Informatica: Elogio di Babele

Simone Martini
Alma Mater Studiorum - Università di Bologna
Dipartimento di Scienze dell'Informazione

In memoria di Alessandro Perutelli, 1947-2007
Aracne Editore, in stampa

L'unità dei linguaggi

La nascita dell'informatica, quale scienza che studia i procedimenti di calcolo "effettivi", precede di almeno dieci anni la disponibilità di quei manufatti che chiamiamo *calcolatori* (*computer*) e l'utilizzo dei quali oggi rischiamo di confondere con quella scienza[1]. Lo studio del calcolabile prende una sua forma matura nei primi anni trenta del novecento, nel cuore della logica matematica, per opera di K. Gödel, A. Church, S. Kleene, E. Post e, soprattutto, Alan M. Turing. Mediante un'analisi dei processi di calcolo umani (in particolare delle limitazioni della memoria e della percezione), Turing identifica il "calcolo" con la possibilità di manipolazione combinatoria di insiemi finiti e discreti di simboli: calcolare è nient'altro che copiare simboli (tratti da un alfabeto finito) secondo regole anch'esse finite e fissate in anticipo. Il modello è così semplice da poter essere visualizzato come un automa (la "macchina di Turing") che, in accordo alle regole memorizzate in una lista finita, sposta simboli su un nastro, al quale accede un simbolo alla volta. L'aspetto sorprendente dell'analisi di Turing è che questo modello così elementare ed apparentemente rudimentale, così basato sull'analisi dei processi di calcolo umani[2], è un modello del tutto generale, tanto da includere una macchina che è in grado di emulare tutte le altre.

Consideriamo, infatti, l'esecuzione di una macchina su un suo dato di ingresso. La macchina funziona in modo deterministico, applicando le regole ai dati. Se abbiamo una descrizione della macchina (ovvero un elenco delle regole in base alle quali opera) possiamo seguire la sua esecuzione, passo passo, con carta e matita. Questo procedimento (partendo da una descrizione della macchina e da un dato di ingresso, seguire l'esecuzione passo passo della macchina) è un procedimento di calcolo. Turing mostra che anche *questo* calcolo[3] è esprimibile da una macchina. Riassumendo: tra tutte le macchine di Turing ce n'è una che è in grado di simulare tutte le altre (la macchina *universale*). Non importa "costruire" una specifica macchina per ogni calcolo di nostro interesse. Basta costruire la macchina universale e fornirle la

[1] «Computer science is no more about computers than astronomy is about telescopes», E. W. Dijkstra, premio Turing 1972. Il premio Turing, il più importante riconoscimento internazionale per un informatico, è attribuito ogni anno dall'associazione professionale degli informatici americani (ACM).

[2] «Turing's "machines": These machines are humans who calculate», L. Wittgenstein, Remarks on the Philosophy of Psychology, Vol. 1, Blackwell, Oxford, 1980.

[3] Quello, cioè, che, data una descrizione della macchina e un particolare dato esegue (simula) la macchina su quel dato.

descrizione (il *programma*) del calcolo che vogliamo eseguire. La macchina universale eseguirà per noi il programma sui dati di nostra scelta.

Sarà John von Neumann a rendersi conto presto delle potenzialità tecnologiche insite in quest'analisi, progettando un manufatto che avrebbe realizzato, con valvole e fili, una macchina di Turing universale, cioè un *computer*, non più *human*, ma *hardware which calculates*[4]. I calcolatori che stanno sulle nostre scrivanie sono, ancor oggi, variazioni tecnologicamente avanzate di quel progetto.

A noi interessa, però, non l'aspetto tecnologico, ma una lettura linguistica. Non vi è calcolo senza linguaggio, ma la varietà dei linguaggi non produce concetti diversi di calcolo, perché tutti sono codificabili nella semplice macchina di Turing universale. Un solo linguaggio (che può essere reso tanto semplice da includere due soli simboli) esaurisce (mediante codifica) tutto l'esprimibile.

Non c'è solo Turing, a studiare i processi di calcolo effettivo. Negli stessi anni, altri descrivono in cosa consiste "calcolare", partendo da presupposti ed analisi diverse; è quindi naturale che ottengano modelli diversi. Alonzo Church introduce un formalismo che chiama "λ-calcolo" ed è basato sulla riscrittura successiva di sequenze di caratteri che corrispondono ad espressioni simboliche; Kurt Gödel un modo di definire funzioni per "ricorrenza" (o ricorsione); Emil Post un sistema di celle contigue che si influenzano tra loro e che evolvono per passi discreti. E in seguito, quando la disponibilità di calcolatori elettronici richiederà di poterli programmare in modo sempre più semplice ed efficace, saranno introdotte altre centinaia di sistemi diversi (i *linguaggi di programmazione*) per descrivere il calcolo. Molti sono oggi dimenticati, altri sono ancora in uso dopo più di cinquant'anni (FORTRAN, per esempio, o LISP), altri ancora sono introdotti o aggiornati in anni recenti (uno tra tutti: Java). L'unità dell'analisi di Turing, il suo ridurre il calcolo all'unità di una sola macchina, si infrange nella moltiplicazione dei linguaggi e dei formalismi, in una Babele in cui non è sempre facile per il principiante mettere ordine.

Già i padri fondatori, tuttavia, sapevano che questa moltiplicazione di linguaggi non intacca la profonda unità, l'invarianza del concetto sottostante. Perché si dà il caso che tutti questi linguaggi sono solo definizioni diverse dello stesso concetto. Se una certa corrispondenza tra numeri naturali (una funzione) è calcolabile secondo Turing, allora è calcolabile anche secondo Church, o Kleene, o con un qualsiasi altro linguaggio di programmazione. E viceversa. Il modo di *descrivere* il calcolo è diverso, ma il concetto di "cosa è calcolabile" è invariante rispetto alla descrizione. Il concetto di "calcolabile" è, in un certo senso, un assoluto della natura, e non un aspetto legato al linguaggio nel quale lo descriviamo: i linguaggi sono diversi, ma sono equivalenti quanto a "cosa" possono calcolare. Questa equivalenza tra sistemi formali viene subito

[4] Cfr. nota 2.

osservata da Church e Turing, e poi dimostrata per ogni altro formalismo per il calcolo (che chiameremo genericamente: linguaggio di programmazione).

Non ci interessa qui discutere come Church e Turing dimostrano questa equivalenza, quanto vedere la forma che essa assume a partire dagli anni cinquanta. Un linguaggio di programmazione è, nella nostra definizione, un sistema formale per la descrizione del calcolo. Si tratta di linguaggi artificiali, alcuni assai rudimentali (quali i primi sistemi formali per la calcolabilità, come la macchina di Turing, cui abbiamo accennato), altri di una notevole sofisticazione (come i cosiddetti linguaggi ad altissimo livello delle ultime generazioni). Dal punto di vista della linguistica, sono comunque linguaggi semplicissimi, la cui sintassi è definita in modo formale con grammatiche generative alla Chomsky[5], e la cui semantica è definita in modo che ad ogni frase corretta corrisponda un significato univoco. Ora, sappiamo dall'analisi di Turing e Church che, data una funzione calcolabile *f* in un certo formalismo *L1*, questa è calcolabile anche in un altro formalismo *L2*. Possiamo dirlo in modo più esplicito: dato un programma che calcola *f* scritto in *L1*, esiste un programma che ancora calcola *f* ed è scritto in *L2*. Questa corrispondenza tra *L1* e *L2* può esser vista come una traduzione: ogni frase corretta (cioè ogni programma) di *L1* può esser tradotta in una frase *equivalente* (cioè un programma che calcola la stessa funzione) scritta in *L2*. Vista la semplicità dei linguaggi in considerazione, questa traduzione può essere "facilmente" automatizzata. Ovvero, ancora una volta, espressa come calcolo: il traduttore stesso da *L1* a *L2* può essere dato come un programma (in gergo si chiama il *compilatore* da *L1* a *L2*), scritto in qualche linguaggio. Possiamo riassumere e sintetizzare queste osservazioni in un enunciato più formale:

> Dati due qualsiasi linguaggi di programmazione *L1* e *L2*, esiste un programma $C_{L1\Rightarrow L2}$ (un *compilatore* da *L1* a *L2*) tale che, per ogni programma *P* scritto in *L1*, $C_{L1\Rightarrow L2}(P)$ è un programma in *L2* equivalente a *P*.

Se il lettore ci perdona, possiamo spingere questa prima riflessione ancora un passo in avanti. Il compilatore da *L1* a *L2* può essere espresso esso stesso in *L1*: nel linguaggio *L1* descriviamo il processo necessario a tradurre ogni frase di *L1* in una frase di *L2*[6].

La pluralità dei linguaggi, dunque, non è che un epifenomeno della sottostante, robusta invarianza del concetto di calcolo. La traduzione realizza la *reductio ad unum*, riconducendo ogni linguaggio ad un altro, e lo fa in modo "uniforme", essa stessa codificata in un calcolo, nello stesso linguaggio che viene tradotto.

---

[5] Anzi, sono grammatiche di tipo 2 (cioè non contestuali), alle quali sono imposti ulteriori semplici vincoli contestuali mediante l'utilizzo di altri sistemi formali.

[6] Compilatori siffatti sono talvolta chiamati "metacircolari". Il primo a descrivere un compilatore di questo tipo è l'italiano Corrado Böhm, anni prima della disponibilità del primo compilatore ad alto livello commerciale, quello per FORTRAN (rilasciato da IBM nel 1957). Cfr. Corrado Böhm, *Calculatrices digitales: Du déchiffrage de formules logico-mathématiques par la machine même dans la conception du programme*. Ann. di Mat. Pura ed Applicata, vol. 37(4), pp. 175-217, 1954.

Una traduzione fedele?

La descrizione dei linguaggi di programmazione ha preso a prestito molte tecniche dalla linguistica formale. Perché, allora, non interrogare le scienze del linguaggio anche sulla questione della traduzione? Per l'informatico, un compilatore è un traduttore fedele tra due linguaggi, un concetto del quale i teorici della traduzione diffidano. Senza entrare nello specialistico, sia concesso all'autore di queste righe di citare solo qualche frase di George Steiner, e da uno dei suoi testi meno specializzati[7].

> Una lingua riempie una cella nell'alveare delle percezioni e delle interpretazioni possibili. Articola una gerarchia di valori, di significati e di supposizioni che non corrisponde esattamente a quella di nessun altra lingua. (...) Parlando, creiamo mondi.[8]

Ogni lingua rappresenta un universo concettuale irriducibile ad ogni altra lingua. Non solo nessuna espressione linguistica complessa, ma perfino le singole parole, e addirittura i termini più semplici, fattuali, portano con sì connotazioni ed evocazioni che vengono definitivamente perse in traduzione:

> Il ricordo registrato di disponibilità, persino di abbondanza, nella parola *bread* (...) per molti aspetti si trova in contraddizione con le connotazioni di penuria e di fame del francese *pain*. Vi è un eccesso intraducibile di storia, di una mistica possessiva, nella *Heimat* tedesca.[9]

Ma se la traduzione non è in grado di mantenere questi "ricordi registrati", la loro disponibilità nella varietà delle lingue è ricchezza inestimabile. Il poliglottismo è una benedizione, un sapere di più, e saper far meglio, rispetto al monoglotta, perfino negli aspetti più connotati fisicamente dell'esistenza:

> L'eros delle persone multilingui, persino di un monoglotta dotato di mezzi verbali e di un buon orecchio, è diverso da quello delle persone linguisticamente negate o stonate. (...) Come doveva esser monotono l'amore in Paradiso.[10]

> Babele è stata il contrario di una maledizione. Il dono delle lingue non è una metafora vuota, è proprio un dono e una benedizione immensa.[11]

A dire il vero gli informatici non sono proprio "vili meccanici" e ben presto hanno osservato anch'essi quanto il linguaggio informi il modo di guardare al calcolo. Il teorema di equivalenza che abbiamo citato poc'anzi non è invalidato da queste considerazioni, ma in un certo senso esprime solo un aspetto della

---

[7] Citazioni scientificamente più argomentate, ma meno evocative, si sarebbe potute trarre, per esempio, da G. Steiner, *After Babel: Aspects of Language and Translation*, Oxford University Press, 1975.

[8] G. Steiner, *Errata: An Examined Life*, Weidenfeld and Nicolson, 1997. Trad. it., Garzanti, 1998, pag. 110.

[9] G. Steiner, *op. cit.*, pag. 108.

[10] G. Steiner, *op. cit.*, pag. 112.

[11] G. Steiner, *op. cit.*, pag. 110.

faccenda, quello della potenza "bruta": non ci troveremo mai a dover cambiare linguaggio di programmazione perché qualche calcolo vi è impossibile. Ma alcuni linguaggi sono più adatti di altri a determinati scopi. Alcuni sono più evocativi, altri più sintetici, altri ancora più astratti. Tra gli aforismi di Alan Perlis[12] troviamo un

> A good programming language is a conceptual universe for thinking about programming,

che ricorda in modo sorprendente il «Parlando, creiamo mondi» di Steiner. E ancora:

> A language that doesn't affect the way you think about programming, is not worth knowing.
>
> There will always be things we wish to say in our programs that in all known languages can only be said poorly.

Un altro pioniere che fa della notazione un cardine della propria riflessione è Kenneth Iverson[13]. Ideatore di un linguaggio di programmazione di una sinteticità ed espressività fulminanti, in specie per l'epoca, quando si utilizzavano schede perforate e non esistevano tastiere con simboli speciali[14], Iverson è tanto convinto che un linguaggio di programmazione influenzi il modo di pensare, da dedicare a questo argomento la propria lezione per il conferimento del premio Turing[15]. Prima di entrare nella parte più tecnica del suo lavoro (in cui mostra con esempi specifici come un attento utilizzo di APL permetta di evidenziare simmetrie e generalizzazioni), Iverson enuncia alcune caratteristiche di un buon linguaggio per l'espressione del calcolo. Lungi dall'essere neutro, un linguaggio *adatto al proprio scopo* è uno strumento raffinato, che *suggerisce* generalizzazioni, *evoca* analogie, *permette* dimostrazioni. È utile riprendere alcune delle caratteristiche enucleate da Iverson. Innanzitutto una buona notazione deve permettere di *esprimere con facilità i vari costrutti*, così come questi si presentano nei problemi:

> Una notazione deve esprimere convenientemente non solo le nozioni che derivano immediatamente da un problema, ma anche quelle che derivano da analisi successive, generalizzazioni, specializzazioni.

---

[12] Alan J. Perlis, famoso per le sue battute fulminanti, è stato un grande pioniere: primo direttore del primo dipartimento di informatica (quello di Carnegie-Mellon University), primo presidente della ACM, primo premio Turing (1966).

[13] Premio Turing nel 1979, «For his pioneering effort in programming languages and mathematical notation resulting in what the computing field now knows as APL, for his contributions to the implementation of interactive systems, to educational uses of APL, and to programming language theory and practice».

[14] A titolo d'esempio, la frase (~R∈R°.×R)/R←1↓⍳R è un programma in APL (il linguaggio di Iverson) che determina i numeri primi da 1 a R; un qualsiasi altro linguaggio, anche a noi contemporaneo, impiegherebbe diverse righe per lo stesso scopo.

[15] Kenneth Iverson, *Notation as a tool for thought*, Comm. of ACM, Vol. 33(8) pp. 444 - 465 (1980), dal quale sono tratte tutte le citazioni che seguono.

È in particolar modo rilevante il requisito di esprimere convenientemente anche le nozioni che derivino da elaborazioni *future* (e in qualche modo, dunque, non esplicitamente presenti alla mente). Il giudizio sulla notazione è un giudizio "storico", a posteriori, che rende ragione di quanto preveggente era stato un linguaggio (o di quanto è possibile adattare le sue costruzioni in modi che il suo progettista non aveva esplicitato).

In secondo luogo, una notazione deve essere *evocativa*:

> Una notazione è evocativa se la *forma* delle espressioni che si presentano in un insieme di problemi suggerisce espressioni collegate che trovano applicazione in altri problemi.

Scegliere la giusta forma sintattica di un'espressione (invece di un altro modo per esprimere lo stesso concetto), permette di vedere simmetrie ed analogie precedentemente nascoste, che suggeriscono *altre* espressioni che trovano applicazione in *altri* problemi. Un buon linguaggio è fecondo, genera alla vita nuovi concetti e proprietà. Davvero, «Parlando, creiamo mondi».

A questo punto Iverson passa ad alcune proprietà più operative della notazione del calcolo, che riguardano cosa possiamo (dobbiamo) fare con essa. Il primo di questi requisiti, intimamente connesso con la capacità di evocare, è un requisito di "astrazione"[16]. La notazione deve permettere di *subordinare dettagli*:

> La brevità facilita il ragionamento. La brevità è ottenuta mediante la subordinazione dei dettagli.

È questo un aspetto cruciale. La complessità può essere dominata solo suddividendo un problema in sottoproblemi, esprimendo ciascuno di questi per proprio conto e, quindi, descrivendo come ricomporre le soluzioni dei sottoproblemi per ottenere una soluzione del problema originale. Permettere la subordinazione dei dettagli vuol dire che un linguaggio di programmazione deve dare gli strumenti linguistici per tutto ciò. Non solo, dunque, come e cosa calcolare, ma costrutti linguistici per esprimere in modo gerarchico e strutturato un modo per risolvere i problemi. E tutto questo deve esser fatto non moltiplicando la verbosità, ma, al contrario, mirando ad una notazione *economica*:

> L'utilità di un linguaggio aumenta col numero di argomenti che esso può trattare, ma diminuisce con le dimensioni del vocabolario e la complessità delle regole sintattiche.

Infine, una notazione per l'espressione del calcolo deve permettere (con facilità) dimostrazioni formali:

---

[16] In informatica "astrazione" indica, di norma, l'operazione concettuale con la quale si passa da una descrizione di basso livello ad una descrizione dello stesso concetto ad un livello più alto, che subordina alcuni dettagli ed evidenzia alcune caratteristiche importanti sulle quali si vuole richiamare l'attenzione.

L’importanza delle dimostrazioni formali è chiara dal loro ruolo in matematica.

Verso una conclusione

L’informatica è la scienza del calcolo: studia i procedimenti effettivi di elaborazione (e memorizzazione, trasmissione, ecc.) dell’informazione. Condivide con altre scienze (prime tra tutte le matematica e la fisica) lo studio delle tecniche risolutive di determinati problemi (il *problem solving*). A questo studio porta almeno due contributi originali. Il primo è quello di *procedimento effettivo*: ricerchiamo una soluzione *calcolabile* (a differenza spesso del matematico...), cioè effettivamente realizzabile (con carta e penna o con un calcolatore) in tempo finito mediante manipolazione simbolica di un insieme finito di dati di ingresso. Secondo la tesi di Church, cui abbiamo accennato, ciò è equivalente a richiedere che la soluzione sia esprimibile mediante una macchina di Turing. Ma i problemi da risolvere sono difficili e complessi. Fare *problem solving* significa decomporre, ristrutturare, risolvere sottoproblemi e ricomporre, poi, le loro soluzioni. L’informatica – e questo è un suo secondo contributo originale – mette a disposizione *strumenti linguistici* progettati affinché ciò sia possibile e, per quanto possibile, semplice. Cioè evocativo, sintetico, economico; talvolta anche bello.

Come nelle lingue naturali, due linguaggi di programmazione non saranno mai equivalenti *relativamente a queste caratteristiche*. Ogni (*buon*!) linguaggio suggerisce alcune analogie, alcune generalizzazioni, alcune semplificazioni. E non altre. Passare da un linguaggio all’altro, da una notazione all’altra, evoca nuovi concetti e suggerisce nuove soluzioni. Conoscere più linguaggi di programmazione non è solo una riga in più su un *curriculum vitae*; la Babele dei linguaggi è una ricchezza e una benedizione. Parafrasando Steiner,

Come doveva esser monotono programmare in Paradiso!