti *imperativi,* come il PASCAL o il C. L’aggettivo imperativo deriva dal fatto che le operazioni sopra citate possono essere viste come *comandi* alla macchina.

La differenza essenziale tra linguaggi funzionali e imperativi è che, mentre i primi sono basati su di un linguaggio di tipo matematico, nei secondi si possono scrivere frasi che nel normale linguaggio matematico sono dei non-sense, come la frase “x = x+1”, in un ambiente in cui si assume che la variabile x indichi un generico numero. Nel linguaggio C, la frase x=x+1 corrisponde al comando “metti nell’area di memoria chiamata x il valore che ora è in x aumentato di 1”. Quindi se l’area chiamata x conteneva, prima dell’esecuzione del comando, il numero 3, dopo la sua esecuzione conterrà il numero 4.

In genere i linguaggi funzionali sono più facilmente comprensibili, la loro semantica è modellabile matematicamente in modo rigoroso, e ci sono strumenti raffinati per controllare e rendere facile e

corretta la programmazione in questi linguaggi. I linguaggi imperativi sono meno comprensibili per gli esseri umani, ma, essendo più legati alla struttura della macchina, sono più efficienti.

Ho scelto finora, per facilità, esempi tratti dall'aritmetica, ma ai calcolatori viene sempre meno richiesto di calcolare, e sempre più invece di manipolare informazioni che non sono necessariamente numeriche. Ma la struttura dei linguaggi di programmazione funzionali e imperativi, che pure permettono un ampio spettro di operazioni non numeriche, è basata sostanzialmente sull'affermazione che *programmare = calcolare*. Ma vi è anche una stretta relazione tra la programmazione e la logica matematica, cioè si è visto che la programmazione può anche essere associata alla dimostrazione di teoremi.
Nella logica classica, possiamo dividere le dimostrazioni in costruttive e non costruttive. Ad esempio, supponiamo di voler dimostrare che, dato un insieme finito di persone, ne esiste sempre almeno una più giovane di tutte le altre. Ne darò due diverse dimostrazioni, una costruttiva e una non costruttiva. Per dimostrare costruttivamente il teorema, si può far vedere come si possa appunto "costruire" il sottoinsieme delle persone più giovani. Si possono radunare tutte le persone da esaminare in una stanza, collegata da due porte a due altre stanze, a loro volta in comunicazione tra di loro. In una di queste ultime saranno inserite le persone che man mano saranno classificate come "le più giovani", nell'altra quelle "scartate". Scegliamo una persona a caso, e invitiamola ad entrare nella stanza delle "più giovani". Poi, per ogni altra persona P nella stanza, seguiamo la seguente procedura:

1. se P è più giovane di quelle "più giovani" , spostiamo tutte le persone "più giovani" in "scartate" e inseriamo P nella stanza delle "più giovani";
2. se P è della stessa età di quelle "più giovani" inseriamo P in "più giovani";
3. altrimenti inseriamo P in "scartate".

E' facile vedere che in ogni istante la stanza delle persone "più giovani" contiene tutte persone della stessa età, che sono tutte più giovani di quelle "scartate". Poiché il numero iniziale di persone è finito, quando la stanza che inizialmente conteneva tutte le persone da esaminare sarà vuota, la stanza delle "più giovani" conterrà esattamente le persone più giovani di tutte.

La dimostrazione non costruttiva procede per assurdo. Assumiamo per assurdo che non esista nemmeno una persona più giovane di tutte le altre. Questo equivale a dire che, per ogni persona P nella stanza, ne esiste almeno un'altra più giovane di lei. Per dimostrare la proprietà bastano solo due stanze, una che conterrà all'inizio tutte le persone da esaminare e una stanza per quelle "scartate". Poi, per ogni persona P nella stanza, cerchiamo una persona che sia più giovane di P (c'è sempre, per ipotesi) e quindi spostiamo P nella stanza delle "scartate". Poiché il numero iniziale di persone è finito, si arriverà a considerare l'ultima persona nella stanza. Sia P l'ultima persona nella stanza. Poiché per ipotesi deve esistere almeno un'altra persona più giovane di P (sia Q) allora Q deve trovarsi tra quelle già "scartate". Ma per costruzione quelle già scartate sono tutte più vecchie di P, e allora Q deve essere contemporaneamente più vecchia e più giovane di P. Ma questo è un assurdo, e quindi è sbagliata l'ipotesi iniziale, che asseriva la non esistenza di persone più giovani.

La differenza fondamentale tra le due dimostrazioni è che, mentre la prima definisce un procedimento effettivo per costruire l'oggetto la cui esistenza voglio dimostrare, la seconda ne dimostra l'esistenza senza costruirlo esplicitamente. La dimostrazione costruttiva può essere quindi vista (tramite un processo di traduzione che si può rendere automatico) come un programma che, nel nostro esempio, dato l'elenco dei dati anagrafici di un numero finito di persone, restituisce i nominativi dei più giovani. Quindi, se ci limitiamo a logiche costruttive, possiamo porre l'equazione *programmare = dimostrare*.

In particolare, esiste un paradigma di programmazione, che si basa appunto sulla associazione tra programmazione e dimostrazione di teoremi, che si chiama *paradigma logico*. In generale non è decidibile, dato una frase in linguaggio logico, se questa è vera o falsa, cioè se corrisponde o no a un teorema (altrimenti, non ci sarebbe più lavoro per i matematici e anche per molti informatici!). Ma esistono logiche particolari in cui questo è decidibile. I linguaggi logici sfruttano questa proprietà per *far generare i programmi direttamente dal calcolatore*. Il programmatore scrive il possibile enunciato del teorema, e poi automaticamente il calcolatore (per mezzo ovviamente di un altro programma, scritto una volta per tutte) controllerà se tale enunciato è vero. In caso positivo, la dimostrazione così generata è il programma cercato.

Le relazioni tra logica e programmazione sono interessanti, ma attenzione a non estrapolare troppo i risultati. In logica, l'autoreferenzialità genera paradossi. Pensiamo al classico paradosso di Russel:"sia X l'insieme di tutti gli insiemi che non appartengono a se stessi. X appartiene a se stesso?" Il paradosso nasce dal fatto che, per definizione, X appartiene a se stesso se e solo se X non appartiene a se stesso.

Questo paradosso ha molte versioni divulgative, di cui però nessuna veramente rigorosa. Una classica è la storia del paesino di frontiera in cui non esiste un barbiere. Il sindaco, che ama l'ordine e non ama le barbe, assolda un barbiere municipale, e annuncia che tutti dovranno radersi la barba, e che quelli che non se la raderanno da soli dovranno pagare una apposita tassa e farsela radere dal barbiere. Ma il barbiere deve o no pagare la tassa?
Un altro esempio classico di paradosso generato dall'auto-referenzialità è la semplice affermazione: "Questa frase è falsa", a cui non si può assegnare nessun valore di verità.
Nell'informatica l'autoreferenzialità non solo non è problematica, ma è lo strumento principale di calcolo. Ho detto all'inizio che ogni programma, prima di essere eseguito, passa attraverso vari stadi di traduzione, finchè viene tradotto nel linguaggio eseguibile. Gli stadi di traduzione sono effettuati a loro volta da programmi. Assumiamo quindi che P sia un programma che traduce dal linguaggio A al linguaggio B, cioè per ogni programma Q_A scritto in linguaggio A, P, applicato a Q_A, lo trasforma in un programma Q_B che ha lo stesso significato ma è scritto in linguaggio B. P, essendo un programma, deve essere scritto in un linguaggio di programmazione, in particolare può essere scritto in linguaggio A. In questo caso può essere applicato a sè stesso e trasformarsi in un programma equivalente scritto in linguaggio B.

L'ultimo paradigma di programmazione che vorrei illustrare è quello *orientato agli oggetti*, che, diversamente dagli altri, non nasce da un concetto astratto di programmazione ma da una problematica molto concreta. Un programma complesso è difficile da capire, anche per gli addetti ai lavori, e c'è in questo campo un grande spreco di risorse: di fronte alla necessità di un programma che risolva un dato problema, è generalmente considerato più semplice riscriverlo completamente piuttosto che riutilizzare un programma già esistente che risolve lo stesso problema, ma che avrebbe bisogno di essere un po' modificato, per adattarlo allo specifico contesto in cui vogliamo utilizzarlo. I linguaggi appartenenti a questo paradigma sono stati progettati per facilitare il riutilizzo dei programmi. L'ispirazione è l'organizzazione del lavoro cooperativo: in questo paradigma un programma è composto di piccoli pezzi (gli *oggetti*) che comunicano tra di loro, ciascuno dei quali svolge un compito specifico. L'idea è che questi pezzi possano essere innanzitutto facilmente comprensibili, date le loro piccole dimensioni, e che possano poi essere utilizzati in contesti diversi. Il lavoro del programmatore si trasforma quindi in quello di un assemblatore, che scompone il problema da risolvere in sotto-problemi elementari, affida la soluzione di ognuno di essi a programmi preesistenti tratti da grandi librerie, e poi compone i risultati così ottenuti in modo opportuno, ottenendo così un programma nuovo che risolve il problema iniziale. L'appellativo "orientato agli oggetti" è decisamente brutto, e non evocativo, perchè nel linguaggio quotidiano un oggetto è qualcosa di passivo, mentre qui ogni oggetto svolge un compito. Storicamente si giustifica proprio nella inversione attivo-passivo. Un elenco telefonico è un oggetto che contiene dei dati, ma possiamo pensarlo come un'entità attiva, che, quando riceve un'informazione composta di nome, cognome e indirizzo, risponde con il corrispondente numero di telefono.

Ma non ho ancora provato a rispondere ad una delle domande che ho posto all'inizio: perché ci sono tanti diversi linguaggi di programmazione, se poi hanno tutti le stesse potenzialità? In effetti la sovrabbondanza deriva dal fatto che, essendo tanti e molteplici i compiti che un calcolatore può svolgere, alcuni linguaggi si prestano meglio di altri per realizzare specifiche applicazioni. Infatti i linguaggi funzionali sono particolarmente adatti per "programmare all'ordine superiore", cioè per manipolare programmi, e realizzare quindi traduttori e intepreti, i linguaggi imperativi per definire compiti strettamente legati alla struttura fisica del calcolatore, e quindi per programmare sistemi operativi o per controllare processi, i linguaggi logici per compiti legati alla descrizione della conoscenza e applicazioni di intelligenza artificiale, i linguaggi orientati agli oggetti per realizzare applicazioni distribuite, che possano viaggiare sulla rete. Il che fa pensare che la necessità futura di nuove applicazioni porterà al disegno di nuovi linguaggi…sperando che la molteplicità rimanga una ricchezza espressiva, e non generi alla fine una "torre di Babele" in cui nessuno si comprenderà più.

BIBLIOGRAFIA

ABELSON H., SUSSMAN G.J., *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

DE LIGUORO U., RONCHI DELLA ROCCA S., *Teoria della Programmazione e Lambda-Calcolo*, in: Il futuro del Computer, Quaderni di "Le Scienze", 121, 2001.

SETHI R., *Programming Languages*, Addison-Wesley, 1989.

STOY J., *Denotational Semantics: the Scoot-Strachey approach to programming language theory*, M.I.T Press,1977.