Intermezzo: Esercizi di Stile

Ivano Salvo – Sapienza Università di Roma

Il presente scritto semi-serio vuole sensibilizzare sull'*estetica* dei programmi. Si possono scrivere *infiniti* programmi che risolvono lo stesso problema. Oltre alle importanti considerazioni relative all'*efficienza*, è importante fin d'ora concentrarsi sullo scrivere programmi *semplici*, *essenziali*, quindi facilmente *leggibili* e *modificabili*.

È importante anche adeguarsi al *galateo* del linguaggio di programmazione in uso: questo permette di *comunicare* efficacemente con la comunità che usa tale linguaggio e leggere facilmente il codice altrui. Tutto ciò diventa più importante via via che i problemi da risolvere saranno più complessi.

Infine, è bene tenere a mente tutte le possibilità che un linguaggio offre. Il secondo aneddoto, vuole appunto sensibilizzare sul fatto che diversi linguaggi o modi di pensare la programmazione, offrono diversi strumenti concettuali che possono aiutare a risolvere problemi che altrimenti risulterebbero molto difficili.

Esercizi di Stile

Raymond Queneau ci ha insegnato che anche il più insignificante avvenimento (come un tragitto in autobus) può essere descritto in (almeno) 100 modi diversi.

Un informatico teorico potrebbe dirvi sorridendo che ogni funzione può essere calcolata da *infiniti* programmi diversi. Questo è banalmente evidente, in quanto è possibile sempre aggiungere sequenze arbitrariamente lunghe di comandi inutili.

Lo *stile di programmazione* non è semplicemente un gioco fine a se stesso: a seconda di *come* scriviamo i programmi, specie quando la loro dimensione cresce, possiamo renderli più o meno *leggibili*, e quindi più o meno facili da *debuggare* (orribile inglesismo per "eliminare gli errori", o bug[1], ossia piccolo insetto, in italiano *baco*) più o meno facili da *mantenere*, cioè aggiornare le loro funzionalità. *Chiarezza* e soprattutto *essenzialità* o *semplicità* nella scrittura dei programmi sono qualità fondamentali che vi aiuteranno se vi cimenterete su problemi complessi. Ovviamente, vedere soluzioni semplici ai problemi è spesso molto difficile!

[1] il 9 settembre 1947 il tenente Grace Hopper ed il suo gruppo stavano cercando la causa del malfunzionamento di un computer Mark II quando, con stupore, si accorsero che una falena si era incastrata tra i circuiti. Dopo aver rimosso l'insetto (alle ore 15.45), il tenente incollò la falena rimossa sul registro del computer e annotò: "1545. Relay #70 Panel F (moth) in relay. First actual case of bug being found" [da Wikipedia].

```
void mergesort(int a[], int inf, int sup){
   int c = (inf + sup)/2;

   if (inf<c-1) mergesort(a, inf, c);
   if (c<sup-1) mergesort(a, c, sup);
   merge(a, inf, c, sup);
}
```

Figura 1: Ordinamento per fusione, Mergesort

Tipici "errori" dei principianti della programmazione consistono nel distinguere troppi casi particolari, oppure nell'usare troppe variabili, o soluzioni troppo arzigogolate. In ogni caso, cercare di vedere con gli occhi della mente uniformità e semplicità nelle soluzioni dei problemi, prima di curvarsi sulla tastiera a picchiettare sui tasti e a imprecare contro il compilatore, richiede un faticoso percorso di esperienza, più o meno lungo a seconda delle inclinazioni personali. Una sorta di *Zen e arte della programmazione*.

Oltre a questo, è in generale bene uniformarsi al *galateo* che ciascun linguaggio di programmazione ha distillato nel tempo, attraverso l'uso intensivo da parte di programmatori esperti. Il codice che trovate nei buoni libri segue regole non scritte di indentazione (cioè dove cominciano le righe del programma), regole nella scelta dei nomi di variabili e funzioni (tipo l'uso di `n` ed `m` per i parametri che rappresentano le lunghezze dei vettori, `i` e `j` per gli indici etc.), scelta della corretta struttura di controllo rispetto al tipo di ciclo che si deve eseguire, e così via. Questo vi aiuterà a capire al volo un programma e a comunicare meglio con altri programmatori.

In questo breve scritto semi-serio, vi racconto due brevi novelle. Nella prima, una personalissima rivisitazione della mia vita da programmatore, suggerita da un'ora in treno a preparare una lezione sulla funzione di fusione ordinata ricorsiva di due vettori ordinati. Spero che il risultato sia un viaggio istruttivo nei diversi stili con cui si può scrivere un programma, a seconda dei propri "retaggi linguistici". Osservate qua e là come la conoscenza di diversi strumenti concettuali, a rigore appartenenti a linguaggi diversi, possano comunque influenzare il codice scritto in C.

Nella seconda, un aneddoto (che molti giurano essere vero) raccontato in forma di fiaba su uno dei grandi padri fondatori dell'Informatica, la cui morale è che, come nella vita, il linguaggio (e gli strumenti da esso offerti), *influenza* la concettualizzazione dei problemi.

1 Un Week-end al SortPub

Il problema dovrebbe essere ben noto: dati due vettori ordinati, produrre un terzo vettore ordinato che contiene gli elementi dei primi due. L'utilizzo principe di questa procedura è come sottoprocedura all'interno dell'algoritmo di ordinamento *MergeSort* (John von Neumann, 1945), di cui ricordiamo al lettore una possibile implementazione in Fig. 1. Un programma che coniuga semplicità e prestazioni, un autentico messaggio promozionale per il *divide et impera*.

Purtroppo, sui vettori, le sue prestazioni (comunque ottime da un punto di vista asintotico, e peraltro indipendenti dall'ordine iniziale del vettore), non sempre reggono il passo con altri algoritmi un po' più sofisticati come *QuickSort* o *HeapSort*, a causa proprio della fusione dei vettori che necessita, inevitabilmente, di un vettore ausiliario e relativi travasi.

```
void mergePascal(int a[], int inf, int medio, int sup){
  /* PREC: forall i. inf<=i<med. a[i]<=a[i+1] &
   *       forall i. med<=i<sup. a[i]<=a[i+1]
   * POST: forall i. inf<=i<sup. a[i]<=a[i+1]
   */
     int i=inf;
     int j=medio;
     int k=0;
     int c[sup-inf];

     while (i < medio && j < sup) {
         if (a[i] <= a[j]) {
            c[k] = a[i];
            i = i+1;
          } else {
              c[k] = a[j];
              j = j+1;
         } /* endif */
         k = k+1;
     } /* endwhile */
     if (i<medio)
       for (; i<medio; i++) { c[k]=a[i]; k=k+1; }
     if (j<sup)
       for (; j<sup; j++) { c[k]=a[j]; k=k+1; }
     copia(c, k, a, inf, sup);
}
```

Figura 2: Fusione ordinata di due vettori, versione madrelingua Pascal

Tuttavia la sua semplicità lo rende alquanto versatile. Vedremo che su altre strutture dati lineari (come le liste) è altrettanto semplice da scrivere ed efficiente, men-

tre gli altri algoritmi si fanno più oscuri nell'implementazione e meno performanti. Infine, *MergeSort* è più semplicemente ed efficacemente parallelizzabile.

Venerdì Sera al SortPub

Il venerdì sera, dopo la settimana di lavoro, è uso vedersi al *SortPub*. Dagli albori dell'estate, quando le stagioni del rugby e del football volgono al termine, è difficile lasciarsi alle spalle il lavoro, e la discussione cade inevitabilmente su quale sia il migliore algoritmo di ordinamento. La scena non cambia molto rispetto alla primavera e all'inverno, e le opposte tifoserie si insulatano allegramente tra un boccale di birra e l'altro, venendo occasionalmente alle mani. Ma alla fine, ci si beve su.

Niklaus, un ragazzotto di origine svizzera, da un po' di tempo ha spostato l'asse della discussione al *modo* in cui si scrivono gli algoritmi, e da un po' di tempo asserisce con vigore la superiorità del PASCAL come linguaggio di programmazione, ma per comunicare con gli altri decide di scrivere la funzione *merge* in C, nello stile però del PASCAL.

Da buon padre di famiglia qual è, dopo aver diligentemente scritto le precondizioni e post-condizioni, e aver inizializzato una variabile alla volta, Niklaus scrive il programma di Fig. 2 e lo sventaglia orgoglioso in giro per il *SortPub*.

Innanzitutto osserviamo che ha fatto la fusione di due pezzi dello stesso vettore dentro un vettore ausiliario. Del resto, è in questa forma che a lui serve nell'algoritmo *MergeSort* (vedi Fig. 1). E gli sarebbe difficile (in un linguaggio di famiglia ALGOL), all'interno di *MergeSort* usare una funzione *merge* più generale, che fonde due vettori ordinati qualsiasi, se a lui serve fondere due porzioni dello stesso vettore. Non avendo ancora padronanza del linguaggio riporta questo limite nel programma C che scrive.

Oltre a questo, scrive ben 6 caratteri per incrementare una variabile contatore (spesso 9 o più a causa della generosità con cui è aduso distribuire spazi tra un operatore infisso e i suoi operandi) laddove un programmatore C ne userebbe al più 4 con l'incremento postfisso.

Accede agli elementi di un vettore *solo* attraverso gli indici, perché con soddisfazione racconta ormai da molti venerdì agli amici programmatori C che il suo linguaggio preferito gli nasconde dettagli "pornografici" come gli indirizzi di memoria. Oltre il litro di birra però, i Veri Programmatori C cominciano a far valere la forza dei puntatori! E non solo.

Sabato Pomeriggio

Dennis, un Vero Programmatore C, ha un demone che lo domina: il piacere di manipolare esplicitamente la memoria del suo inseparabile portatile: lo sa fare, va ammesso, con leggiadra maestria, grazie anche ad un uso spregiudicato dell'aritmetica dei puntatori. I vettori, per lui, sono una inutile sovrastruttura. Accetta, a malincuore, di dichiarare `int a[1000];` solo perché si diteggia più rapidamente di

```
void merge(int a[], int m, int b[], int n, int c[]){
    int i=0;
    int j=0;
    int k=0;

    while (i<m && j<n)
         if (a[i]<=b[j]) c[k++]=a[i++];
                     else c[k++]=b[j++];
    while (i<m) c[k++]=a[i++];
    while (j<n) c[k++]=b[j++];
}
```

Figura 3: Fusione ordinata di due vettori, versione Vero Programmatore C

`int* a=calloc(1000, sizeof(int));`. E se, testando un programma, compare a video la temuta scritta `Segmentation Fault` si tratta solo di una conferma che si sta percorrendo la strada giusta.

Ha ancora il cerchio alla testa a causa del venerdì sera al *SortPub*. Ha bevuto un po' troppo e, guardandosi allo specchio, scopre un'ecchimosi al labbro. Non si rammenta cosa possa essere, ma uscendo di casa, scopre nella tasca del giubbetto di pelle un foglietto scritto e autografato da Niklaus, che contiene la `mergePascal` e allora si ricorda quale fosse la discussione che ha portato alla zuffa. Guarda con un po' di tenerezza il foglietto, e quel codice così prolisso e si rammarica di dover andare a prendere la ragazza per andare al parco; infatti non vede l'ora di mettersi al lavoro per accettare la sfida e scrivere del codice C degno di tal nome.

Il pomeriggio, senza strafare, e senza usare l'artimetica dei puntatori che il suo amico Niklaus, madrelingua Pascal, non capirebbe, scrive la funzione in Fig. 3. Innanzitutto, scrive una funzione più generale, che fonde due vettori qualsiasi. Tanto lui, nella *MergeSort* la potrà comunque chiamare con `merge(&a[inf], c-inf, &a[c], sup-inf, int c[]);` e poi chiamare la funzione `copia`.

Subito dopo, compatta tutti gli incrementi degli indici dentro le assegnazioni tra elementi dell'array. In fondo vanno incrementati proprio quelli coinvolti nelle assegnazioni. Immagina già l'obiezione dell'amico-nemico: "ma è un caso fortunato". Lui pensa con il suo spirito pratico tipico dei Veri Programmatori C: "sarà pure fortuna, ma di solito è così...".

C'è ancora qualcosa che lo disturba. I due `if` finali sono evidentemente inutili. Sono sussunti dalle guardie dei `for`. Un ultimo sorriso interiore mentre termina di scrivere: "programmare in Pascal atrofizza il cervello...".

```
void mergeRec(int a[], int i, int m, int b[],
                       int j, int n, int c[], int k){
     if (i==m) {
        copiaRec(b, j, n, c, k);
        return;
     } else if (j==n) {
        copiaRec(a, i, m, c, k);
        return;
     } else if (a[i]<=b[j]) {
        c[k]=a[i];
        mergeRec(a,i+1,m,b,j,n,c,k+1);
     } else {
        c[k]=b[j];
        mergeRec(a,i,m,b,j+1,n,c,k+1);
     }
}
```

Figura 4: Merge ricorsiva, versione madrelingua ML

Sabato Sera, Ritorno al SortPub

Per fortuna, la fidanzata quel sabato sera è impegnata con una festa di addio al nubilato e così Dennis può recarsi al *SortPub* a mostrare il piccolo capolavoro composto nel pomeriggio. Al *SortPub* però, Niklaus è già immerso in una fervente discussione con un signore straniero, tale Xavier.

Xavier non conosce l'iterazione, avendo studiato programmazione all'università di Paris VII, dove si insegna da subito il CAML LIGHT, il dialetto di ML sviluppato in Francia a fini educativi. Per lui, i programmi sono sequenze di *equazioni ricorsive* e la loro esecuzione evoca il piacevole quanto lontano ricordo delle espressioni aritmetiche di cui calcolava il valore riducendo un passo alla volta una qualche sottoespressione non ancora *ridotta a forma normale*, cercando alla fine il sorriso di approvazione della maestra. Si trova, per la verità, anche a disagio coi vettori. Gli indici, infatti, sono per lui un dettaglio tecnico che non parla dell'intima struttura algebrica del tipo di dato in uso: "oui, d'accord, j'ai un vector, mais qu'est-ce que c'est?" sta infatti sbraitando a Niklaus, quando Dennis entra al *SortPub*. Vuole manifestare il suo disagio, in quanto non lo associa a nessuna struttura algebrica a lui nota, una sequenza, un insieme, o quant'altro.

Dovendo risolvere il problema della fusione di due vettori, si altera un po', esclamando a gran voce "il faut écrire la specification functionelle!" e non trova di meglio che *specificarla* con equazioni ricorsive sull'insieme delle *sequenze ordinate* di interi, e visto che c'è da loro anche un nome: $SeqOrd[\mathbb{Z}]$. In pochi secondi, scrive tre equazioni:

$$\mathsf{merge}(a_1 \cdot s_1, a_2 \cdot s_2) = \begin{cases} a_1 \cdot \mathsf{merge}(s_1, a_2 \cdot s_2) & \text{se } a_1 \leq a_2 \\ a_2 \cdot \mathsf{merge}(a_1 \cdot s_1, s_2) & \text{se } a_2 < a_1 \end{cases}$$
$$\mathsf{merge}(\langle\,\rangle, s) = \mathsf{merge}(s, \langle\,\rangle) = s$$

In quel momento Niklaus nota Dennis e gli dice: "Vieni a vedere che sta combinando il mio amico francese!". Lo aiutano un po' con il C e il risultato è la funzione in Fig. 4. Sostanzialmente si tratta della funzione `mergePascal`, trasformata in ricorsiva, usando dei parametri ausiliari per simulare gli indici del programma iterativo. È purtroppo *necessario* usare degli indici per trattare l'array, la testa della sequenza è rappresentata dall'indice corrente, mentre la sequenza vuota si riconosce quando l'indice corrente ha raggiunto la lunghezza del vettore. Inoltre i vettori non si possono restituire come risultato, e quindi occorre un altro parametro vettore su cui accumulare il risultato e soprattutto è necessario progettare un altra funzione per copiare le code del vettore.

Xavier prende i due foglietti, gli allontana dagli occhi per vederli entrambi insieme contemporaneamente, vede come la bellezza delle sue tre equazioni ricorsive si sia corrotta nel programma C, sbuffa pesantemente bofonchiando in francese. Dennis e Niklaus distinguono solo la parola "merde".

La Sfida dei Sottobicchieri

Dennis non ama la ricorsione. Guarda con diffidenza i programmi ricorsivi dal giorno in cui ha scoperto con disgusto che una chiamata di funzione necessita della corrispondente allocazione di un activaction record sulla pila di sistema. La usa solo quando è messo spalle al muro su problemi molto ardui da risolvere altrimenti.

Di recente però, qualcuno gli ha fatto notare che questa operazione sulle moderne architetture costa poco più di un ciclo macchina. Non solo, ma un tipo un po' anziano che frequenta il *SortPub*, che tutti chiamano "Professore" per il tono delle sue affermazioni, addirittura asserisce che alcuni programmi ricorsivi, tipo appunto *MergeSort*, hanno una maggiore *località*, cioè tendono a trattare dati che sono memorizzati in celle contigue di memoria e quindi aiutano il compilatore a generare codice efficiente che alloca questi dati in memorie veloci (*cache* o *registri*).

Dennis, prima ha voluto verificare in prima persona, armeggiando con la sua inseparabile compagna, l'amata libreria `time.h`), poi, amando le sfide, ha deciso per impratichirsi di riprogrammare la funzione `merge` in versione ricorsiva. Quello che è riuscito a capire delle osservazioni di Xavier però lo hanno stimolato a fare meglio.

E allora, sul sottobicchiere del boccale di birra (si, questa è la sua fissa, sfidare gli astanti a scrivere un certo programma in modo che trovi posto sul sottobicchiere dei boccali) scarabocchia la funzione in Fig. 5.

Grazie alla discussione sul significato astratto delle sequenze, riesce a fare meno degli indici tra i parametri. Nella sua versione infatti i parametri interi non sono

```
void mergeRecVPC(int a[], int ra, int b[], int rb, int c[]){
     if (ra==0) {copiaRec(b,rb,c); return;}
        else if (rb==0) {copiaRec(a,ra,c); return;}
        else if (*a<=*b) { *c++=*a++; ra--;}
        else { *c++=*b++; rb--;}
     mergeRecVPC(a,ra,b,rb,c);
}
```

Figura 5: Merge Ricorsiva da Vero Programmatore C - I

```
void mergeRecVPC(int a[], int ra, int b[], int rb, int c[]){
     if (ra==0 && rb==0) return;
     if (rb==0 || *a<=*b) { *c++=*a++; ra--;}
        else if (ra==0 || *b<=*a) { *c++=*b++; rb--;}
     mergeRecVPC(a,ra,b,rb,c);
}
```

Figura 6: Merge Ricorsiva, da Vero Programmatore C - II

tanto le lunghezze dei vettori, quanto il numero degli elementi *che mancano* alla fine del vettore, e infatti chiama i parametri `ra` e `rb`, cioè resto di `a` e resto di `b`.

Niklaus prova un po' di invidia per l'effetto che il risultato di Dennis provoca sul suo amico Xavier, e si sente un po' scavalcato. Xavier, infatti, lisciandosi la barba si calma un po': "Mais, bon, le Se, c'est pas mal!".

Domenica Mattina

La mattina, dopo la consueta generosa "dose" di caffè, Dennis sul terrazzo fuma la prima sigaretta pensando alla bella serata al *SortPub*, ai colori caldi dell'autunno entrante e alla stagione del rugby ormai alle porte.

Dennis, a un tratto però è preso da un momento di sconforto e si lascia andare a una sonora imprecazione. Allora rientra in casa, apre con impazienza il suo inseparabile portatile e si mette di nuovo febbrilmente al lavoro: si è reso conto che la funzione `copiaRec` è inutile! In fondo, `mergeRec` di suo, fa già molto di più. Ecco il suo super-compresso programma in Fig. 6.

Questo programma, oltre all'aritmetica dei puntatori, specula per esempio sul fatto che se la prima espressione di una condizione `||` valuta a vero, non si va a calcolare la seconda. Infatti, quando $ra = 0$ oppure $rb = 0$, le condizioni $*a \leq *b$ o $*b < *a$ non sarebbero ben definite.

Si rammarica un po' perché la fusione di vettori venerdì prossimo al *SortPub* sarà già fuori moda e non potrà farsi bello col suo programma. Ora però non gli interessa più: contemplare il suo piccolo capolavoro lo appaga completamente!

2 Una favola informatica a lieto fine

Per completare questo scritto sulle sottili relazioni tra "forma" e "sostanza", il seguente aneddoto in forma di fiaba vorrebbe farvi ancora una volta riflettere sull'importanza di usare i corretti costrutti linguistici. Insomma, per dirla con Nanni Moretti, ma in positivo, "chi parla bene, pensa bene".

C'era Una Volta...

... negli anni '50 del secolo scorso (ahimè), un giovane laureato in lettere ad Oxford, C. A. R. Hoare, oggi meglio noto come Tony Hoare[2]. Assecondando il suo spirito curioso e aperto alle novità, di ritorno da Mosca, dove si era recato a imparare il russo, trovò lavoro in un piccola industria che si occupava degli allora modernissimi elaboratori elettronici. Il titolare, un rispettabile ingegnere, chiese un giorno a Tony di implementare un algoritmo di ordinamento.

Capisco, miei giovani lettori, che l'inizio di questa favola faccia rabbrividire: ma nonostante nomini nelle prime righe le parole *ingegnere*, *algoritmo* e *ordinamento* una vicina all'altra, non si tratta di una favola dell'orrore.

All'epoca i calcolatori venivano programmati prevalentemente in *linguaggio macchina* e i programmi venivano scritti su schede perforate (nell'epoca del Web2.0, tutto ciò sembra preistoria, ma questo era ancora in uso fino ai primi anni '80 in numerosi centri di calcolo universitari). Nei casi più sfortunati, era necessario modificare *fisicamente* la struttura della macchina.

Nonostante ciò, non mancavano i pionieri visionari. Nei primi anni '50, per esempio, John Backus aveva progettato e implementato il primo linguaggio di programmazione ad *alto livello*, il FORTRAN (*FORmula TRANslator*), che però non aveva l'*allocazione dinamica* dei record di attivazione delle procedure, e perciò non ammetteva la ricorsione. Per la verità, non aveva nemmeno delle vere e proprie procedure con un proprio *stato locale*.

All'epoca dei fatti, l'algoritmo di ordinamento più usato era lo *ShellSort*, oggi praticamente dimenticato e uscito dai programmi di studio universitari. Si tratta di una sorta di *BubbleSort* che però fa scambi a distanze variabili decrescenti. Alle brutte, l'ultima passata, che fa gli scambi a distanza 1, sistemerà le cose. Asintoticamente è un algoritmo quadratico e non c'è un chiaro motivo per cui debba funzionare bene (la sua analisi del caso medio è infernale, e tutto sommato non ha senso se non ricorrendo a una possibile distribuzione di probabilità sugli input). Sono anche stati fatti studi approfonditi (spesso puramente sperimentali) su quali siano le sequenze di distanze migliori da scegliere con cui fare gli scambi. L'intuizione alle spalle di *ShellSort* è che la strada che deve percorrere un elemento per raggiungere il posto

[2] credo la 'A' stia per Antony: come forse sapete, è tipico del mondo anglosassone avere un secondo nome. Ad esempio, Homer J. Simpson.

giusto nel vettore ordinato è (nel caso medio) molto maggiore di 1. Non a caso, *ShellSort* ha delle prestazioni spettacolari per un ordinamento quadratico.

Il giovane neo-informatico Tony Hoare era un po' turbato dalla richiesta del suo principale di implementare *ShellSort*. Un po' non capiva a fondo questo algoritmo di ordinamento, un po' gli sembrava che si sarebbe potuto fare meglio, e si mise a giocare a ordinare dei fogli di carta sulla sua scrivania. Il giorno seguente, andò dal principale con l'aria trionfante di entusiasmo, tipico dei giovani convinti di aver raggiunto una soleggiata radura di verità nell'oscuro bosco dell'ignoranza.

Tony descrisse al principale la sua idea, aiutandosi ancora con i fogli di carta, stavolta quelli sulla scrivania del capo. "Si tratta", cominciò Tony "di scegliere un elemento e poi percorrere il vettore da entrambi i lati, finchè a sinistra non si trova un elemento maggiore dell'elemento scelto, e a destra non si trova un elemento minore: si scambiano e poi si continua...".

Il capo, un po' infastidito dalla presunzione del giovane collaboratore, e decisamente scosso dal vedere i propri fogli perdere la loro geometrica sistemazione sulla sua scrivania, ascoltava con un misto di sospetto e scocciata disattenzione. Inoltre, da buon ingegnere navigato, trovava rassicuranti le procedure standardizzate, mentre era messo a disagio dalla fantasia e l'estro. Non era però una strega cattiva, e decise comunque di dare una possibilità al suo sottoposto: "ti do una settimana di tempo per sperimentare la tua idea. Se mi dimostri che funziona meglio, la useremo".

Tony si mise alacremente al lavoro, ma si scontrava contro una difficoltà apparentemente insormontabile: dopo aver correttamente partizionato gli elementi tra i maggiori e i minori dell'elemento scelto, occorreva riapplicare l'idea alle due partizioni ottenute. La cosa gli riusciva facilmente con i suoi fogli sulla scrivania, ma non riusciva a codificarla nel programma. E si trattava, badate bene, non di un neolaureato qualsiasi, ma del nostro eroe, Tony Hoare, un futuro Turing Award. Alla fine della settimana, scoraggiato, ne concluse che la sua idea non era implementabile, e cominciò a implementare *ShellSort*.

L'Alba della Ricorsione

In quegli stessi anni, un altro pioniere visionario, John McCarthy conduceva i primi studi sull'Intelligenza Artificiale, ma era profondamente insoddisfatto dei mezzi a sua disposizione per programmare i suoi prototipi: gli sembrava inconcepibile dover preoccuparsi di gestire i registri di una stupida macchina nella descrizione di una procedura per distinguere un verbo da un complemento.

Trovandosi a Princeton, decise di seguire un seminario di un eminente logico del tempo, Alonzo Church, che sosteneva di aver trovato un eccellente formalismo per rendere rigorosa la nozione di funzione e, più importante ai suoi occhi, di "funzione calcolabile". La visione del λ-calcolo lo folgorò. Tre sole regole sintattiche e una sola regola di computazione "quasi" algebrica, potevano rappresentare ogni computazione. La cosa più importante era che queste computazioni, però, non erano

mirate a comandare la sequenza di azioni di una macchina, ma parlavano di oggetti matematici chiari (in particolare *funzioni ricorsive*) che si prestavano egregiamente a risolvere problemi nell'elaborazione del linguaggio naturale.

L'entusiasmo fu tale che, prima di capirne a fondo le finezze, John McCarthy dette la sua personalissima interpretazione al λ-calcolo, implementando il secondo linguaggio di programmazione, il LisP, acronimo per *LISt Processing*. In LisP, coerentemente con il λ-calcolo, si possono scrivere *solo* equazioni ricorsive e i programmi hanno la *stessa struttura sintattica* dei dati su cui operano: entrambi sono liste.

Il LisP, per la verità, è rimasto a lungo uno slang privato parlato da scienziati che si occupavano di Intelligenza Artificiale. Per programmare calcoli scientifici o di Ricerca Operativa (spesso basati sulla manipolazione di vettori, immaginate un solutore di sistemi lineari) il ForTran sembrava perfettamente adeguato e soprattutto, produceva codice macchina estremamente più efficiente. Per le nascenti applicazioni gestionali invece, era stato implementato un linguaggio noioso e prolisso, il CoBOL (*COmmon Business Oriented Language*), che piaceva poco ai matematici e agli ingegneri, ma metteva a loro agio i ragionieri (per farvi un un'idea, quello che in C si scrive con `a-=i;` in CoBOL si scrive `SUBTRACT I FROM A.`).

Accadde una Notte

Non ci siamo dimenticati del nostro eroe, Tony. Forse fu proprio a causa del ghigno soddisfatto di superiorità con cui il suo capo aveva salutato il suo fallimento nell'impresa di implementare un nuovo algoritmo di ordinamento, che Tony aveva nel frattempo lasciato il mondo dell'industria per abbracciare la carriera accademica. Il suo talento cominciava a emergere in vari campi a livello internazionale, dalle logiche per stabilire la correttezza dei programmi, alla programmazione concorrente, agli algoritmi.

Dev'essere stato in un corridoio un po' buio dell'Università di Oxford, che accadde. Un collega di Tony gli disse: "ehi, Tony, hai visto l'ultimo linguaggio che è uscito? È una vera forza. Pensa che una procedura può chiamare se stessa! Guarda questo programmino per il fattoriale, quant'è carino! Solo due righe di codice".

Parlava dell'AlgoL (*ALGOrithmic Language*). Ormai gli anni '60 erano alle porte, e benchè AlgoL sia un linguaggio imperativo (quindi basato essenzialmente sull'assegnazione come il C), i suoi progettisti avevano tenuto nel giusto conto gli insegnamenti del LisP: funzioni ricorsive e stato locale delle procedure apparivano per la prima volta in un linguaggio imperativo.

Il collega di Tony rimase un po' perplesso: gli si era rivolto in modo amichevole ed entusiasta ed ora lo vedeva pensoso con lo sguardo rivolto al pavimento. Tony alzò gli occhi guardando nel vuoto come per fissare un punto nell'orizzonte, e non seppe urlare altro che: "Allora... si pu-ò fa-re!".

Si riferiva, ovviamente, al suo algoritmo di ordinamento, cioè *QuickSort*.

Hanno cortesemente partecipato (a loro insaputa)

Tony Hoare: partecipò poi allo sviluppo e all'implementazione dell'ALGOL '60, che da sempre è il linguaggio a cui si ispira lo pseudocodice dei libri di algoritmi. Proprio relativo all'ALGOL è il più celebre aforisma a lui attribuito: "Tra i linguaggi di Programmazione, l'ALGOL rappresenta un progresso notevole rispetto ai suoi successori".

Uno dei principali sistemi logici che formalizza l'idea di *asserzione logica*, *invariante* etc., si chiama *Logica di Hoare* (o Triple di Hoare). Molto importanti anche i suoi studi nella Teoria della Concorrenza, cioè lo studio di proprietà di programmi che "collaborano" concorrentemente a uno stesso scopo, accedendo alle stesse risorse o strutture dati. Tutto lo sviluppo dei Sistemi Operativi, ad esempio, si fonda su questo concetto.

Frequenta tuttora gli avvenimenti internazionali di Informatica Teorica.

John Backus: Oltre al FORTRAN, John Backus ha partecipato anche al progetto dell'ALGOL. Il suo nome è tuttavia legato soprattutto alla cosìdetta *Backus-Naur–Form* (BNF)[3], un modo per descrivere la sintassi di un Linguaggio di Programmazione (ma non solo).

Celeberrimo anche un suo lavoro *"Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs"* del 1978: ravvedutosi sulla via del Turing Award, dopo aver progettato i più importanti linguaggi imperativi, nella dissertazione in occasione del ricevimento del più prestigioso riconoscimento scientifico in informatica, riaccese l'interesse per il λ-calcolo e i linguaggi funzionali, osservando la facilità con cui si possono analizzare le loro proprietà con ragionamenti di carattere algebrico ed equazionale.

John McCarthy: è venuto a mancare nel 2011, pochi giorni dopo Steve Jobs, e pochi giorni prima di Dennis Ritchie (uno dei progettisti del C). È riconosciuto come il padre fondatore dell'Intelligenza Artificiale. Già che era di strada, è stato il primo a implementare il λ-calcolo nel LISP.

Per onor di cronaca, va ricordato il celebre errore di implementazione di McCarthy della regola di computazione del λ-calcolo: il LISP usa il cosìdetto *scoping dinamico* per dare un senso alle variabili libere di una funzione (è come se le variabili libere di una funzione C riferissero a eventuali variabili con lo stesso nome del *chiamante* – che non è necessariamente sempre lo stesso – piuttosto che, come accade, a variabili globali): un errore gravido di ricchi sviluppi futuri! Un po' come Cristoforo Colombo, che parte alla volta delle Indie e scopre le Americhe. Infatti,

[3] È in BNF, per esempio la definizione del linguaggio TinyC data nella dispensa D5, Sezione 2.

oggi lo scoping dinamico è ritenuto il "modo corretto" di decidere quale metodo mettere in esecuzione in un linguaggio Orientato agli Oggetti[4].

Nella prima storiellina, i personaggi sono di pura fantasia. Tuttavia i nomi, alludono a importanti informatici. Niklaus allude a **Niklaus Wirth**, progettista del PASCAL e Turing award nel 1984. Il suo libro più famoso (*Algorithms + Data Structures = Programs*) è un grande classico nella biblioteca ideale di un informatico ed è costruito attorno al parallelo tra strutture di controllo e strutture dati: array e iterazione, strutture dati induttive (alberi, liste, sequenze) e ricorsione, etc., mentre la metodologia di sviluppo top-down dei programmi (scomposizione in sottoproblemi) viene fatta tradizionalmente risalire a un suo celebre scritto *Program Development by Stepwise Refinement*.

Dennis allude a uno dei progettisti del Linguaggio C, **Dennis Ritchie**, premio Turing nel 1983. Anche per Dennis Ritchie, un po' come il LISP per John McCarthy, il C è stato soprattutto uno strumento per scrivere il codice di Unix, di cui fu uno dei progettisti e implementatori principali. Tutti i Sistemi Operativi moderni degni di tal nome, derivano più o meno direttamente da Unix, e in ogni caso, ne hanno incorporato i concetti principali.

Xavier allude a **Xavier Leroy**, uno dei progettisti di CAML e dei suoi dialetti (il più importante dei quali sicuramente è OCAML, la versione orientata agli oggetti). È l'intruso del gruppo, non essendo un premio Turing (il progettista di ML, **Robin Milner**, viceversa lo ha vinto nel 1991). CAML è sicuramente uno dei più maturi linguaggi funzionali, e fa parte di un più ampio progetto di ricerca francese, che va dall'educazione allo sviluppo di *programmi certificati*, cioè dimostrati corretti (prevalentemente attraverso un tool che permette dimostrazioni semi-automatiche di proprietà di programmi che si chiama CoQ).

[4]Per la verità, linguaggi diversi fanno scelte diverse. Per JAVA, nella tradizione di Smalltalk, il binding dinamico è lo standard. In C++, a causa di motivi di efficienza, è il programmatore che sceglie il binding dinamico dichiarando un metodo `virtual`.