

Storia dei Linguaggi di Programmazione

In principio non c'erano sistemi operativi, non c'erano compilatori, non c'erano linguaggi di programmazione, e tutto l'universo era solo una infinita stringa di numeri...

Storia dei Linguaggi di Programmazione

- C'è voluto molto tempo perché i linguaggi di programmazione assumessero la forma che hanno oggi, e il concetto stesso di *linguaggio di programmazione* si è evoluto nel tempo insieme con le architetture di computer.
- Nel corso degli anni, alcuni linguaggi hanno introdotto nuovi concetti che si sono rivelati fondamentali e sono stati adottati in (almeno un sottoinsieme di) tutti i linguaggi successivi, formando un nucleo consolidato di idee e pratiche di buona programmazione.
- Nuovi linguaggi sono nati, evoluti e scomparsi in un vero e proprio processo di selezione naturale, fino ad arrivare al panorama che conosciamo oggi (ovviamente ancora in evoluzione).

Storia dei Linguaggi di Programmazione

- In poco più di 60 anni, sono stati concepiti e implementati centinaia di linguaggi di programmazione ad alto livello. Se poi consideriamo anche i vari dialetti di un singolo linguaggio (il solo Algol ne conta una settantina circa), il numero totale è probabilmente superiore al migliaio.
- E tuttavia, sono poco più che una decina i linguaggi che fino ad oggi hanno avuto un impatto davvero significativo sullo sviluppo e l'evoluzione del concetto di *Linguaggio di Programmazione*.
- Sono questi i linguaggi che prenderemo in considerazione, dato che è da essi che hanno avuto origine quasi tutti i concetti fondamentali poi utilizzati (in molti casi introducendo qualche miglioria) nei linguaggi sviluppati negli anni successivi. 3

Storia dei Linguaggi di Programmazione

- Noi oggi diamo per scontato il concetto di “*Linguaggio di Programmazione*”, e pensiamo principalmente a come un linguaggio possa essere migliorato in termini di espressività, semplicità d’uso ed efficienza di esecuzione.
- Dimentichiamo così che la realizzazione stessa di un linguaggio di programmazione è un processo estremamente complicato, che si fonda su un insieme di nozioni teorico-pratiche molto complesse che hanno richiesto anni di sforzi intellettuali per emergere.
- Addirittura, negli anni ‘40, John Von Neumann dichiarava di non vedere l’utilità nel cercare di sviluppare forme di programmazione diverse dall’uso del linguaggio macchina...

Storia dei Linguaggi di Programmazione

- Naturalmente, come per le architetture, è impossibile stabilire una data precisa con cui incomincia la storia dei linguaggi di programmazione, e alcuni episodi si qualificano piuttosto come **Preistoria** dei linguaggi di programmazione.
- **Charles Babbage** derivò l'idea di pilotare l'Analytical Engine mediante schede perforate dal **telaio di Jaquard**, e intorno al 1843 **Ada Lovelace** elaborò una specifica precisa per calcolare i numeri di Bernoulli con l'Analytical Engine.
- Esagerando un po', alcuni storici amano considerare questa specifica **il primo programma per computer**, e di conseguenza **Ada Lovelace la prima programmatrice della storia**.

Storia dei Linguaggi di Programmazione

- I computer analogici degli anni ‘30 potevano calcolare funzioni diverse, ma la loro “programmazione” consisteva nel riconfigurare i circuiti elettrici di cui erano composti.
- Del resto ciò era vero anche per l'**ENIAC** degli anni ‘40: per risolvere un certo problema occorreva “programmare” l’ENIAC riconfigurando a mano migliaia di switch e collegamenti elettrici.
- Lo **Z1** di **Konrad Zuse** del 1936/38 leggeva semplici istruzioni aritmetiche e di I/O da un nastro perforato. Nel 1943/45 Zuse concepisce per lo **Z3** il **Plankalkül**, una sorta di linguaggio ad alto livello che permette di specificare anche salti condizionati e iterazioni, ma che non verrà mai implementato.

Storia dei Linguaggi di Programmazione

- Nel **Mark I** di **Howard Haiken**, completato nel 1943, le istruzioni macchina (operazioni aritmetiche + input e output) erano codificate su nastro perforato, e i loop erano ottenuti incollando letteralmente insieme la fine e l'inizio del nastro.
- Tra il ‘43 e il ‘45 **Goldstine** e **Von Neumann** sviluppano l'idea di rappresentare un programma mediante **Flow Diagrams**, una notazione grafica che diverrà poi universalmente nota come **Flowchart**.
- Nei Flow Diagrams incomincia a manifestarsi il punto di transizione dal concetto matematico di “uguaglianza” a quello informatico di “**assegnamento**”.

Storia dei Linguaggi di Programmazione

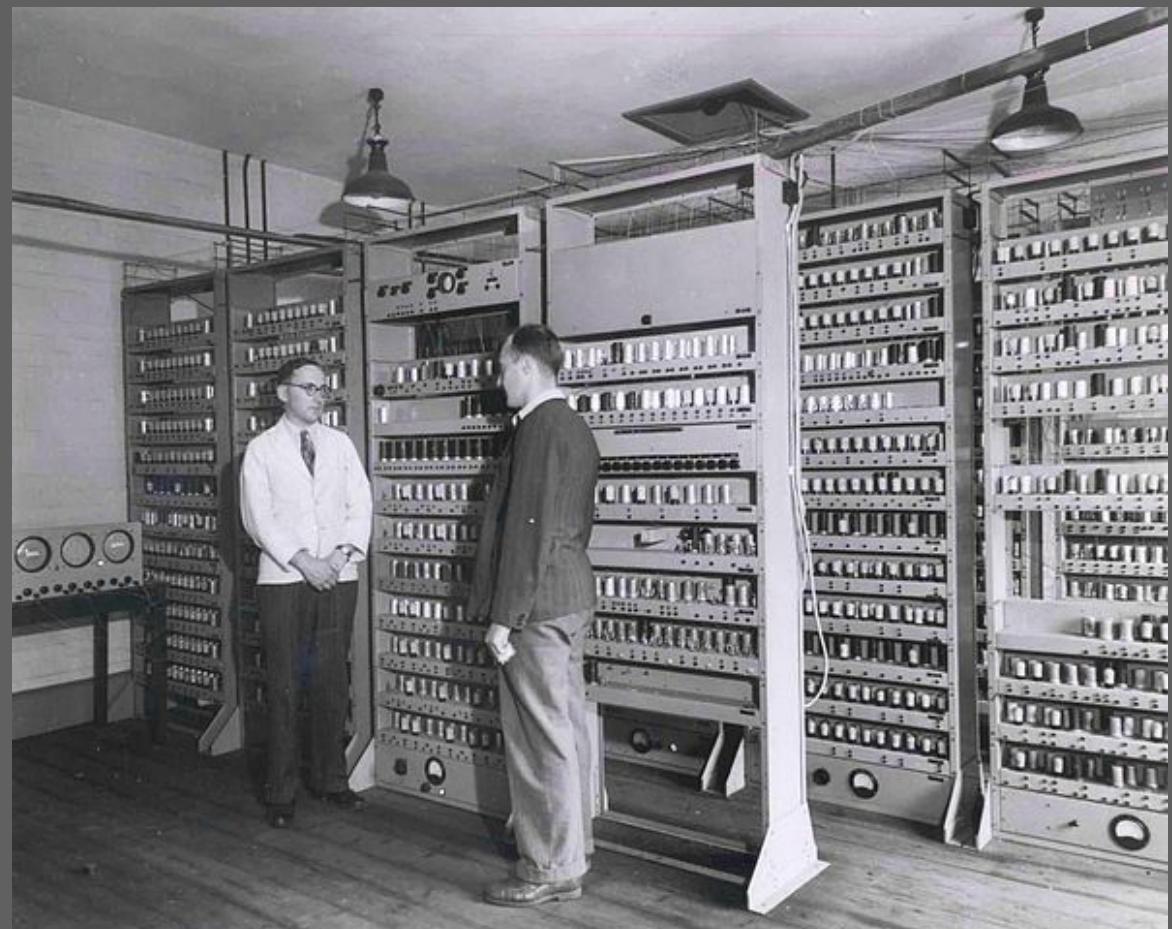
- Nel **Manchester SSEM** del 1948, il primo computer a programma memorizzato, le istruzioni macchina e i dati, scritti in binario, venivano inseriti a mano uno per uno in memoria usando 32 switch per stabilire il valore di ciascun bit di ogni istruzione/dato.
- Nell'**EDVAC**, i bit che componevano il programma (scritto in linguaggio macchina) e i dati, tutti rappresentati in binario, erano inseriti uno a uno nelle **Mercury Delay Lines**.

Storia dei Linguaggi di Programmazione

- In sostanza, alla fine degli anni ‘40, erano in funzione sicuramente meno di 20 computer, il concetto di *programmazione* come lo intendiamo noi era totalmente nuovo, e si potevano contare poche decine di *programmatori*.
- Ovviamente non esistevano corsi o tutorial di programmazione, e si imparava a programmare semplicemente seguendo il lavoro di qualcuno che aveva imparato prima oppure, ancora più semplicemente, provando per conto proprio.
- Pochissimi programmi erano stati scritti, e naturalmente erano tutti in linguaggio macchina, dato che assembler e compilatori non erano ancora stati inventati (e neppure era chiaro che ci fosse bisogno di qualcosa come un *linguaggio ad alto livello*). 9

Gli “Initial Orders” dell’EDSAC

- Il 6 maggio 1949 A Cambridge (UK) diviene operativo l’**EDSAC** (Electronic Delay Storage Automatic Calculator), progettato e costruito da un team guidato da **Bill Renwick** e **Maurice Wilkes**.
- Wilkes e Renwick di fronte all’EDSAC



Licensed under CC BY 2.0 via Wikimedia Commons

Gli “Initial Orders” dell’EDSAC

- L’EDSAC è il secondo computer a programma memorizzato dopo l’SSEM, ma il primo in termini di operatività non solo sperimentale, e il primo ad essere usato per applicazioni commerciali.
- L’EDSAC usa tubi catodici per la logica, una MDL da 512 words a 18 bit, due soli registri (in seguito portati a tre), un lettore di nastro perforato per l’input e una telescrivente per l’output.
- Ha un Instruction Set di 18 istruzioni macchina, ed è in grado di eseguire in media 600 istruzioni al secondo, che sono state inserite nella MDL leggendole da un nastro perforato alla velocità di circa 50 istruzioni al secondo.

Gli “Initial Orders” dell’EDSAC

- Nell’EDSAC viene introdotta per la prima volta l’idea di dare alle istruzioni macchina (chiamate **Initial Orders**) una forma mnemonica e dunque più facilmente usabile da essere umani.
- In sostanza, il codice operativo di ognuna delle 18 istruzioni macchina era associato ad una diversa lettera dell’alfabeto.
- Al codice operativo seguiva un numero scritto su due cifre decimali, che rappresentava l’indirizzo in memoria dell’operando esplicito dell’istruzione, e che usava come operando implicito il registro Accumulatore. Ad esempio l’istruzione:

“*S 25*” corrispondeva a:

Accumulator := Accumulator – value stored at address 25 ¹²

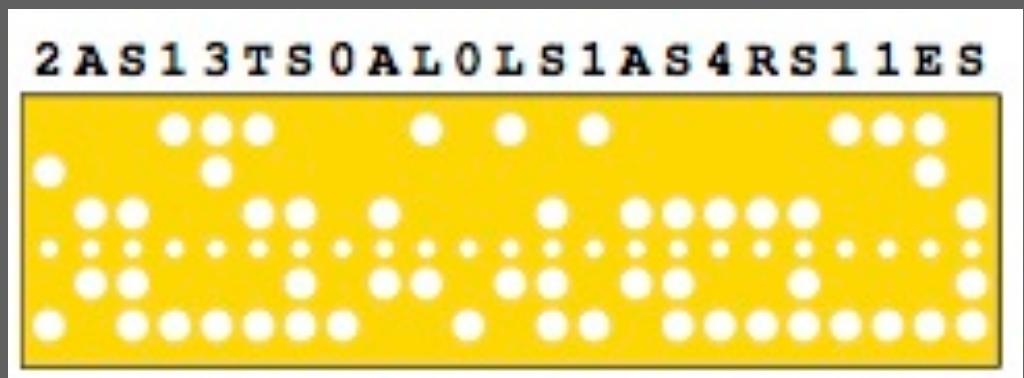
Gli “Initial Orders” dell’EDSAC

- 5 bit erano dunque sufficienti a codificare 18+10 caratteri, e una istruzione macchina poteva essere codificata in 5+5+5 bit.
- Un programma veniva quindi inizialmente stilato su carta scrivendo le istruzioni in forma mnemonica. Ad esempio:

```
A 10    // A := A + [10]
S 11    // A := A - [11]
T 12    // [12] := A
O 12    // print on teletype content of memory location [12]
```

Gli “Initial Orders” dell’EDSAC

- Il programma andava poi codificato su nastro perforato usando un perforatore manuale dotato di un tasto per ciascuno dei caratteri usati per scrivere le istruzioni.
- Premendo il tasto corrispondente ad esempio alla lettera “A” il nastro veniva perforato così da riprodurre il pattern di 5 bit 0 (foro assente) e 1 (foro presente) corrispondenti al codice di A.
- Il nastro veniva poi letto da un lettore in modo da inserire le istruzioni che conteneva nella memoria dell’EDSAC



Gli “Initial Orders” dell’EDSAC

- In sostanza quindi gli Initial Orders rappresentavano una **forma primitiva di Assembler**.
- In realtà, il concetto di un vero è proprio “programma assemblatore” che converte in linguaggio macchina il codice scritto in forma simbolica non era ancora presente nell’idea degli Initial Orders.
- Piuttosto, era il codice macchina stesso che era stato progettato in modo tale da poter essere inserito in memoria attraverso una sua rappresentazione mnemonica, associando il codice operativo di ogni istruzione ad una diversa lettera dell’alfabeto.

Gli “Initial Orders” dell’EDSAC

- Con l’EDSAC nasce anche il concetto di **subroutine**: porzioni di codice che possono essere richiamate più volte, anche in modo nidificato. L’idea si deve a **David Wheeler**, che è stato anche il primo ricercatore a conseguire un PhD in Computer Science.
- Nota di colore: nel 1952 Sandy Douglas sviluppa sull’EDSAC il **primo videogioco della storia**: il Tris (TIC TAC TOE) con output grafico su tubo catodico.
- Con gli Initial Orders nasceva l’idea di codificare all’interno della memoria di un computer simboli e caratteri alfanumerici attraverso specifici codici numerici, un’idea sviluppata più ampiamente nello Short Code.

Lo Short Code di John Mauchly

- Più o meno contemporaneamente agli Initial Orders dell'EDSAC, Nel 1949 **John Mauchly** concepisce quello che possiamo considerare il primo linguaggio di programmazione ad alto livello: lo **Short (Order) Code** (inizialmente chiamato **Brief Code**)
- Lo Short Code viene implementato alla fine dell'estate del 1949 da **William Shmitt**, un giovane laureato assunto come lavoratore stagionale alla **Eckert-Mauchly Computer Corporation (EMCC)**.
- Concepito inizialmente da Mauchly per il **BINAC** (uno dei computer sviluppati a partire dall'esperienza dell'**EDVAC**) non girò mai su quel computer, e nel 1950 fu poi adattato e migliorato per essere usato sull'**UNIVAC I**.

Lo Short Code di John Mauchly

- Nota di colore: abbiamo già incontrato **Grace Murray Hopper**, che nel 1945 scopre sul **Mark II** il primo bug (letteralmente) della storia dell'informatica.
- Nel 1949 la Hopper era a capo del Laboratorio di Analisi Computazionale (sostanzialmente il Dipartimento Programmazione) della EMCC, formato da 9 persone.
- Tra le altre cose, questo gruppo doveva istruire i programmati esterni dei computer costruiti dalla EMCC.
- Il primo corso fu fornito al personale della *Northup Aircraft Corp.* (che aveva commissionato il BINAC). Questo si può considerare probabilmente **il primo corso di programmazione della storia**.¹⁸

Lo Short Code di John Mauchly

- Poiché il BINAC (e qualsiasi altro computer di quegli anni, se non si considerano gli Initial Orders dell'EDSAC) lavorava solo usando numeri, l'idea dello Short Code era di descrivere le equazioni matematiche da calcolare in modo simbolico, assegnando un numero specifico, scritto su sei bit, ad ogni simbolo dell'equazione.
- Ad esempio, l'equazione: $X = Y + (X * Z)$

diventava:

40 03 41 07 09 40 42 02

- Con l'operatore di moltiplicazione rappresentato implicitamente dal codice di due variabili adiacenti.

Lo Short Code di John Mauchly

- In termini moderni, questo equivaleva ad un calcolo con assegnamento del tipo:

$$X := Y + (X * Z)$$

- In cui per le variabili venivano riservati alcuni valori specifici, e gli altri simboli erano associati ciascuno ad un numero ben preciso: “+” = 07, “=” = 03, “(“ = 09, e così via.
- In sostanza, con lo Short Code emergeva l’idea di codificare all’interno della memoria di un computer simboli e caratteri alfanumerici attraverso specifici codici numerici, poi opportunamente interpretati all’interno del computer.

Lo Short Code di John Mauchly

- Le equazioni da calcolare, convertite in Short Code, venivano inserite nella memoria dell'UNIVAC a mano, configurando una opportuna sequenza di switch.
- Mauchly e Shmitt erano consapevoli che il procedimento di traduzione di una equazione in Short Code avrebbe potuto essere automatizzato, ma la cosa non fu mai implementata.
- Shmitt lavorò anche alla scrittura del programma che interpretava le equazioni da calcolare una volta inserite nell'UNIVAC:
il primo interprete della storia dell'informatica
(con un rallentamento dei calcoli da eseguire di circa 50 volte...).

Nota a margine: il “compilatore” di Curry

- Più o meno tra il 1948 e il 1950, il logico **Haskell Curry**, che aveva scritto alcuni programmi per l’ENIAC, sviluppa una notazione che contiene alcune idee di base della programmazione strutturata (tra l’altro, *Haskell* e *Curry* sono i nomi di due linguaggi funzionali così chiamati in suo onore).
- Inoltre, egli illustra alcuni algoritmi ricorsivi per convertire la sua notazione matematica in linguaggio macchina. Questi algoritmi (benché rimasti solo su carta) possono considerarsi il primo esempio di descrizione della fase di **generazione del codice di un compilatore**.
- Va detto che nel suo lavoro Curry non prendeva in considerazione la fase di **analisi sintattica**, dando per scontato che ogni espressione aritmetica potesse essere parsificata correttamente².

La nascita del costrutto FOR

- Tra il 1949 e il 1951 **Heinz Rutishauser**, mentre lavora sullo **Z4** ricostruito e trasferito a Zurigo, fornisce la descrizione su carta di un nuovo computer, e di un algoritmo che traduce in linguaggio macchina un formalismo algebrico derivato dal Plankalkül).
- Il linguaggio algebrico formulato da Rutishauser introduceva per la prima volta un costrutto fondamentale dei moderni linguaggi di programmazione: **il costrutto FOR**.
- Come nello Short Code, Rutishauser usava codici numerici specifici per ogni carattere alfanumerico usato nel suo linguaggio algebrico e, cosa quasi straordinaria per quel periodo, il codice (ipoteticamente) generato era **rilocabile**.

Il compilatore di Corrado Böhm

- Nel 1950 l’italiano **Corrado Böhm** (che conosce il Plankalkül e i Flow Diagrams ma non il lavoro di Rutishauser) definisce un linguaggio ad alto livello, un linguaggio macchina, e un metodo di traduzione del primo nel secondo.
- Molti e notevoli sono gli aspetti innovativi del lavoro di Böhm, sia nel linguaggio ad alto livello che nel relativo compilatore:
 - L’uso dell’equivalente dei concetti di **goto** e **if-then-else**
 - L’uso dello **statement di assegnamento** in tutte le istruzioni
 - La possibilità di usare **subroutine**
 - La definizione completa di **un compilatore scritto nello stesso linguaggio dei programmi che deve compilare.**

Il compilatore di Corrado Böhm

- Una volta venutone al corrente, nella sua tesi di dottorato del 1951 Böhm confronta il suo lavoro con quello di Rutishauser, e la qualità del lavoro di Böhm risulta ancora più chiara:
- L'intero compilatore di Böhm consta di sole 130 istruzioni per il controllo sintattico e la generazione del codice macchina.
- Il compilatore genera codice in un numero di passi proporzionale alla lunghezza n del programma da compilare, mentre quello di Rutishauser richiedeva un numero di passi proporzionale a n^2 .
- Infine, di tutti i lavori di quegli anni, quello di Böhm è l'unico (e quindi di gran lunga il primo) a menzionare e a mostrare che **il suo linguaggio è universale, cioè capace di calcolare qualsiasi funzione computabile.**

Il primo vero compilatore operativo: AUTOCODE

- Il primo compilatore (degno di questo nome, come vedremo fra poco) effettivamente implementato e usato viene sviluppato da **Alick Glennie** in Inghilterra, e comincia a diventare operativo dal settembre 1952.
- Glennie chiamò il suo sistema **AUTOCODE**. Negli anni successivi, il suo lavoro influenzerà lo sviluppo di molti altri “Autocode” non necessariamente correlati fra loro.
- L’AUTOCODE fu sviluppato per il Manchester Mark 1, il cui linguaggio macchina era particolarmente astruso e complicato.

Il primo vero compilatore operativo: AUTOCODE

- In effetti, un evidente vantaggio nella definizione dei compilatori concepiti su carta da Rutishauser e Böhm era che facevano riferimento ad un sottostante linguaggio macchina “virtuale”, pensato apposta per semplificare il lavoro del compilatore.
- Con un computer reale invece la cosa non era possibile: quale che fosse il linguaggio ad alto livello adottato, andava tradotto nelle istruzioni macchina del computer su cui doveva girare.
- Glennie era tuttavia consapevole di una cosa (che oggi noi diamo per scontata): **possiamo usare lo stesso computer per tradurre il codice ad alto livello in codice macchina e poi farci girare sopra quello stesso programma.**

Il primo vero compilatore operativo: AUTOCODE

- Benché più semplice del linguaggio macchina del Mark 1, l'AUTOCODE era ancora piuttosto complicato, e il corrispondente compilatore era lungo circa 750 istruzioni macchina.
- Alcune osservazioni di Glennie nell'usare l'AUTOCODE (invece che linguaggio macchina) suonano oggi piuttosto famigliari:
- *“sono colpito dalla velocità con la quale si possono scrivere programmi e essere ragionevolmente certi che siano corretti”*.
- *“una caratteristica importante è la facilità con cui si può ragionare mentalmente sul programma che si sta scrivendo”*
- *“La perdita di efficienza è limitata a circa il 10%”*

Il primo vero compilatore operativo: AUTOCODE

- In realtà, il lavoro di Glennie non ebbe l'impatto che potremmo supporre, poiché era sostanzialmente troppo in anticipo sui tempi.
- Di tutti i problemi che i programmatori di quegli anni dovevano affrontare, alle prese con macchine instabili e lente, il meno grave era ancora quello della scrittura dei programmi.
- Ben più difficile e oneroso era il lavoro necessario per **adattare le grandezze numeriche del problema da risolvere alla limitata capacità del computer**, per non parlare delle difficoltà di **usare bene la pochissima memoria disponibile** e i continui **malfunzionamenti hardware che facevano fallire l'esecuzione dei programmi senza alcun indizio del perché...**

Nota a margine: la terminologia in uso all'inizio degli anni '50.

- All'inizio degli anni '50 il termine **Linguaggio di Programmazione** non era ancora entrato in uso, e si parlava piuttosto di **pseudo code** o di **automatic programming**. Non si parlava di **compilatori** ma di **automatic coding**.
- Ma nella primavera del 1951 (dunque prima dell'AUTOCODE di Glennie) Grace Hopper sviluppa un sistema in cui alcune “pseudo istruzioni” possono essere tradotte in linguaggio macchina da un programma che lei chiama **compiling routine**.
- Il programma di Hopper usava un linguaggio molto più involuto dell'AUTOCODE, e di fatto eseguiva quella che chiameremmo oggi una **espansione di Macro istruzioni assembler**.

Il fermento degli anni 1954-1956

- Il periodo che va dal 1954 al 1956 risulta particolarmente fertile di nuove idee ed implementazioni varie per la storia dei linguaggi di programmazione.
- Grace Hopper tiene interventi in varie parti degli USA, e soprattutto lavora all'organizzazione delle prime due edizioni del **Symposium on Advanced Programming Methods for Digital Computers** che in quegli anni era la principale (se non l'unica) occasione che i ricercatori avevano di venire a conoscenza del lavoro svolto in altre sedi e di scambiarsi opinioni e esperienze.
- Ecco qui di seguito un breve elenco dei contributi di quegli anni.

Il fermento degli anni 1954-1956

- È proprio al simposio del 1954 che **Halcombe Laning** e **Neal Zierle** presentano il loro sistema, implementato per il Whirlwind.
- Il sistema, da alcuni chiamato “**George**”, per la prima volta:
 - 1) usava un linguaggio svincolato dal computer su cui girava
 - 2) non richiedeva conoscenze pregresse sul computer usato
- Il codice generato dal compilatore era circa 10 volte più lento dei programmi scritti direttamente in codice macchina, ma si dimostrò molto efficiente nel permettere una stesura veloce di programmi per problemi molto complessi.

Il fermento degli anni 1954-1956

- Sempre nel 1954 **Ralph Brooker** implementa a Manchester un nuovo **AUTOCODE** per il Mark 1.
- Rispetto al precedente AUTOCODE di Glennie, quello di Brooker è più elegante e meno machine-dependent. Inoltre, era in grado di lavorare sia con variabili intere che con **variabili floating point**.
- Sulla Base del lavoro di Brooker, altri due AUTOCODE furono sviluppati negli anni successivi per il Ferranti PEGASUS e per il Mark II.

Il fermento degli anni 1954-1956

- Anche in Russia vengono portate avanti in quegli anni ricerche su computer, linguaggi e compilatori.
- Nel 1954 vengono messi a punto i computer **STRELA** e **BESM**, e nel 1955/56 vengono sviluppati sistemi (linguaggio + compilatore corrispondente) chiamati, piuttosto appropriatamente:
Programming Programs.
- I linguaggi usati erano semplici ed eleganti, e i relativi compilatori erano in grado di riconoscere sottoespressioni già compilate in modo da non ripetere inutilmente un lavoro già fatto.

Il fermento degli anni 1954-1956

- Tra il 1954 e il 1956 alla **Boeing Airplane Company** di Seattle viene sviluppato il sistema **BACAIC**, in cui espressioni algebriche vengono tradotte in subroutine di linguaggio macchina.
- Nel 1955 all'università della California di Livermore, **Kenton Elsworth** lavora ad un sistema per la traduzione di equazioni algebriche in linguaggio macchina (dell'IBM 701), e chiama il suo sistema **Kompiler** (termine che finalmente fa la sua comparsa)
- Nel 1955/56 **E. Blum** presenta **ADES**, il primo linguaggio di programmazione dichiarativo, basato sulla teoria delle funzioni ricorsive di **Stephen Kleene**.
- Nel 1954 viene sviluppato il primo assembler moderno, per l'IBM 701, seguito nel '55 dal più famoso assembler del 650, il **SOAP: Symbolic Optimal Assembly Program**

Il fermento degli anni 1954-1956: IT

- Nel 1956, al **Carnegie Institute of Technology**, **Alan Perils** e **Joseph Smith** sviluppano sull'IBM 650 un linguaggio ed un compilatore che chiamano **IT (Internal Translator)**
- Il compilatore IT funzionava in 2 fasi: prima generava codice assembler intermedio, e poi generava da questo il codice macchina (l'uso del SOAP come codice intermedio rendeva estremamente efficiente l'intero processo di compilazione)
- IT fu il primo sistema realmente utile, e fu infatti installato in centinaia di copie su altrettanti IBM 650, e rimase in uso fino a che queste macchine non divennero obsolete.

Il fermento degli anni 1954-1956: IT

- Nessuno dei sistemi visti fino ad ora riuniva insieme tutte le caratteristiche contenute in IT: un **linguaggio potente ed espressivo**, una **implementazione efficiente**, una **adeguata documentazione** (ovviamente, rispetto agli standard medi di quegli anni)
- Finalmente, queste caratteristiche riunite assieme potevano avere un impatto significativo sull'uso di un computer.
- Inoltre, IT ebbe l'importantissimo ruolo di dimostrare che era possibile sviluppare compilatori utili anche per computer relativamente poco potenti (come l'IBM 650) senza richiedere enormi investimenti.

Il fermento degli anni 1954-1956: IT

- Ecco come una porzione di codice C (supponiamo che gli elementi dell'array di numeri reali “a” siano stati letti in precedenza):

```
for (i = 0; i < 11; i++)  
{  
    x = sqrt(a[i]) + 5*a[i]*a[i]*a[i]);  
    if ( x > 400.0 ) x = 999.0;  
    printf("%d, %f", i, x);  
}
```

- risulterebbe scritto in IT. Notate che i numeri delle righe sono semplici etichette usate all'interno delle istruzioni

```
1: READ  
2: 3, I1, 10, -1, 0,  
5: Y1 ← "20E, AC(I1+1)"  
   +(5×(C(I1+1)*3))  
6: G3 IF 400.0 ≥ Y1  
7: Y1 ← 999  
3: T11 TY1  
10: H
```

Il fermento degli anni 1954-1956: IT

- Ovviamente, rispetto agli standard a cui siamo abituati oggi questo codice risulta piuttosto criptico. Possiamo però apprezzare quanto fosse sintetico.
- Inoltre, risulta sostanzialmente svincolato da qualsiasi aspetto specifico dell'hardware sottostante. Una caratteristica fondamentale di un linguaggio ad alto livello.
- In realtà, l'IBM 650 non aveva un set di caratteri sufficientemente ricco per scrivere il programma del lucido precedente, che veniva invece perforato su schede così, usando opportune lettere al posto dei caratteri mancanti:
K = “,” M = “—” L = “(”
eccetera...

0001	READ	F
0002	3K 11K 10K M1K OK	F
0005	Y1 Z Q 20EK ACL11S1R Q	F
0005	S L5 X LCL11S1R P 3RR	F
0006	G3 IF 400JO W Y1	F
0006	Y1 Z 999	F
0003	T11 TY1	F
0010	H	FF

Il FORTRAN

- A fine 1956 la terminologia in uso si è ormai stabilizzata ed è accettata più o meno da tutti, e i termini **compilatore (compiler)** e **istruzione (statement)** vengono usati nel senso con cui li usiamo ancora oggi.
- E nell'aprile del 1957 fa finalmente la sua comparsa sulla scena il **FORTRAN**, il linguaggio più famoso e celebrato di quegli anni, e anche il primo che (attraverso successive incarnazioni, fino al FORTRAN 2015, previsto in realtà per il 2018) sia sopravvissuto fino ai giorni nostri.
- In realtà il FORTRAN è in gestazione già da più di due anni, il suo lavoro di sviluppo ampiamente pubblicizzato, e l'attesa per poterlo finalmente usare è carica di aspettative.

Il FORTRAN

- All'inizio del 1954, alla IBM, **John Backus, Harlan Herrick e Irvin Ziller** si riuniscono per studiare lo sviluppo di un *sistema per la programmazione automatica* (notate la terminologia antiquata).
- Nel maggio del 1954 si recano al MIT per vedere in azione e studiare il sistema di Laning e Zierler.
- Dieci anni dopo Backus osserverà come:

“in quel periodo i programmi erano scritti in assembler, se non in ottale o decimale, e nessuno credeva che un compilatore avrebbe saputo applicare l'ingegnosa versatilità che si pensava fosse necessaria per scrivere programmi, e che ogni programmatore pensava di possedere”

Il FORTRAN

- Dunque:

“tutti concordavano che i compilatori potevano solo produrre codice [macchina] intollerabilmente meno efficiente di quello scritto da un programmatore.”

- Backus e il suo gruppo temevano quindi che, una volta sviluppato il loro sistema, questo non sarebbe stato capace di generare codice competitivamente efficiente con quello scritto a mano. Ovviamente nessuno avrebbe voluto usare un tale sistema di programmazione.
- In ogni caso, per la fine del 1954 il gruppo guidato da Backus aveva completato le specifiche del “*IBM Mathematical **FOR**mul**A** **TRAN**slating System*”, da tutti universalmente conosciuto come il **FORTRAN**

Il FORTRAN

- Nel report delle specifiche Backus e colleghi osservavano che:
 1. Mentre i sistemi esistenti fornivano facilità di sviluppo del codice ma inefficienza, o viceversa, sarebbe stato facile scrivere i programmi in FORTRAN e questi sarebbero stati efficienti da eseguire.
 2. Non ritenevano fondamentale che il FORTRAN fosse svincolato dall'hardware sottostante (In questo caso l'IBM 704), ma che avesse una notazione matematica concisa e non *assembler-like*.
 3. Un'ora di tutorial del linguaggio sarebbe stato sufficiente a capire i passi di un qualsiasi programma scritto in FORTRAN senza dover aggiungere alcun commento.

Il FORTRAN

- Ecco come la porzione di codice che abbiamo già visto per l'IT sarebbe stata scritta nella versione originale del FORTRAN.
- Notate la dichiarazione di un array di 11 elementi, e come la lettura di un valore (su scheda perforata) avvenga con la sola istruzione “READ”
- L’istruzione 3 significa: esegui le istruzioni dalla label 3 alla 8 per J che va da 1 a 11.
- Notate il GOTO: non fa ripartire dall’inizio il loop, inizia solo la successiva iterazione.

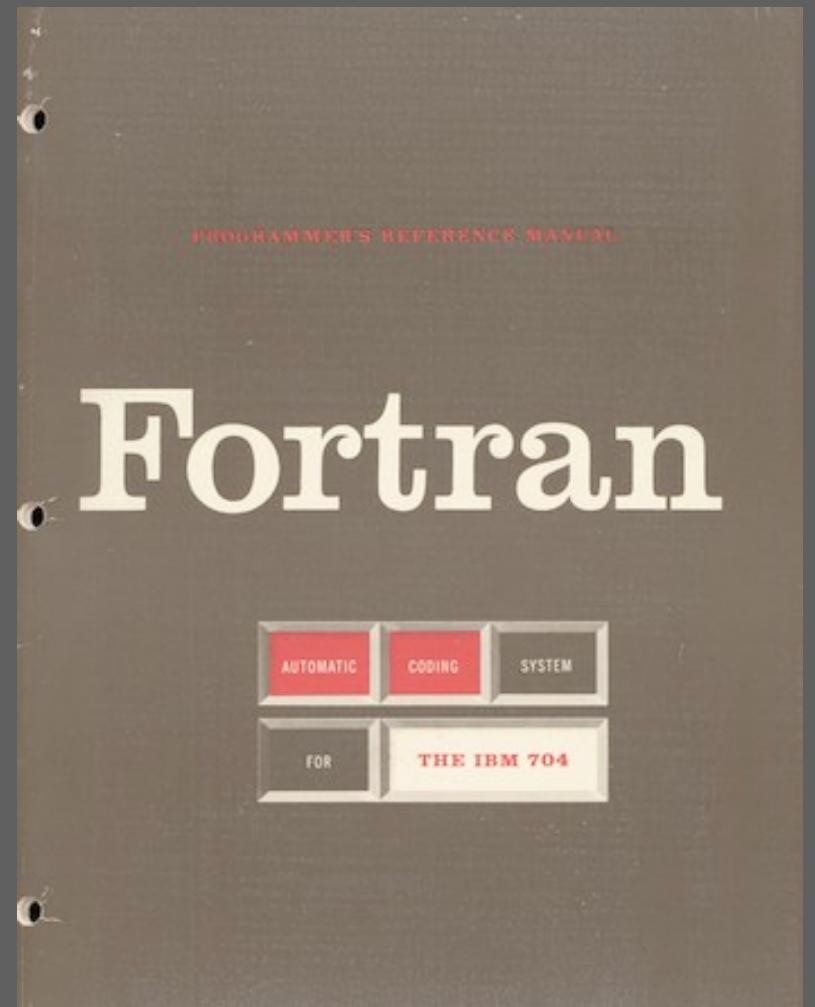
```
1      DIMENSION A(11)
2      READ A
3      2 DO 3,8,11 J=1,11
4      3 I=11-J
5      Y = SQRT(ABS(A(I+1)))+5*A(I+1)**3
6      IF (400. >= Y) 8,4
7      4 PRINT I,999.
8      GO TO 2
9      8 PRINT I,Y
10     11 STOP
```

Il FORTRAN

- Naturalmente nel 1954 molti aspetti della programmazione ad alto livello che noi diamo per scontate nei linguaggi moderni non esistevano, o stavano appena emergendo.
- Il costrutto “GOTO” era considerato normale e non fonte di potenziali errori di programmazione.
- Per la prima volta era possibile usare nomi di variabili composti da due lettere, il che era una innovazione notevole rispetto alle convenzioni matematiche.
- Il report del FORTRAN era il primo tentativo di specificare in modo rigoroso la sintassi di un linguaggio di programmazione, e in esso cominciava a manifestarsi in modo embrionale quella che diventerà universalmente nota come **BNF**: la **Backus-Naur Form**

Il FORTRAN

- Dunque dopo più di due anni, annunci e ritardi vari, il FORTRAN viene reso disponibile, nell'aprile del 1957.
- Il suo manuale ha una grafica professionale e promette meraviglie.
- In realtà, come tutti i sistemi nuovi di quel periodo, era pieno di errori e con molte parti incomplete.
- Molti programmati erano convinti che non avrebbe mai funzionato, ma col tempo divenne plausibile aspettarsi che il compilatore funzionasse, e persino che il codice generato riuscisse a girare...



Il FORTRAN

- Per fortuna la situazione cambiò rapidamente, e un anno dopo circa la metà delle installazioni di computer IBM 704 (in tutto comunque poco più di una cinquantina) avevano usato il FORTRAN per almeno la metà dei programmi sviluppati.
- Nel frattempo la sua sintassi era un po' cambiata rispetto al 1954:
- L'uso di output FORMATtato.
- L'introduzione dello statement CONTINUE
- La possibilità di aggiungere commenti alle istruzioni.

```
FUNF(T) = SQRTF(ABSF(T))+5.0*T**3
DIMENSION A(11)
1  FORMAT(6F12.4)
      READ 1, A
      DO 10 J = 1,11
           I = 11-J
           Y = FUNF(A(I+1))
           IF (400.0-Y)4,8,8
4   PRINT 5, I
5   FORMAT(1I0, 1OH TOO LARGE)
           GO TO 10
8   PRINT 9, I, Y
9   FORMAT(1I0, F12.7)
10  CONTINUE
      STOP 52525
```

Il FORTRAN

- È impossibile sottostimare l'importanza che ha avuto il FORTRAN nella storia dei linguaggi di programmazione.
- È stato il primo linguaggio di programmazione ad alto livello dotato di un compilatore in grado di generare codice efficiente, e ha influenzato lo sviluppo della maggior parte dei linguaggi degli anni successivi.
- Fino a quasi tutti gli anni '70 è stato il linguaggio di programmazione più usato per applicazioni scientifiche.
- Si è tuttavia portato dietro per molti anni i difetti della versione originale. Solo col FORTRAN 66 viene definita una versione standard del FORTRAN, e solo col FORTRAN 77 vengono introdotte primitive adeguate per la programmazione strutturata

MATH-MATIC e FLOW-MATIC

- Più o meno in parallelo allo sviluppo del FORTRAN, alla Remington Rand, Grace Hopper guida il gruppo di lavoro che sviluppa sull'UNIVAC due linguaggi caratterizzati da una particolare leggibilità. Il primo è il **MATH-MATIC**.
- Disponibile dall'aprile del 1957, il MATH-MATIC ha purtroppo il difetto di essere estremamente inefficiente (rispetto ad esempio al FORTRAN) e in più è stato sviluppato per l'UNIVAC, una macchina molto più lenta dell'IBM 704.

```
(1) READ-ITEM A(11) .
(2) VARY I 10(-1)0 SENTENCE 3 THRU 10 .
(3) J = I+1 .
(4) Y = SQR |A(J)| + 5*A(J)3 .
(5) IF Y > 400, JUMP TO SENTENCE 8 .
(6) PRINT-OUT I, Y .
(7) JUMP TO SENTENCE 10 .
(8) Z = 999 .
(9) PRINT-OUT I, Z .
(10) IGNORE .
(11) STOP .
```

MATH-MATIC e FLOW-MATIC

- Al MATH-MATIC segue, nel 1958, il **FLOW-MATIC**, il quale è ancora più orientato ad istruzioni scritte in “linguaggio naturale”, ed è pensato per applicazioni gestionali (Business), dove non sono usate complesse operazioni matematiche:

```
(1)  COMPARE PART-NUMBER (A) TO PART-NUMBER (B) ; IF GREATER GO TO  
      OPERATION 13 ; IF EQUAL GO TO OPERATION 4 ; OTHERWISE GO TO  
      OPERATION 2 .  
(2)  READ-ITEM B ; IF END OF DATA GO TO OPERATION 10 .
```

- Il FLOW-MATIC è importante soprattutto perché è considerato il linguaggio di programmazione che ha ispirato lo sviluppo del **COBOL** (che ritroveremo fra poco).

Il LISP

- Nel 1958 fa il suo debutto quello che, dopo il FORTRAN, è il linguaggio di programmazione più vecchio ancora in uso, il **LISP**.
- Interessante osservare che alcune parti del primo sistema LISP furono già scritte in FORTRAN: una buona idea è spesso utile nello sviluppo di altre buone idee.
- Il LISP (**LIS**t Processing language) fu concepito e sviluppato tra il 1956 e il 1858 da **John McCarthy** come strumento per scrivere programmi seguendo un preciso formalismo matematico.
- Di fatto, il LISP è quello che oggi chiameremmo un **linguaggio di programmazione funzionale**, basato sul formalismo del λ -calcolo e sulla teoria delle funzioni ricorsive di **Alonso Church** e **Stephen Kleene** degli anni ‘30.

Il LISP

- Il LISP è un **linguaggio di programmazione per rappresentare e manipolare espressioni simboliche**, e il numero di idee innovative contenute nel LISP è impressionante:
 1. primo esempio di linguaggio funzionale
 2. Uso della ricorsione
 3. Uso di liste concatenate e alberi come strutture dati (tutte le espressioni LISP sono rappresentate da liste concatenate)
 4. Allocazione dinamica della memoria e implementazione del garbage collection
 5. Possibilità di comporre funzioni in funzioni più complesse
 6. Metaprogrammazione: un programma LISP è a sua volta una lista che può essere manipolata da un altro programma LISP

Il LISP

- Il sistema LISP fu sviluppato inizialmente per l'IBM 704 ed era interpretato (il primo compilatore fu sviluppato nel 1962).
- Tuttavia:
 - la scelta di adottare, anziché la più comune notazione *infissa*, quella *prefissa* per le espressioni da valutare, ossia:
 $(+ 20 30)$ invece di $(20 + 30)$ e:
 - la possibilità di implementare direttamente nel codice macchina dell'IBM 704 alcuni degli operatori principali del LISP (CAR e CDR in particolare)
- Rendevano i programmi LISP interpretati sorprendentemente efficienti.

Esempi di codice LISP

: (car '(this is a list))

this

: (cdr '(this is a list))

(is a list)

: (cons this '(is a list))

(this is a list)

: (+ 20 30)

50

(define fact (n))
 (if (<= n 1)
 1
 (* n (fact (- n 1))))
))

*;lisp code is also data, and eval turns
data into code (thanks to S.R. Russell)*

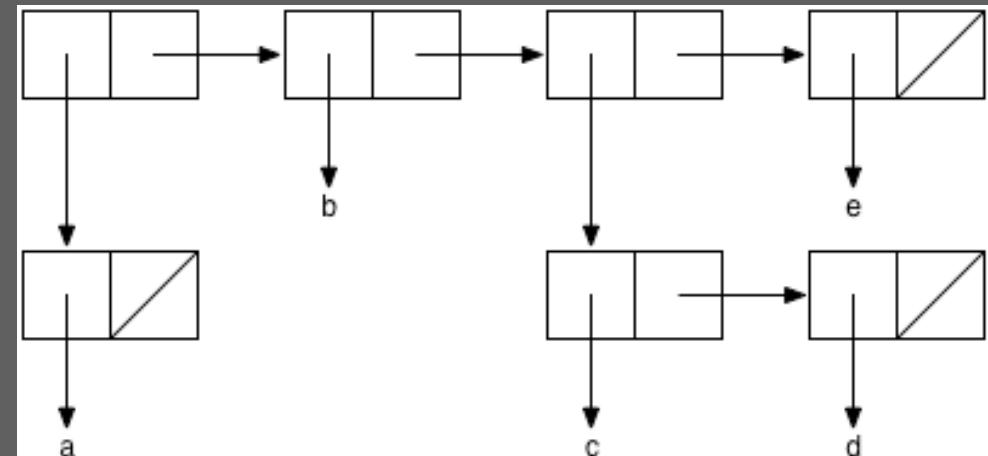
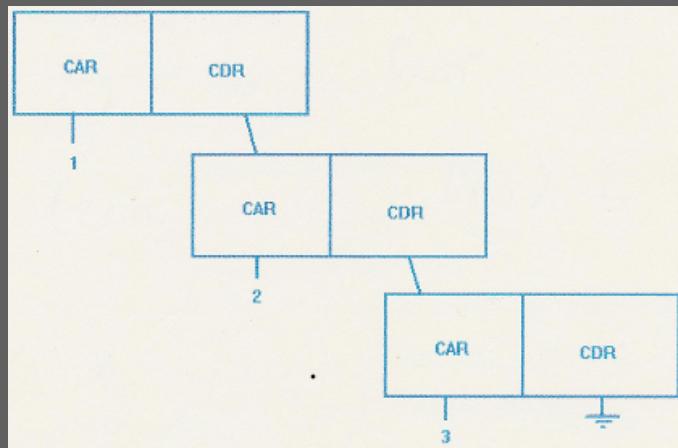
:(cons '* (cdr '(- 20 10)))

(* 20 10)

:(eval (cons '* (cdr '(- 20 10)))))

200

Le strutture dati del LISP



- In LISP tutto è rappresentato mediante liste concatenate, che portano ad una naturale formulazione ricorsiva dei dati.
- La lista concatenata a destra è la rappresentazione interna di dati (o codice) LISP che in codice sorgente sarebbe rappresentata così:

((a) b (c d) e)

Nota di colore: le parentesi del LISP

- Nelle liste concatenate del LISP gli elementi sono separati da spazi e delimitati da parentesi, e ogni elemento può a sua volta essere una lista concatenata.
- Dunque in LISP anche il più semplice programma abbonda di parentesi, e queste tendono a proliferare in modo incontrollato.
- Questo fu naturalmente soggetto a critiche e battute varie, e l'acronimo LISP fu ridefinito corrispondentemente:
- **LISP: Lots of Irritating Superfluous Parentheses**
- **LISP: Lost In Stupid Parentheses**
- **LISP: Lots of Isolated Silly Parentheses**

Il LISP

- Il LISP ebbe comunque un enorme successo, soprattutto negli ambienti accademici, e divenne velocemente uno dei (se non *il*) linguaggi di programmazione di riferimento per tutta la ricerca nel campo dell’Intelligenza Artificiale e della computazione simbolica in generale, almeno fino all’avvento del PROLOG.
- Negli anni ne sono state sviluppate diverse varianti e dialetti, tra cui:
 - **Franz LISP** (fine anni ‘70; notare il gioco di parole)
 - **Common LISP** (1984)
 - **Scheme** (c.a. 2007, ma con radici addirittura negli anni ‘70)

Il COBOL

- Nel 1959 fa il suo esordio il **COBOL**: **C**OMMON **B**USINESS **O**RIENTED **L**ANGUAGE. Un linguaggio concepito esplicitamente per applicazioni in campo amministrativo e commerciale.
- Infatti, il COBOL aveva una sintassi fortemente English-like, era facilmente comprensibile e le istruzioni non richiedevano ulteriori commenti.
- Ecco un esempio di codice in COBOL (in giallo costanti e variabili, in bianco le istruzioni e clausole riservate):

MULTIPLY **X** BY **X** GIVING **X-SQUARED**

MULTIPLY **5** BY **Y** GIVING **FIVE-Y**

MULTIPLY **FIVE-Y** BY **Z** GIVING **FIVE-Y-Z**

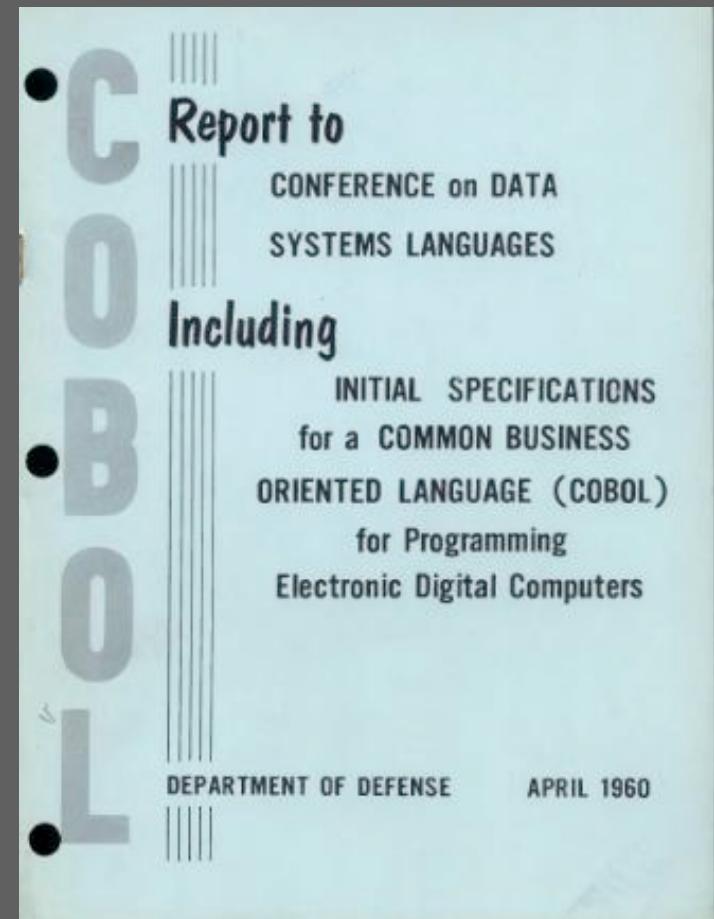
SUBTRACT **X-SQUARED** FROM **FIVE-Y-Z** GIVING **RES**

Il COBOL

- Il COBOL fu sviluppato da un consorzio (il **CODASYL**, formato da rappresentati dell'industria e del governo USA) fondato nel 1959 per guidare lo sviluppo di un linguaggio di programmazione fortemente portabile.
- Il **Department of Defense (DoD)** USA obbligò le compagnie produttrici di computer a supportare e fornire il COBOL nelle loro installazioni, e così il linguaggio si diffuse velocemente.
- La sintassi del COBOL fu il frutto di un laborioso processo decisionale da parte dei membri del consorzio, ma in definitiva il COBOL fu fortemente influenzato dal FLOW-MATIC (la Hopper servì come consigliere tecnico per il CODASYL), e solo molto parzialmente dal COMTRAN della IBM (poiché il CODASYL non voleva favorire la IBM...)

Il COBOL

- Tra gli obiettivi del COBOL vi era una ragionevole portabilità dei programmi su sistemi diversi (una qualità per nulla scontata per quei tempi), una buona efficienza, e la facilità d'uso da parte di programmatori inesperti.
- Il primo programma COBOL fu fatto girare nell'agosto del 1960.
- Negli anni '60 il COBOL ebbe una enorme diffusione, e negli anni '70 era di gran lunga il linguaggio di programmazione più usato al mondo.



Il COBOL

- Non di meno, il mondo degli informatici ha sempre guardato al COBOL (e ai programmatori in COBOL) con un'aria di sufficienza, se non di disprezzo (si veda più avanti l'osservazione in proposito di Dijkstra).
- Per molti anni fu data per scontata la dicotomia per cui i veri programmi, e in particolare quelli per applicazioni scientifiche, fossero scritti in FORTRAN, e quelli per applicazioni commerciali e amministrative in COBOL.
- Benché ancora ampiamente diffuso e modernizzato, il COBOL non sembra avere un brillante futuro: sempre meno applicazioni vengono scritte in COBOL, e in molte compagnie si tende a favorire la migrazione verso soluzioni alternative.

Gli anni ‘60: l’ALGOL 60

- La storia dell’**ALGOL 60** (**ALGO**rithmic **L**anguage 19**60**) inizia con l’ALGOL 58, che fu sostanzialmente il frutto di una collaborazione tra la **ACM** (Association for Computer Machinery) e la **GAMM** (l’istituto tedesco per la promozione delle scienze).
- In effetti, le implementazioni dell’ALGOL 58 (inizialmente chiamato **IAL**: International **A**lgebraic **L**anguage) furono ben poche, e questo linguaggio è ricordato principalmente per la sua influenza sull’ALGOL 60.
- Come vedremo nel seguito, al pari (e forse più) del FORTRAN, è impossibile sottostimare l’influenza avuta dall’ALGOL, influenza indicata anche dal gran numero (più di 70) di *dialetti* sviluppati nel corso degli anni, con l’evolvere della comprensione delle problematiche dei linguaggi di programmazione.

L'ALGOL 60

- La quantità di “record” posseduti dall’ALGOL 60 è notevole, e non a caso divenne il linguaggio di riferimento del mondo scientifico per tutti gli anni ’60: il linguaggio standard per la descrizione degli algoritmi (meno per la loro effettiva implementazione...).
- **Tony Hoare**, l’inventore del **Quicksort** (e anche del *Null Pointer*) osservò che l’ALGOL 60 era “*un linguaggio così avanti nei tempi da essere non solo un miglioramento rispetto ai suoi predecessori, ma anche rispetto a quasi tutti i suoi successori*”.
- L’ALGOL 60 però non era particolarmente adatto per le applicazioni commerciali, in quanto mancava di primitive di input/output veramente slegate dall’hardware sottostante, il che ovviamente ne limitava la portabilità su piattaforme diverse.⁶³

Principali innovazioni dell'ALGOL 60

- Nell'ALGOL 60 vengono definite per la prima volta in modo accurato le modalità di passaggio dei parametri alle procedure:
- **call by value**: l'espressione passata come argomento viene valutata e il risultato viene legato al parametro formale corrispondente nella procedura (di solito *copiando* il valore in una opportuna cella di memoria)
- **call by name**: una variante sofisticata del call by reference, in cui l'espressione passata come argomento viene valutata ogni volta (ma solo se) il parametro formale corrispondente nella procedura viene usato.
- **call by reference** (introdotto nelle versioni successive del linguaggio): in cui si passa un *riferimento* alla variabile usata⁶⁴ come argomento: tipicamente, il suo indirizzo in memoria.

Principali innovazioni dell'ALGOL 60

- Il *call by name* dell'ALGOL 60 è un interessante esempio di una nuova idea che non è sopravvissuta all'evoluzione.
- Sebbene potenzialmente molto espressivo, pochi linguaggi hanno adottato il *call by name*, che è stato velocemente sostituito dal *call by reference*, molto più facile da implementare e meno soggetto a possibili errori di programmazione.
- In effetti, il *call by reference* è una forma più naturale di passaggio dei parametri di una procedura, dato che in assembler e in linguaggio macchina le variabili vengono accedute attraverso l'indirizzo della cella di memoria che ne contiene il valore.
- È dunque singolare che l'ALGOL non abbia usato fin da subito questa modalità di passaggio dei parametri.

Principali innovazioni dell'ALGOL 60

- Ecco un esempio in ALGOL di uso elegante ed efficace del call by name, difficile da implementare con il call by value o reference (questo esempio è noto in letteratura come **Jensen's Device**):

calcolare la sommatoria di $a[i] \times i$ per i che va da 1 a n in una subroutine chiamata come “sum (i, 1, n, $a[i] \times i$)”

```
real procedure Sum(j, lo, hi, Ej);
  value lo, hi; integer j, lo, hi; real Ej;
begin
  real S;
  S := 0;
  for j := lo step 1 until hi do
    S := S + Ej;
  Sum := S;
end;
```

Principali innovazioni dell'ALGOL 60

- D'altra parte, il call by name non permette, in alcuni casi, di implementare in maniera semplice una procedura banale come lo swap del valore di due variabili:

```
procedure swap (a, b);  
integer a, b, temp;  
begin  
    temp := a;  
    a := b;  
    b:= temp  
end;
```

- Cosa succede se chiamiamo **swap (i, x[i])**?

Principali innovazioni dell'ALGOL 60

- L'ALGOL è il primo linguaggio a richiedere la dichiarazione del tipo delle variabili usate nei programmi con parole riservate: **INTEGER, BOOLEAN, REAL, DOUBLE, LONG, ARRAY**. Mancavano però i *record*, introdotti in seguito.
- Nell'ALGOL compare per la prima volta la distinzione precisa fra il simbolo di uguaglianza tra variabili: **a = b;** e il simbolo di assegnazione del valore di una variabile ad un'altra **a := b;**
- Compaiono per la prima volta **primitive di controllo strutturate** (anche se il GOTO non viene eliminato):

if then else

while

for until do

Principali innovazioni dell'ALGOL 60

- Compaiono per la prima volta i **blocchi di istruzioni**: porzioni di codice delimitati da **begin -- end** in cui è possibile dichiarare funzioni nidificate e variabili locali, note cioè solo all'interno del blocco corrispondente (**static scope** o **lexical scope** dei nomi):

```
begin;          // A program
integer I; long K;
procedure B;
real K; integer L;
procedure C; real L;
begin
    .... .... ....
end;    // scope = A+B+C
end;    // scope = A+B
end;    // scope = A
```

Il bubblesort in Algol (68)

```
MODE DATA = INT;  
PROC swap =  
  (REF[]DATA slice)VOID:  
  (  
    DATA tmp = slice[1];  
    slice[1] := slice[2];  
    slice[2] := tmp  
  );
```

```
main:(  
  [10] INT random :=  
    (1,6,3,5,2,9,8,4,7,0);  
  sort(random);  
  printf($10(g(3))l$, random))
```

```
PROC sort = (REF[]DATA array)VOID:  
  (  
    BOOL sorted;  
    INT shrinkage := 0;  
    FOR size FROM UPB array - 1 BY -1  
      WHILE  
        sorted := TRUE;  
        shrinkage +:= 1;  
        FOR i FROM LWB array TO size DO  
          IF array[i+1] < array[i] THEN  
            swap(array[i:i+1]);  
            sorted := FALSE  
          FI  
        OD;  
        NOT sorted  
        DO SKIP OD  
  );
```

La Backus Naur Form

- Un'altra innovazione fondamentale introdotta con l'ALGOL è l'uso di una notazione formale per descrivere la sintassi del linguaggio stesso: la **Backus Naur Form (BNF)**.
- Di fatto **la BNF è una notazione per grammatiche context free**, e dunque è usata non solo per descrivere la sintassi dei linguaggi di programmazione, ma anche, ad esempio, il formato di documenti e i protocolli di comunicazione.
- John Backus sviluppò (quella che ora chiamiamo) la BNF per descrivere la sintassi dell'ALGOL 58, definendola un *metalinguaggio di formule metalinguistiche*
- Peter Naur apportò alla notazione alcune modifiche e la usò per l'ALGOL 60, proponendo di chiamarla **Backus Normal Form**.

La Backus Naur Form

- Ecco ad esempio la descrizione della sintassi della dichiarazione di una procedura ALGOL 60 in BNF:
 - $\langle \text{procedure heading} \rangle ::= \langle \text{procedure identifier} \rangle \langle \text{formal parameterpart} \rangle ;$
 - $\langle \text{value part} \rangle \langle \text{specification part} \rangle$
 - $\langle \text{procedure body} \rangle ::= \langle \text{statement} \rangle \mid \langle \text{code} \rangle$
 - $\langle \text{procedure declaration} \rangle ::= \text{procedure } \langle \text{procedureheading} \rangle$
 - $\langle \text{procedure body} \rangle \mid \langle \text{type} \rangle \text{ procedure}$
 - $\langle \text{procedure heading} \rangle \langle \text{procedure body} \rangle$
- Può essere interessante notare come la sintassi stessa della BNF possa essere descritta usando la BNF.

Nota a margine: la Pāṇini Backus Form

- Fu Donald Knuth, nel 1964, a osservare che la Backus Normal Form non era affatto una forma normale, e che comunque avrebbe più correttamente dovuto chiamarsi, appunto, Backus Naur Form.
- Nel 1967 P. Ingerman suggerì sulle *Communication of the ACM* che la BNF avrebbe piuttosto dovuto chiamarsi **Pāṇini** Backus Form.
- Pāṇini è una figura quasi leggendaria tra i linguisti. Vissuto intorno al 400 A.C., compilò una grammatica completa del Sanscrito, la lingua colta dell'India la cui regole sintattiche e fonetiche furono definite a tavolino dai letterati del tempo (tra cui lo stesso Pāṇini).
- Questa grammatica era composta di 3959 regole, descritte mediante una notazione identica a quella della BNF, fino addirittura all'uso di simboli equivalenti a quelli utilizzati nella BNF: “|” “<“ e “>”. 73

Nota a margine: i parsificatori

- Le **grammatiche context free** furono definite in modo formale da **Noam Chomsky** alla metà degli anni '50, e la sintassi dei linguaggi di programmazione viene normalmente descritta usando grammatiche context free perché ciò porta alla costruzione di parsificatori efficienti.
- Il progetto dell'ALGOL portò ad investigare tecniche di parsificazione adeguate e allo sviluppo, nel 1961, del concetto di **parsificatore LL**, in grado di parsificare i linguaggi generati da un sottoinsieme delle grammatiche context free (ma sufficiente per l'ALGOL).
- Nel 1965 Donald Knuth definì il **parsificatore LR**, in grado di parsificare i linguaggi generati da qualsiasi grammatica context free deterministica. Fu tuttavia solo a partire dalla fine degli anni '60 che furono descritte implementazioni efficienti per questi parsificatori.⁷⁴

Il BASIC

- Nel 1964 **John Kemeny** e **Thomas Kurtz** concepiscono il **Beginner's All-purpose Symbolic Instruction Code**, meglio noto come **BASIC**.
- In realtà, fu un gruppo di studenti della Università di Dartmouth (USA), guidati da Lemeny e Kurtz, a sviluppare il compilatore, che fu reso disponibile a partire dal primo maggio 1964.
- In quegli anni, la scrittura di programmi era considerata una attività piuttosto peculiare, accessibile solo ad esperti del settore, in particolare ingegneri e matematici.
- Lo scopo del BASIC era di permettere a studenti di altre discipline di poter usare i computer e scrivere programmi relativamente complessi senza avere troppe conoscenze specifiche nel campo dell'informatica.

Il BASIC

- In effetti negli anni ‘60 l’idea di usare un computer come strumento di supporto all’insegnamento di materie scientifiche era piuttosto originale, e un linguaggio di programmazione poteva essere utile solo se esso stesso non richiedeva troppo lavoro di apprendimento.
- Nello stesso periodo poi cominciavano a diffondersi i sistemi time sharing, accessibili anche attraverso terminali telescriventi **remoti**: ciò ne allargava naturalmente il bacino di potenziali utenti.
- Il BASIC fu inizialmente pensato per la scrittura di algoritmi di calcolo matematici, in modo da renderne facile l’implementazione da parte di studenti di matematica e scienze affini.

Il BASIC

- Kemeny e Kurtz furono molto bravi a promuovere l'utilizzo del BASIC: resero disponibile il compilatore gratuitamente e ne incoraggiarono l'uso nelle scuole della loro provincia.
- Il loro impegno ebbe successo, perché negli anni l'uso del BASIC si diffuse, e un numero sempre maggiore di produttori rese il BASIC disponibile sui propri modelli di computer.
- Ma la vera esplosione del BASIC si ebbe a partire dagli anni '70, con la diffusione di quelli che al tempo erano chiamati **microcomputer**: computer dotati di microprocessore, economici e dunque alla portata di molti (più avanti saranno chiamati **Personal Computer**, anche se bisognerebbe fare qualche piccolo distinguo)

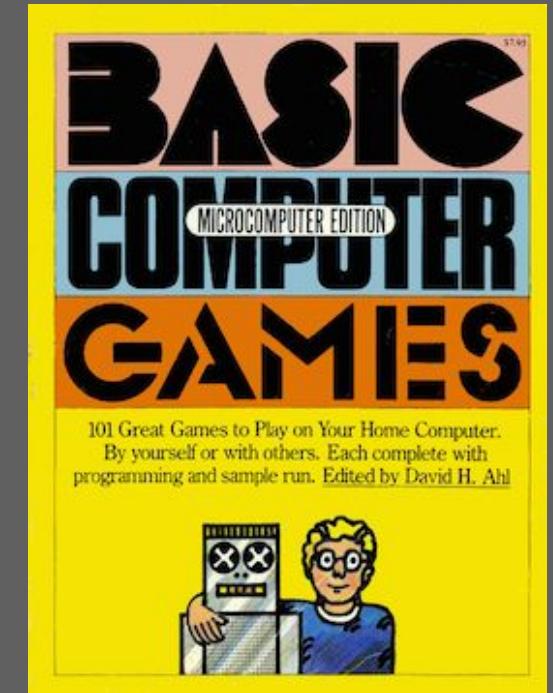
Il BASIC

- Dalla seconda metà degli anni ‘70, in pratica qualsiasi personal computer forniva un compilatore o un interprete BASIC.
- In alcuni casi addirittura il BASIC costituiva l’ambiente di lavoro del PC: l’interprete dei comandi era un interprete BASIC che poteva ricevere istruzioni BASIC direttamente dal terminale (proprio come una qualsiasi shell Unix, che è contemporaneamente un interprete di comandi e un linguaggio di programmazione)
- Ad esempio funzionavano così l'**Apple II** e il **Commodore PET 2001** (in figura), prodotti a partire dal 1977 e tra i più diffusi di quegli anni.



Il BASIC

- La fortuna del BASIC fu che era **sufficientemente leggero** da poter girare su computer di potenza limitata, ma allo stesso tempo **sufficientemente ad alto livello e semplice** da poter essere usato da chiunque anche senza una fase di addestramento e studio specifici.
- Il BASIC cominciò a diffondersi anche come strumento didattico nelle scuole, mentre molte riviste di informatica degli anni ‘70 e ‘80 pubblicavano codice BASIC per videogiochi e applicazioni varie che poteva essere copiato e fatto girare sul proprio PC *istantaneamente*.
- Il **BASIC COMPUTER GAMES** di **David Ahl**, del 1978, fu il primo libro di informatica a superare il milione di copie vendute.



Il BASIC

- Negli anni ‘70 e ‘80 il BASIC contribuì in modo determinante alla diffusione tra il grande pubblico di una cultura informatica.
- C’è da chiedersi se fosse vera cultura, dato che il BASIC è sempre stato guardato con una certa aria di sufficienza da molti. È già del 1975 questo famoso giudizio *tranchant* di Edsger Dijkstra:

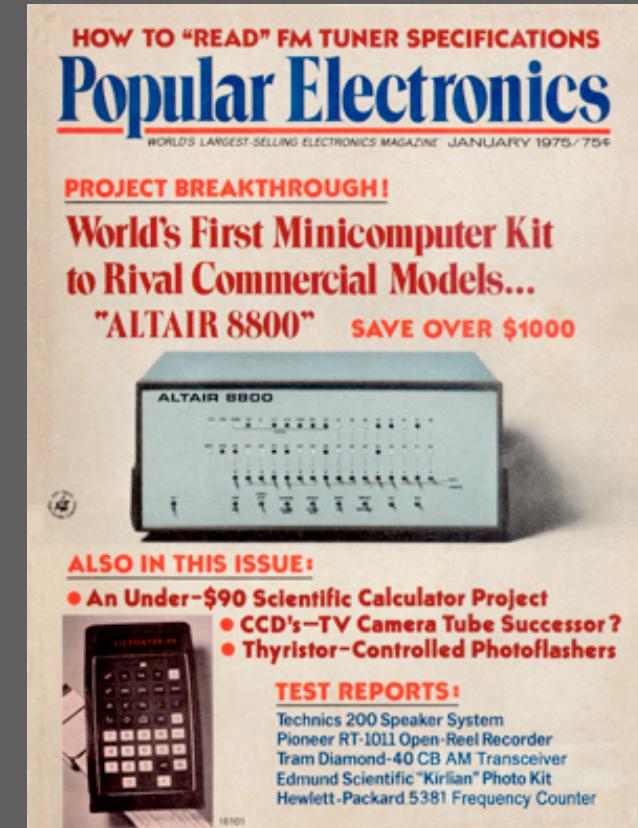
“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration”
- L'affermazione si trova in un articolo dal titolo significativo: “*How do we tell truths that might hurt?*” dove Dijkstra elenca una serie di verità scomode dell'informatica, tipo: “*the use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence*⁸⁹”.

Il BASIC

- A onor del vero, negli anni il BASIC fu sottoposto a varie migliorie, soprattutto in seguito all'ingresso nel mercato dei PC della IBM, che ovviamente offrivano un ambiente BASIC *built-in*.
- Lungo gli anni ‘80 quindi al BASIC furono aggiunti costrutti per la programmazione strutturata e per una gestione più sofisticata delle subroutine e delle variabili locali.
- Il culmine di questo processo si ebbe probabilmente con il **Visual BASIC**, introdotto dalla Microsoft nel 1991, sebbene il linguaggio si trovasse ormai a competere con linguaggi molto più avanzati (ad esempio C e C++) a disposizione dei programmatore professionali.

Nota a Margine: Bill Gates' BASIC

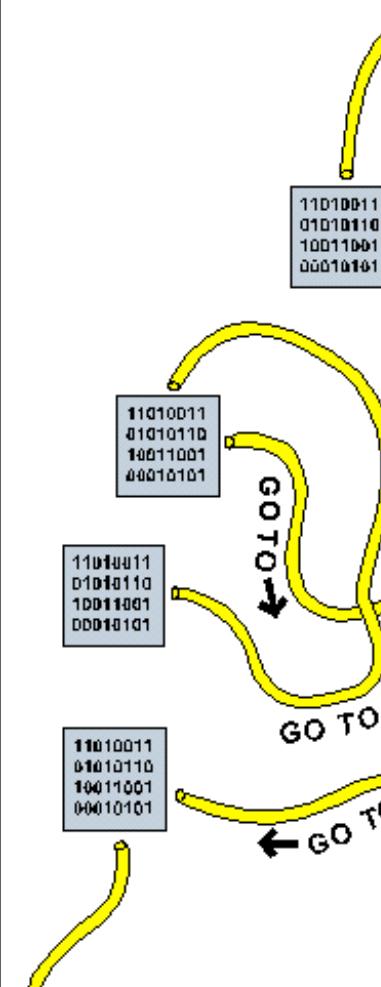
- Nel 1974 la **MITS**, una azienda che produceva calcolatrici elettroniche, progetta un *Home Computer*, l'**Altair 8800** (con CPU Intel 8080)
- Recensito sulla rivista *Popular Electronics* nel novembre del 1974 l'Altair ha un enorme successo di pubblico e vendite, e diviene il primo PC a grande diffusione della storia.
- **Bill Gates e Paul Allen** (co-fondatori della Microsoft) leggono dell'Altair e nel marzo 1975 contattano la MITS per offrire un interprete BASIC (che in realtà dovevano ancora sviluppare!)
- Il 4 aprile 1975 Allen e Gates fondano la **Micro-Soft** (notare il trattino) il cui primo prodotto sarà l'**Altair BASIC**.



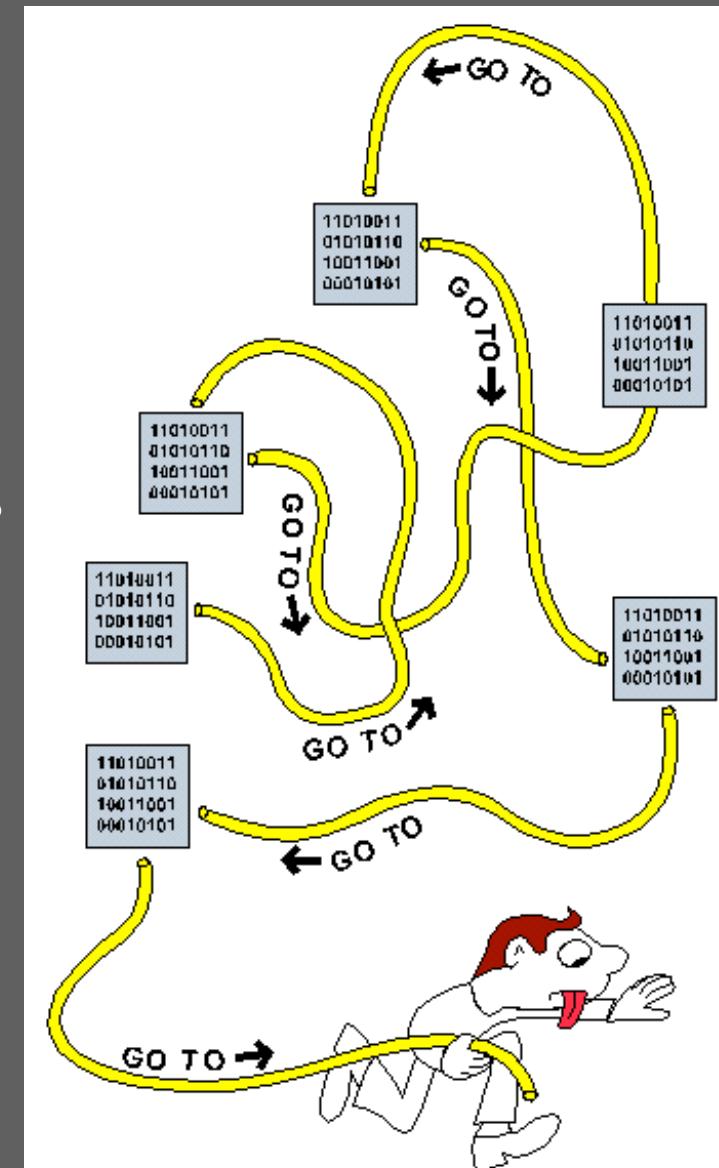
Nota a margine: Steve Wozniak's BASIC

- All'inizio degli anni '70 **Steve Wozniak** si era impraticchito col BASIC usandolo principalmente per scrivere video games.
- Venuto a sapere dell'Altair BASIC, decise di sviluppare qualcosa di simile per la CPU “MOS Technology 6502”, che scrisse direttamente in codice macchina (e senza praticamente avere nessun background su come si progetta un linguaggio di programmazione!).
- Il **GAME BASIC** di Wozniak fu poi ridenominato **Integer BASIC**, dato che implementava solo la matematica intera, più che sufficiente per scrivere giochi.
- Integer BASIC fu adottato come interprete e ambiente di lavoro nell'Apple I e II, fino a che la Apple acquistò dalla Microsoft una versione completa del BASIC chiamandola **Applesoft BASIC** e disponibile a partire dal 1979.

Spaghetti code

- Sebbene l'ALGOL contenesse primitive di controllo del flusso di computazione strutturate, la presenza del GOTO permetteva comunque uno stile di **programmazione non strutturata**.
 - Semplificando, in un programma non strutturato la computazione procede attraverso test e un uso estensivo di GOTO, anziché subroutine, blocchi di istruzioni adeguatamente nidificati, e while.
 - Il codice non strutturato e quindi con un flusso della computazione ingarbugliato viene normalmente detto **Spaghetti Code**. (il termine però risale alla seconda metà degli anni 70)

The diagram illustrates the concept of spaghetti code. It features four separate boxes, each containing a different sequence of binary digits (1s and 0s). From each box, a yellow curved arrow labeled 'GO TO' points to another part of the code, creating a complex web of connections that resemble spaghetti. A cartoon character with a red bow tie and a mustache is shown at the bottom right, holding a piece of spaghetti, which serves as a metaphor for the tangled flow of control in non-structured programs.



Spaghetti code

- Linguaggi come BASIC e FORTRAN non favorivano certo uno stile di programmazione strutturato che oggi noi invece diamo per scontato.
- Inoltre questi linguaggi erano spesso usati da persone senza uno specifico background né una formazione informatica, le quali dunque a maggior ragione tendevano a scrivere spaghetti code.

Unstructured BASIC program:

```
10  i = 0  
20  i = i + 1  
30  PRINT i; " square = "; i * i  
40  IF i >= 10 THEN GOTO 60  
50  GOTO 20  
60  PRINT "Program Done."  
70  END
```

Structured BASIC program:

```
10  FOR i = 1 TO 10  
20  PRINT i; " square = "; i * i  
30  NEXT i  
40  PRINT "Program Done."  
50  END
```

Il teorema di Böhm-Jacopini

- Nel 1966 **Corrado Böhm** e **Giuseppe Jacopini** dimostrano un teorema che diventa subito popolarissimo nel mondo dell'informatica, e in special modo tra i sostenitori della programmazione strutturata.
- Il teorema ha diverse formulazioni equivalenti, ma sostanzialmente asserisce che qualsiasi funzione computabile può essere calcolata da un programma costituito esclusivamente da una combinazione di:
- **Sequenze** di istruzioni eseguite una dopo l'altra;
- Istruzioni di **selezione** (del tipo *if then else*)
- Istruzioni di **iterazione** (del tipo *while do*)

Contro il GOTO

- Il teorema di Böhm-Jacopini segna l'inizio di un acceso dibattito a favore (ma anche contro) la programmazione strutturata.
- In sostanza il teorema mostra che si possono scrivere programmi senza usare il GOTO, dunque senza saltare dentro e fuori a blocchi di istruzioni (ad esempio il corpo dei cicli) in modo disordinato (il che ovviamente rende i programmi meno comprensibili)
- Nel 1968 Dijkstra critica aspramente l'uso del GOTO in una famosa lettera alle **Communication of the ACM** intitolata:
“Go To Statement considered Harmful”
- Per Dijkstra il GOTO dovrebbe essere abolito nei programmi scritti con linguaggi ad alto livello perché ne complica l'analisi e la verifica di correttezza.

Contro il GOTO

- La lettera di Dijkstra non fa che alimentare il dibattito. In effetti tutti sono d'accordo che un uso eccessivo del GOTO sia chiaramente negativo, ma per molti il GOTO è comunque utile. Ad esempio:
- In un famoso articolo del 1974, **Structured Programming with Go To Statement**, Donald Knuth mostra che in alcuni casi il GOTO è il costrutto ideale da usare.
- Nel loro mitico manuale sul C del 1978 **Brian Kernighan** e **Dennis Ritchie** osservano che il GOTO è fondamentale per gestire le condizioni di errore e per uscire velocemente dai cicli nidificati.
- Linguaggi moderni come **Java** e **Python** non usano il GOTO (ma già il **BLISS**, nel 1970, un predecessore del C, non usava il GOTO).

Nota di colore: the *Italian code*

- Nella terminologia informatica la cucina italiana non è presente solo con lo Spaghetti code, ma anche con altri due termini, sebbene molto meno usati e noti:
- **Ravioli code**: programmi composti da diversi componenti software ognuno dei quali è strettamente encapsulato e fortemente separato dagli altri componenti.
- **Lasagna code**: programmi composti da diversi strati di codice in cui ogni strato interagisce con quelli adiacenti mediante protocolli e interfacce definite in modo preciso e rigoroso.
- Queste espressioni fanno poi esse stesse parte di una espressione che le raggruppa tutte e tre sotto il comune denominatore di **Programming Pasta...**

Il Simula I e il Simula 67

- Il Simula merita un posto d'onore nella storia dell'informatica perché è l'antesignano dei linguaggi di programmazione object oriented.
- Sviluppati lungo gli anni '60 al **Norwegian Computing Center** di Oslo da **Ole-Johan Dahl** e **Kristeen Nygaard**, sono tecnicamente dei soprainsiemi dell'ALGOL, anche se furono pensati specificamente per compiere **simulazioni** (da cui il nome) in vari domini.
- Dahl e Nygaard svilupparono il Simula I tra il 1962 e il 1965 su un Computer UNIVAC, basandosi sul compilatore dell'ALGOL 60.
- Negli anni successivi, grazie anche a corsi tenuti da Dahl e Nygaard, il Simula I si diffuse un po' ovunque (Russia Inclusa), e fu implementato su diversi sistemi.

Il Simula I e il Simula 67

- In effetti il Simula I si dimostrò particolarmente adatto per modellare il funzionamento di sistemi ad eventi discreti come protocolli di comunicazione e modelli di processi dinamici di qualsiasi tipo, siano essi industriali o sociali.
- Nel 1966 Dahl e Nygaard decidono di estendere il SIMULA per renderlo il più possibile un linguaggio *general purpose*.
- In particolare, osservano che i “processi” così ben modellati dal Simula I, condividono di solito un certo numero di proprietà, sia in termini di attributi (*dei processi*) che di azioni (*sui processi*).
- Si chiesero dunque se ci fosse modo di modellare le proprietà comuni di processi coinvolti in modelli di simulazioni distinte.

Il Simula 67 e l'origine del termine “classe”

- Dahl e Nygaard si ispirarono al concetto di *record class* proposto da Tony Hoare nel 1965 (concetto poi inglobato nell’ALGOL).
- Nella sua proposta Hoare usava l’espressione **record class** per indicare il generico concetto che si vuole modellare (ad esempio il concetto di *tavolo*), mentre i record sono le istanze della classe *tavolo*.
- Hoare osservava di essere stato a sua volta ispirato dalla “*McCarthy’s proposed class union declaration*” anche se non si è mai riuscito a chiarire a cosa si riferisse...
- E nel dicembre 1966 finalmente Dahl e Nygaard riescono a definire un modello di **classi e sottoclassi** sufficientemente chiaro e facile da implementare mediante un meccanismo a prefissi che permettesse di definire una gerarchia di classi.

Il Simula 67

- Naturalmente, le istanze di una classe erano gli **oggetti**, e l'operazione **new** creava una nuova istanza di una data classe: un nuovo oggetto con il proprio set specifico di variabili dichiarate per quella classe.
- La dichiarazione di una nuova classe nel Simula 67 era molto simile alla dichiarazione di una nuova procedura nell'ALGOL:

```
class Point (x,y); real x, y;  
begin  
    boolean procedure equals(p); ref(Point) p;  
        if p == none then  
            equals := abs(x - p.x) + abs(y - p.y) < 0.00001;  
    real procedure distance(p); ref(Points) p;  
        if p == none then error else  
            distance := sqrt((x - p.x)**2 + (y - p.y)**2);  
    end **** Point ****  
p := new Point(1.0, 2.5);
```

Il Simula 67

- La **sottoclasse** C2 di una classe C1 era definita per **ereditarietà** rispetto a C1 (che ovviamente era la **sopraclass** di C2), e per creare un nuovo oggetto della classe C2 era necessario creare prima un oggetto di classe C1 (un po' come se in ALGOL la procedura C2 fosse dichiarata e chiamata all'interno di C1):

```
Point class ColorPt(c); color c;      ! A colored point is a subclass of the Point class
begin
  boolean procedure equals(q); ref(ColorPt) q;
  ...
  ...
end **** ColorPt ****
ref(Point) p;                      ! Class reference variables
ref(ColorPt) cp;
p :- new Point(2.7, 4.2);
cp :- new ColorPt(3.6, 4.9, red);   ! Inlcude parent class parameter
cp.c := green;                      ! Changes color of cp
```

L'implementazione del Simula 67

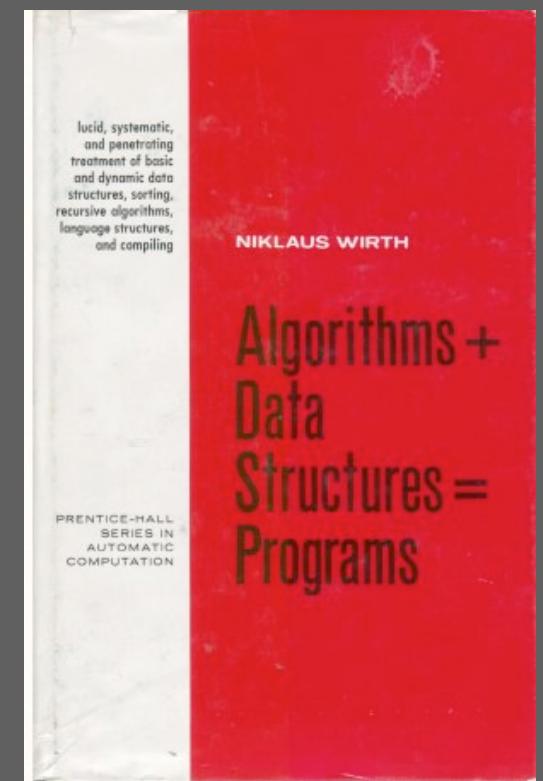
- L'idea di oggetto nasceva dal fatto che dopo la chiamata di una procedura P era possibile lasciare sullo stack il record di attivazione di (quella chiamata) di P, e restituire un puntatore al record. Dunque:
- In Simula una **classe** è implementata come una procedura che restituisce *il puntatore al suo record di attivazione*, e
- Un **oggetto** è un *record di attivazione* prodotto dalla chiamata di una classe (che, appunto, è un tipo speciale di procedura), ed è anche detto *istanza di una classe*.
- Per implementare questo meccanismo era necessario introdurre i puntatori (assenti nell'ALGOL), chiamati **ref** nel Simula.
- Ed era necessario implementare un meccanismo di **garbage collection**⁹⁵ per i record di attivazione non più attivi.

Il Simula I e il Simula 67

- Dunque con il Simula I e il Simula 67 vengono introdotti alcuni concetti fondamentali di **data abstraction**, anche se l'importanza del paradigma **object oriented** verrà riconosciuto solo più avanti.
- Col Simula nascono i concetti (e i termini) di **classe** e **sottoclasse**, **tipo** e **sottotipo**, gli **oggetti**, e **l'ereditarietà** (non l'ereditarietà multipla).
- Ma il Simula va ricordato anche per la sua capacità di **modellare la concorrenza**: ossia simulazioni di eventi complessi composti da parti indipendenti ciascuna delle quali definita da una successione di eventi.
- Le sequenze indipendenti erano rappresentate da processi indipendenti, e all'esecuzione il programma principale alternava l'esecuzione di ciascun processo in modo da farli progredire tutti simulandone il loro procedere in parallelo.

Gli anni ‘70: il Pascal

- Il **Pascal** (dal nome del filosofo **Blaise Pascal**) ha dominato la scena dei linguaggi di programmazione degli anni ‘70. Si può considerare il diretto successore dell’ALGOL, del quale migliora vari aspetti.
- Progettato intorno alla fine degli anni ‘60 e presentato ufficialmente nel 1971 da **Niklaus Wirth** (Svizzera) il Pascal aveva nientemeno che l’ambizione di essere **efficiente da compilare ed eseguire**, permettere la **scrittura di programmi ben strutturati e ben organizzati**, e servire come strumento per **insegnare la programmazione**.
- E l’obiettivo fu ampiamente raggiunto.



Il Pascal

- Come abbiamo visto, l'ALGOL era stato progettato da un comitato internazionale (del quale Wirth faceva parte), e doveva essere un linguaggio indipendente dalla piattaforma su cui usarlo.
- Ovviamente ciò dava maggiore libertà sulle caratteristiche da dare al linguaggio, ma lo rendeva anche più difficile da implementare adattandolo alle diverse piattaforme hardware. Scrivere un compilatore per l'ALGOL era difficile, e i programmi scritti in ALGOL erano spesso inefficienti.
- In più l'ALGOL non implementava il tipo CHAR né i puntatori, e dunque ingegneri e programmatore tendevano di gran lunga a preferire il FORTRAN, e l'ALGOL rimaneva in uso principalmente per la descrizione degli algoritmi.

Il Pascal

- Dunque il progetto del Pascal fu guidato tanto dalle caratteristiche dell'ALGOL quanto dal tentativo di correggerne i difetti. Anche il Simula 67 e alcune varianti dell'ALGOL ispirarono il lavoro di Wirth.
- Il Pascal incoraggiava una **programmazione strutturata e modulare** utilizzando i costrutti già usati nell'ALGOL, la definizione di **procedure e funzioni anche nidificate**, e l'uso di **strutture dati dinamiche**. Usava i **puntatori**, il **call by value** e il **call by reference**.
- Ma in più aveva un **sistema di tipi molto ricco** (era possibile definire nuovi tipi) **ma anche molto rigido**, con un meccanismo di **type checking in fase di compilazione** che cercava di evitare possibili errori nel modo di usare e combinare fra loro variabili di tipo diverso.⁹⁹

Il sistema di tipi del Pascal

- **Tipi di dati predefiniti:** *integer, real, boolean, char, string.* A partire dai quali si possono costruire i tipi complessi: *array e record.*
- La **conversione fra tipi** è ammessa solo usando specifiche funzioni built-in. Ad esempio:

`y := round(x)` converte la variabile reale x nella variabile intera y

- **Subrange types:** sottosinsieme definito dall'utente dei tipi di dati *ordinali* (quindi non del tipo real):

var

```
x: 5 .. 27;  
y: 'a' .. 'f';
```

Il sistema di tipi del Pascal

- **Set types:** il Pascal fu il primo linguaggio ad implementare il concetto di insieme:

var

```
set1: set of 5 .. 27;  
set2: set of 'a' .. 'f';
```

- Che permetteva di implementare in modo estremamente efficiente e comprensibile alcune operazioni su insiemi:

if i in [7 .. 21] then *invece di* **if (i > 6) and (i < 22) then**

if i in [3 .. 10, 15, 18, 25 .. 32] then *invece di...*

Il bubblesort in Pascal

Program bubblesort (input, output);

```
const MAX = 50;  
var i, j, size : integer;  
type myarray : array [1 .. MAX] of integer; numbers : myarray; (* a new type! *)
```

```
procedure swap( var a, b: integer );
```

```
    var temp : integer;
```

```
    begin
```

```
        temp := a; a := b; b := temp;
```

```
    end;
```

```
    begin
```

(* main program begins)

```
        readarr(size, numbers);
```

(* readarr defined somewhere else *)

```
        for i := size-1 downto 1 do
```

```
            for j := 2 to i do
```

```
                if (numbers[j-1] > numbers[j]) then swap(numbers[j-1], numbers[i]);
```

```
        printarr (size, numbers);
```

(* printarr defined somewhere else *)

```
    end.
```

Il successo del Pascal

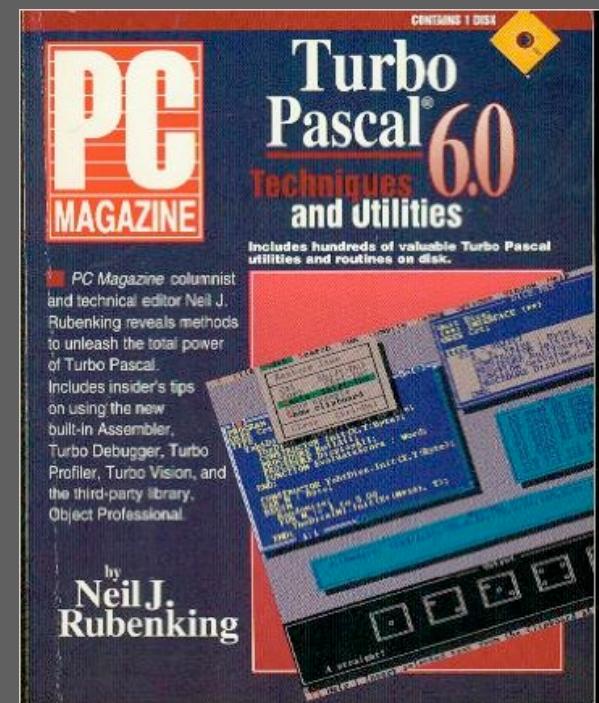
- Un programma scritto in Pascal era di solito facile da comprendere, e risultava efficiente da eseguire perché il compilatore del Pascal non generava direttamente codice macchina, ma un codice oggetto per una macchina virtuale, il **P-code**.
- Un ulteriore modulo compilatore specifico per ogni piattaforma trasformava poi il P-code in codice macchina vero e proprio. In alternativa era possibile (e ragionevolmente efficiente) usare un interprete per il P-code, senza nemmeno dover scrivere un compilatore ad hoc.
- Grazie a questa soluzione il Pascal poteva essere rapidamente installato e fatto girare su piattaforme diverse e con diversa potenza di calcolo.

Il successo del Pascal

- Verso la fine degli anni ‘70 il Pascal era ormai in uso e insegnato in tutte le università, e due eventi ne stimolarono ulteriormente la diffusione negli anni ‘80:
- L’Educational Testing Service (**ETS**), la compagnia che negli USA amministra e scrive i test di ingresso ai college americani decise di inserire un esame di Informatica e di usare il Pascal per il test.
- Ovviamente, tutti gli studenti delle superiori dovettero mettersi a studiare il Pascal, che rimase in uso nei test ETS fino al 1999, quando fu sostituito prima dal C++ e poco dopo da Java.

Il successo del Pascal

- Nel 1983 **Anders Hejlsberg** della **Borland**, allora una piccola compagnia di sviluppo software, progetta il **Turbo Pascal** per i Personal Computer IBM (Hejlsberg diverrà poi responsabile dello sviluppo software alla Microsoft).
- Il Turbo Pascal era davvero rivoluzionario: al costo di modifiche marginali allo standard del linguaggio permetteva ridottissimi tempi di compilazione e produceva eseguibili estremamente efficienti.
- Il Turbo Pascal divenne velocemente lo standard di fatto per lo sviluppo di programmi al PC. La rivista **PC Magazine** pubblicava i suoi programmi in assembler o in Turbo Pascal.



Apice e declino del Pascal

- Quando la Apple lanciò la linea dei Macintosh, scelse il Pascal come linguaggio di programmazione standard, e gli esempi di programmi forniti ai programmatori dei Macintosh erano scritti in Pascal.
- Le successive versioni del Turbo Pascal includevano sempre nuove caratteristiche: la possibilità di scomporre grossi programmi in **moduli compilabili separatamente** (caratteristica assente nel Pascal originale); una **interfaccia grafica per l'ambiente Windows**, una versione **object oriented**.
- Ma intanto era già in corso una silenziosa e discreta ma costante avanzata della diffusione del C, dell'ambiente Unix e più tardi del C++, che verso la fine degli anni ‘80 avrebbero decretato il tramonto dell’era Pascal (ci torneremo più avanti).

Il Prolog

- Al pari del LISP, sicuramente il Prolog non ha avuto neanche lontanamente la diffusione in campo commerciale e industriale degli altri linguaggi che abbiamo visto e che vedremo, e il suo uso è sempre rimasto confinato alle università e ai centri di ricerca.
- Tuttavia il Prolog ha diritto ad un suo posto nella storia dei linguaggi di programmazione in quanto iniziatore del **paradigma di programmazione logica** e alla base del progetto dei **calcolatori della quinta generazione**, poi miseramente fallito (vedremo tra poco)...
- Il **Prolog** (**P**rogrammation en **L**ogique) fece il suo esordio nel 1972, un progetto portato avanti da **Alan Colmerauer** e **Philippe Roussel** dell'Università di Marsiglia, e con i contributi teorici di **Robert Kowalsky**, dell'Università di Edinburgo.

Il Prolog

- Il Prolog è un **dimostratore automatico di teoremi** che usa come inferenza la **regola di risoluzione**, secondo i lavori teorici sviluppati da **Martin Davis** e **Hilary Putnam** (1960) e **John Robinson** (1965).
- Durante il processo di dimostrazione, alcuni valori possono venire calcolati (ossia assegnati alle variabili contenute nel programma Prolog), e ciò permette quindi di portare avanti una computazione.
- Un programma Prolog è semplicemente un elenco di clausole del tipo:

head (x_1, \dots, x_n) :- body₁(x_m, \dots, x_p), ..., body_n(x_l, \dots, x_q).

in cui è vero l'atomo di testa se sono veri tutti gli atomi del corpo.
Una clausola il cui corpo è vuoto è sempre vera.

Il Prolog

- Una cosa particolarmente elegante del Prolog è dunque che funziona come dimostratore di teoremi per verificare il valore di verità di una asserzione, e come linguaggio di programmazione per calcolare valori.
- Eccone un semplice esempio:

```
member (X, [X | _]).      // X è membro della lista la cui testa è X
member (X, [_ | Y]) :- member (X, Y).      // ???
```

```
?- member (5, [ 6, 5, 1 ]).
```

Yes

```
?- member (X, [ 6, 5, 1 ]).
```

Y = 6;

Y = 5;

Y = 1;

no.

Il bubblesort in Prolog

- Per definizione, nel Prolog tutti i programmi sono ricorsivi, il che porta a codice molto elegante e conciso. Ecco una possibile versione del bubblesort in Prolog:

```
sort (List, Sorted) :- swap (List, List1), !, sort (List1, Sorted).  
sort (Sorted, Sorted).
```

```
swap ([ X, Y | Rest ], [ Y, X | Rest]) :- X > Y.  
swap ([ Z | Rest ], [ Z | Rest1 ]) :- swap (Rest, Rest1).
```

- Il simbolo “!” è il *cut*, e in questo caso serve per dire al programma di fermarsi quando ha completato l’ordinamento.

```
?- sort ([ 3, 5, 2, 1, 6, 4 ], Sorted).
```

```
Sorted = [ 1, 2, 3, 4, 5, 6 ]
```

Il Prolog

- Il Prolog nacque per la processazione del linguaggio naturale, ma il suo uso si estese velocemente in molte altre aree di ricerca in intelligenza artificiale, soprattutto in Europa, mentre negli USA si continuò a preferire il LISP.
- Inizialmente il Prolog era interpretato, con l'interprete scritto in ALGOL, ma nel 1983 **David Warren** propose una macchina astratta e un corrispondente instruction set che servisse da “target” per implementare compilatori Prolog e rendere il linguaggio molto più efficiente.
- Il paradigma di programmazione logico ebbe influenza anche in campi diversi dall'intelligenza artificiale, ad esempio per la creazione di **database deduttivi** che estendono quelli relazionali con la capacità di compiere inferenze sulla conoscenza memorizzata.

5th Generation Computer Systems (FGCS)

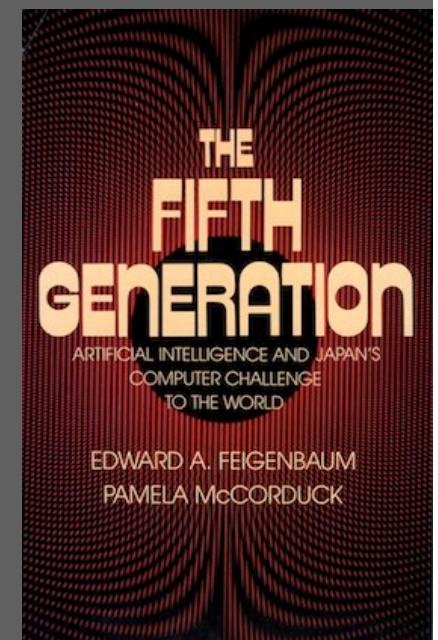
- Nel 1981 il governo giapponese lanciò un faraonico programma per la progettazione di supercomputer a sfruttamento massivo del parallelismo con interfaccia basata sul linguaggio naturale e AI.
- La “**quinta generazione**” doveva indicare un nuovo “salto quantico” rispetto alle tecnologie delle precedenti quattro: valvole termoioniche, transistor, circuiti integrati, microprocessori.
- Il progetto ruotava intorno allo sviluppo di processori specificamente orientati all’esecuzione efficiente delle clausole dei programmi logici, che hanno una natura intrinsecamente parallela.
- Nel 1982 poi, **Ehud Shapiro** aveva inventato il **Concurrent Prolog**, che integrava programmazione logica e programmazione concorrente.¹¹²

5th Generation Computer Systems (FGCS)

- Le macchine della quinta generazione dovevano eseguire operazioni logiche, e dunque le loro performance sarebbero state valutate in **Logic Inferences Per Seconds** o **LIPS**.
- L'obiettivo era di costruire macchine in grado di eseguire fin a un Giga LIPS, quando le macchine esistenti arrivavano al massimo a 100 K-LIPS.
- Negli anni '80 l'industria giapponese aveva ormai acquisito capacità tecnologiche altamente raffinate, e si era guadagnata una fama di invincibilità sullo scenario mondiale.
- Dunque il progetto FGCS ebbe una enorme risonanza in tutto il mondo, e destò non poca apprensione nei produttori di processori, computer e software (e nei relativi governi), e progetti simili vennero velocemente approntati negli USA e in Europa.

5th Generation Computer Systems (FGCS)

- Ma dopo dieci anni, e investimenti di circa un miliardo di dollari (attuali), il progetto si rivelò un fallimento, per varie ragioni.
- Intanto, negli anni '80 si verificò il passaggio dalle architetture CISC alle RISC, il che produsse processori molto più veloci, e con prestazioni ben superiori alle macchine parallele del progetto FGCS.
- Lo sfruttamento esplicito del parallelismo era un obiettivo troppo avanti per quei tempi, e infatti fu messo da parte e rispolverato solo verso la fine degli anni '90.
- Infine, il paradigma di programmazione logica si rivelò una scelta errata, risultando difficile, se non impossibile, da implementare in maniera efficiente.



Lo Smalltalk

- Lo Smalltalk fu concepito da **Alan Kay** tra la fine degli anni ‘60 e l’inizio degli anni ‘70 allo **Xerox PARC**.
- Abbiamo già incontrato lo **Xerox Palo Alto Research Center**, che dagli anni ‘60 fino agli anni ‘90 è stato una vera e propria fucina di idee originali e innovative, tra cui le *Graphical User Interfaces* e gli editor “*WYSIWYG*”, la rete *Ethernet*, il concetto di *Personal Computer*, e le *stampanti laser*.
- L’ambiente di lavoro allo Xerox PARC era informale e creativo, molti anni prima di Google...

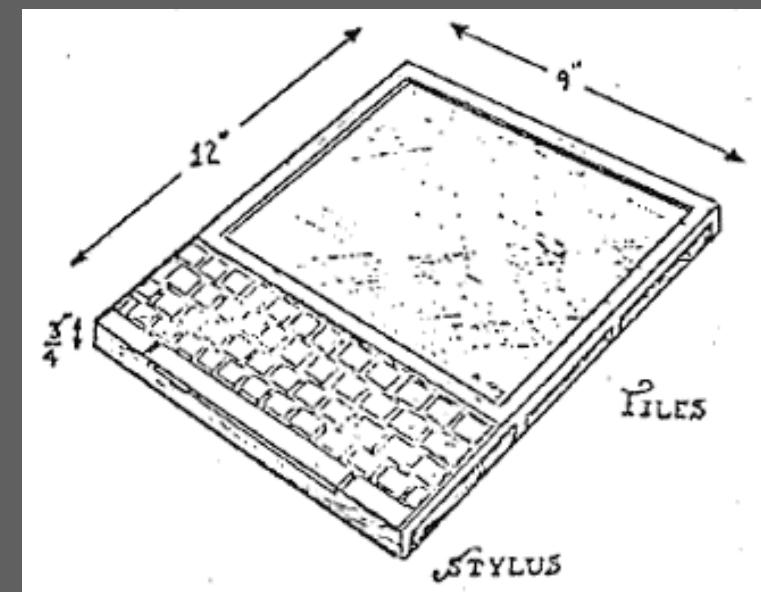


Lo Smalltalk

- E al pari di *Vannevar Bush*, *Douglas Engelbart* e *Steve Jobs*, Alan Kay è stato uno dei grandi visionari dell’Informatica, e un pensatore con idee originali, particolarmente interessato allo sviluppo di metodologie innovative per l’apprendimento, soprattutto nei bambini.
- Il suo discorso in occasione del ricevimento del **Turing Award** (il premio Nobel dell’Informatica) nel 2004 si intitolava: “*First Courses in Computing Should be Child’s Play*” e si chiudeva con l’esortazione a insegnare l’informatica come una meravigliosa forma d’arte.
- Nel 1972 descrive l’idea (concepita già nel 1968) del **Dynabook**: “*A Personal Computer for Children of All Ages*”, un dispositivo portatile pensato per favorire l’apprendimento nei bambini.

Lo Smalltalk

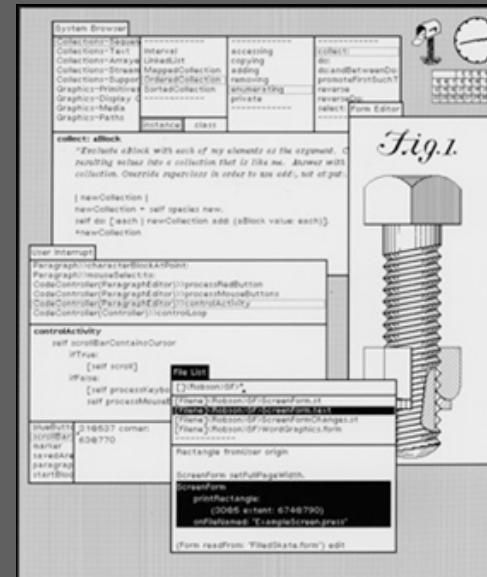
- Il Dynabook era in tutto e per tutto ciò che adesso chiamiamo tablet PC. Ma all'inizio degli anni '70 un tale dispositivo era persino difficile da immaginare, viste le dimensioni dei computer di allora.
- Nell'idea di Kay, il Dynabook doveva essere un piccolo computer portatile capace di immagazzinare informazioni personali e di far girare applicazioni diverse. Doveva avere memoria sufficiente per memorizzare numeri telefonici, dati di lavoro e una copia di libri da leggere.
- L'idea di computer miniaturizzati al punto tale da stare in una tasca e di usarli per tutta una serie di scopi personali era assolutamente rivoluzionaria per il suo tempo...



Il Dynabook nell'articolo del 1972

Lo Smalltalk

- Lo Smalltalk non era altro che l’interfaccia del sistema operativo e il linguaggio di programmazione del Dynabook: tutte le interazioni con il Dynabook dovevano avvenire attraverso lo Smalltalk.
- Lo sviluppo dello Smalltalk proseguì in maniera indipendente dal Dynabook, e passò attraverso diverse versioni, dallo Smalltalk-72 allo Smalltalk-80, grazie principalmente a **Dan Ingalls** e **Adele Goldberg**.
- Il primo ambiente di programmazione ad interfaccia grafica fu sviluppato per lo Smalltalk, e fu implementato sullo Xerox Alto, il PC sperimentale della Xerox di cui abbiamo già parlato nella storia delle architetture (e a cui quelli della Apple si ispirarono per il Macintosh).

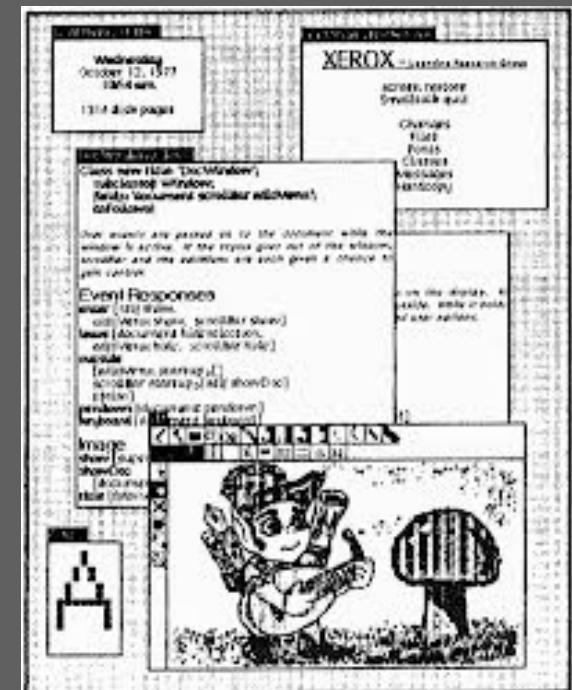


Lo Smalltalk

- Sebbene, in quanto linguaggio object oriented, si ispirasse al Simula, lo Smalltalk non ne rappresenta una modifica o estensione.
- Fu infatti sviluppato da zero in modo completamente indipendente, con una sua terminologia e sintassi originali, e l' elenco degli aspetti innovativi dello Smaltalk è lungo.
- **Nello Smalltalk esistono solo oggetti:** anche **le classi sono oggetti**. Dunque, **tutte le operazioni sono messaggi inviati agli oggetti**.
- **I programmi sono costituiti solo da classi e oggetti.**
- Gli oggetti hanno **dati privati** e comunicano con l'esterno solo attraverso i **metodi**: codice per eseguire operazioni.
- **Un sistema di tipi dinamico e flessibile** permetteva di adattare facilmente vecchi programmi a nuove applicazioni.¹¹⁹

Lo Smalltalk

- Tutta la terminologia usata negli ambienti di programmazione ad oggetti, e il significato dei vari termini, si deve sostanzialmente allo Smalltalk assai più che al Simula, nel quale i concetti del paradigma object oriented sono ancora imprecisi e parzialmente nascosti.
- Con lo Smalltalk nasceva anche l'idea di un editor specificamente progettato per quel linguaggio. Usando l'editor, le classi potevano essere definite semplicemente riempiendo una tavola suddivisa nei vari campi predefiniti (*class name, super class, class methods, etc, ...*)
- La GUI dello Smalltalk già nel 1976 includeva, oltre all'editor delle classi, anche un editor per semplici illustrazioni, un word processor, e un editor di font.



Nota a margine: la visione di Alan Kay

- “**The Early History of Smalltalk**” dice molto della filosofia alla base del lavoro di Alan Kay (l’articolo si trova facilmente in rete).
- Tra l’altro Kay nota come i linguaggi si dividano in quelli prodotti da comitati di programma e quelli frutto della visione di singoli individui. Questi ultimi creano uno stile di programmazione nuovo, mentre i primi sono solo un agglomerato di caratteristiche (spesso preesistenti).
- Kay osserva anche che, poiché in Smalltalk tutto è un oggetto, l'*idea di oggetto è ricorsiva*, un po’ come le idee platoniche che includono la “*idea di un’idea*”. (tra l’altro un concetto ricorrente, in informatica. Vi viene qualche esempio?)
- Il punto di vista di Kay è anche riassunto bene dalla citazione che trovate sulla home page della didattica del Dipartimento.

ADA: il linguaggio definitivo

- Il linguaggio ADA merita un posto nella storia dell'informatica perché costituisce il più faraonico progetto di sviluppo mai affrontato per dare alla luce un nuovo linguaggio di programmazione.
- Il progetto parte nel 1974, quando al **Department of Defense (DoD)** degli USA ci si rende conto che i diversi progetti in corso usano più di 450 diversi linguaggi di programmazione!
- Ciò naturalmente dava origine a enormi problemi di comunicazione e interfaccia tra progetti diversi, e minimizzava le possibilità di riutilizzo del software prodotto all'interno di ciascun progetto.
- I vertici dell'esercito, della marina e dell'aviazione americani decisero allora di avviare lo sviluppo di un linguaggio di programmazione specificamente adatto per i sistemi embedded. Al progetto partecipò anche il Ministero della Difesa Britannico.
122

ADA: il linguaggio definitivo

- Dopo circa tre anni il gruppo di lavoro aveva prodotto una specifica completa del nuovo linguaggio di programmazione.
- Quattro ditte contraenti vennero scelte per proporre ciascuna una diversa implementazione delle specifiche, e alla fine venne giudicato come migliore il progetto di **Jean Ichbiah** della **Honeywell-Bull**.
- Reso pubblico, il progetto ricevette più di 500 report da 15 diverse nazioni con commenti e suggerimenti che portarono alla versione definitiva del linguaggio.
- Al nuovo linguaggio venne dato il nome di **ADA**, in onore di Ada Lovelace. Il manuale in uso alle forze armate venne approvato il 10 dicembre del 1980, data di nascita di Ada, e gli venne assegnato il codice MIL-STD-1815, anno di nascita di Ada.



ADA: il linguaggio definitivo

- Per ovvie ragioni, il progetto ADA fece molto rumore nella comunità scientifica dell'informatica, in molti si aspettavano che ADA fosse destinato a divenire il linguaggio di riferimento.
- Il suo creatore, Jean Ichbiah, prevedeva che nel giro di dieci anni sarebbero sopravvissuti solo due linguaggi: ADA e il LISP. D'altra parte vi erano anche dei detrattori: ad esempio, *Tony Hoare*, nel suo discorso in occasione del *Turing Award* del 1980 criticò l'ADA per essere troppo complesso, e dunque inaffidabile.
- In effetti, i primi compilatori ADA erano lenti, e producevano codice inefficiente, cosa che comunque migliorò col tempo. Nel 1991 il DoD obbligò all'uso dell'ADA per lo sviluppo di tutto il software usato al DoD stesso, vincolo che durò fino al 1997.

ADA: il linguaggio definitivo

- L'ADA derivava le sue caratteristiche principali dall'ALGOL (**per la struttura generale a la sintassi**), dal Pascal (**per il sistema di tipi**) e dal Simula (per la gestione di **concorrenza** e **parallelismo**).
- Specifica caratteristica dell'ADA era la capacità di gestire grandi progetti software, facilitando lo sviluppo in parallelo dei diversi pacchetti di cui era composto il progetto. L'ADA non era tuttavia orientato agli oggetti, che vennero introdotti nella versione 1995.
- Molta attenzione era riservata alla scrittura di programmi liberi da errori, cosa fondamentale in campo militare, come pure nei software di gestione del traffico aereo e ferroviario (ad esempio, al posto di semplici *end* si usavano *end if* ed *end loop*; al posto di \parallel e $\&\&$ si usava *or* e *and*, e così via)

ADA: il linguaggio definitivo

- Nonostante, *o più probabilmente forse proprio a causa di*, tutte le caratteristiche dell'ADA, questo linguaggio non ha mai raggiunto la diffusione e la popolarità che ci si sarebbe aspettati date le sue origini.
- Se nel 1985 ADA era stimato essere il terzo linguaggio più usato (con l'aspettativa di divenire il primo in breve tempo), nel 2015 si posiziona intorno alla trentesima posizione della classifica (riparleremo più avanti di queste classifiche).
- Indubbiamente però, l'ADA, soprattutto dopo l'introduzione della versione object oriented, è davvero adatto allo sviluppo di grandi sistemi software e dove sicurezza e affidabilità sono fondamentali. La versione ADA del 2012 mette particolare attenzione a queste caratteristiche.

Il bubblesort in ADA

```
procedure Bubble_Sort (A : in out Arr) is
    Finished : Boolean;
    Temp : Element;

begin
loop
    Finished := True;
    for J in A'First .. Index'Pred (A'Last) loop
        if A (Index'Succ (J)) < A (J) then          -- “<“ must be defined elsewhere
            Finished := False;
            Temp := A (Index'Succ (J));
            A (Index'Succ (J)) := A (J);
            A (J) := Temp;
        end if;
    end loop;                                     -- end of construct is explicit
    exit when Finished;
end loop;
end Bubble_Sort;                                -- end of procedure is explicit
```

Il linguaggio C

- Se ne si considera la longevità e la diffusione, il C è probabilmente il linguaggio di programmazione di maggior successo nella storia dell'informatica, ancor più se si contano anche le varianti a cui ha dato luogo senza per questo scomparire.
- Discendenti diretti del C sono linguaggi come il C++, il C#, e l'Objective-C, ma anche Java e Javascript e altri linguaggi derivano molta della loro sintassi dal C. Anche la C-shell e la TCSH hanno una sintassi ispirata direttamente al C.
- L'elevata **portabilità** ed **efficienza** del C ne hanno decretato il successo in molti campi di applicazione, tra i quali l'implementazione di **sistemi operativi** e di **applicazioni embedded**. Spesso poi, **compilatori e interpreti** di altri linguaggi sono scritti in C.

Il linguaggio C

- La storia del C comincia a metà degli anni ‘60 quando i **Bell Labs**, insieme al **MIT** e alla **General Electric** iniziano il progetto di un nuovo sistema operativo, il MULTICS (di cui ripareremo).
- I Bell Labs furono fondati nel XIX secolo da **Graham Bell** col nome di **Volta Laboratory** (Bell usò i soldi del **Volta Prize**, assegnatogli per l’invenzione del telefono).
- Passati di proprietà più volte (ora appartengono alla **Nokia**), i “Labs” sono sempre stati capaci di sforzare ricerca di altissima qualità: otto premi Nobel sono stati assegnati per ricerche condotte nei Bell Labs.
- Tra l’altro, ai Bell Labs hanno avuto luogo scoperte e innovazioni come *la natura ondulatoria della materia*, *l’eco del Big Bang*, il *transistor*, la *teoria dell’informazione*, il *laser*, lo *Unix*, il *C* e il C^{++} .

Il linguaggio C

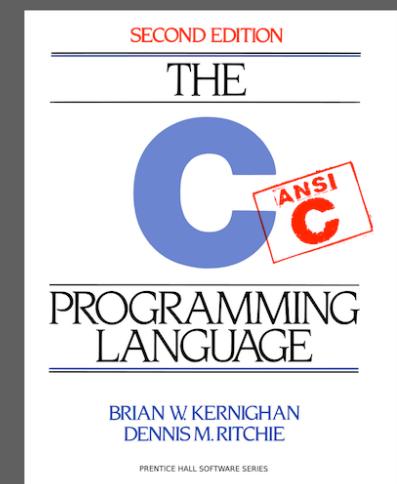
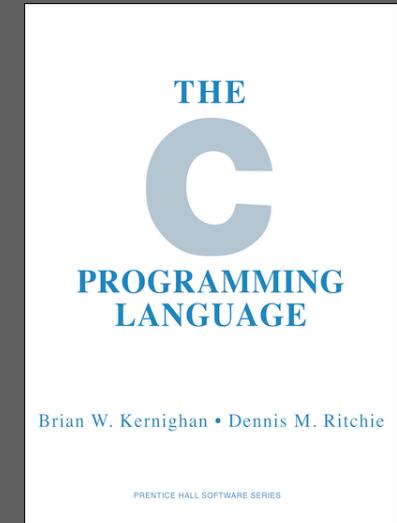
- Dennis Ritchie, Ken Thompson e Brian Kernighan erano tra i ricercatori dei Bell Labs coinvolti nel progetto MULTICS, da cui però la AT&T (la American Telegraph and Telephone Company, allora proprietaria dei Bell Labs) decise di ritirarsi nel 1969.
- Thompson stava scrivendo il codice del file system del MULTICS, e insieme ad altri colleghi continuò a lavorarci e a espanderlo fino a sviluppare un sistema operativo completo, che Kernighan suggerì di chiamare Unix come scherzo nei confronti del MULTICS.
- Il sistema era scritto nell'assembler del PDP-7, ma nel 1969 Thompson e Ritchie avevano sviluppato per lo Unix anche un nuovo linguaggio ad alto livello che avevano chiamato “B”, e il relativo interprete.

Il linguaggio C

- Il B era a sua volta derivato dal **BCPL**, un linguaggio sviluppato nel 1966 che derivava (come tanti altri) dall'ALGOL. BCPL stava per **Basic Combined Programming Language**, ma qualche anno dopo venne “rinominato” il **Before C Programming Language**...
- Il BCPL è ormai dimenticato, ma è a lui che si deve la comparsa di una notazione sopravvissuta fino ad oggi: i **blocchi di codice erano delimitati dalle parentesi graffe**. In realtà, sulle tastiere limitate degli anni ‘60, spesso le graffe dovevano essere sostituite da \$(e \$)...
• Thompson trasformò il BCPL nel B principalmente per renderlo sintatticamente più conciso, e introdusse alcune notazioni familiari a chi usa il C: **x++** e **++x**, **x += y** e **x =+ y**, **=** per l’assegnamento e **==** per il test di uguaglianza (**||** e **&&** vennero introdotti poi nel C).¹³¹

Il linguaggio C

- Il B era un linguaggio non tipato, e tra il 1971 e il 1972 Dennis Ritchie lo modificò introducendo un sistema di tipi e scrivendo un compilatore per il nuovo linguaggio, che chiamò prima **NB** (**New B**) e poco dopo semplicemente, **C**.
- Il manuale del C, chiaro e conciso, è stato spesso indicato come modello di scrittura tecnica. Infarcito di ottimi suggerimenti di buona programmazione, ha formato intere generazioni di programmatori.
- Il secondo autore, Brian Kernighan, ha sempre dichiarato di non aver partecipato allo sviluppo del C, ma è a lui che si deve l'esempio di “primo programma” in C, quel “hello world” ormai divenuto paradigmatico in quasi tutti i manuali di programmazione.



Gli anni ‘80: l’ascesa del C

- Per buona parte del XX secolo, la AT&T aveva detenuto il monopolio del servizio telefonico, e come misura di bilanciamento le era stato vietato di entrare nel mercato dei computer.
- Dunque agli inizi degli anni ‘70 alla AT&T trovarono inutile investire ricerca in un sistema operativo, e decisero di rilasciare copie dello Unix a varie università, **insieme ai sorgenti del codice**, scritti in C.
- Grazie anche all’eleganza e relativa semplicità dello Unix dunque, un’intera generazione di studenti cominciò a studiare il C all’interno di corsi di programmazione e di introduzione ai sistemi operativi.
- Negli anni ’80, lo sviluppo e il successo del C++, compatibile col C, e la scelta della Microsoft di adottare il C come linguaggio standard (gli esempi delle API Windows erano in C) decretarono la definitiva ¹³³ affermazione del C e il tramonto dell’era Pascal.

Le ragioni del successo del C

- Naturalmente, il C non avrebbe avuto successo se non avesse anche avuto molte ottime qualità:
- **Portabilità**: un programma C gira facilmente su qualsiasi piattaforma, senza dovervi apportare alcuna modifica (a parte ricompilarlo).
- **Modularità**: grandi progetti software possono essere spezzati in moduli più piccoli sviluppabili e testabili separatamente.
- **Efficienza**: i programmi scritti in C girano in media più velocemente dei programmi scritti in qualsiasi altro linguaggio ad alto livello. Ciò rende il C perfetto per la programmazione di sistema (lo Unix, appunto, è scritto in C), per i sistemi embedded, e come linguaggio intermedio (molti compilatori e interpreti sono scritti in C)

Le ragioni del successo del C

- **Compattezza:** è possibile scrivere programmi in C molto sintetici. Il che fa risparmiare tempo (naturalmente questa caratteristica può anche essere fonte di errori, o di scarsa comprensibilità del codice).
- **Tipi flessibili:** al contrario di linguaggi come il Pascal, in C il sistema di tipi permette di combinare fra loro variabili di tipo diverso, il che di solito aumenta la velocità di scrittura e di esecuzione dei programmi. (anche questa ovviamente può essere una sorgente di errori)
- **Gestione efficiente della memoria:** attraverso un uso estensivo dei puntatori (che naturalmente sono anche potenziali sorgenti di errore)
- **Bitwise programming:** il C permette di manipolare con opportuni operatori anche i singoli bit di un byte, il che può essere molto utile nella programmazione di sistema.

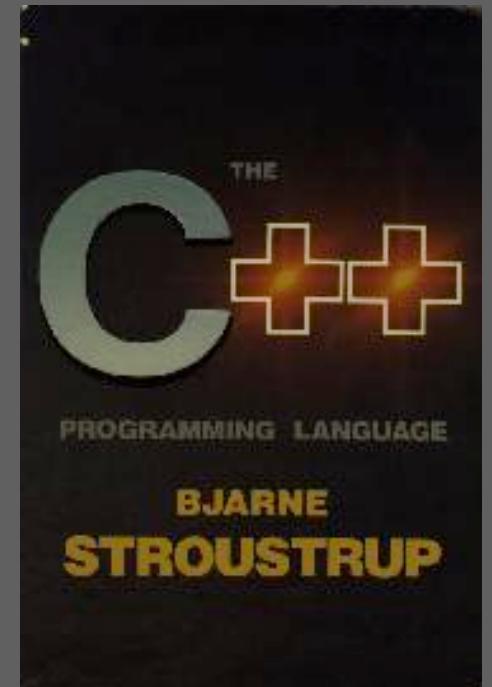
Nota di colore: the obfuscated C code

- È risaputo che il C permette di scrivere codice particolarmente conciso ma altrettanto oscuro e incomprensibile, sebbene funzionante.
- L'**International Obfuscated C Code Contest** premia ogni anno gli autori dei programmi più astrusi (roba da nerd estremi, ovviamente):
<http://www.ioccc.org>
- Del resto, una semplice funzione come *strcpy* mostra bene come la sinteticità del C possa andare a scapito della sua leggibilità:

```
char *strcpy(char *t, const char *s) {  
    char *p = t;  
    while(*t++ = *s++);  
    return p;  
}
```

Il C incontra gli oggetti: il C++

- E in effetti, inizialmente il C++ fu chiamato **C with classes**, mentre il nome odierno, **C++**, fu adottato solo intorno al 1984.
- Il C++ nasce quasi per caso all'inizio degli anni '80 ai Bell Labs, grazie a **Bjarne Stroustrup**. Questi conduceva le sue ricerche sul *calcolo distribuito*, ma aveva anche un interesse personale nella simulazione e dunque conosceva bene il Simula.
- Stroustrup decise così di estendere il C aggiungendovi oggetti e classi: la prima implementazione del C++ era semplicemente un preprocessore che convertiva i programmi C++ in C.



Il C incontra gli oggetti: il C++

- Lo sviluppo del C++ non aveva obiettivi commerciali, ma l'idea della programmazione object oriented stava prendendo piede, e in pochi anni il C++ divenne il linguaggio object oriented più usato (in attesa dell'arrivo del suo principale concorrente, Java).
- Il successo del C++ si deve anche al fatto che coniuga le qualità della programmazione ad oggetti senza perdere nessuna delle qualità del C.
- Mantenere la compatibilità col C ha però portato ad un linguaggio con aspetti complessi e difficili da comprendere, e la scrittura di programmi chiari e in generale l'uso di un elegante stile di programmazione non sono incoraggiati dalla sintassi del C++.
- Nota a margine: nel 2001 è stato rilasciato il **D**. Una evoluzione¹³⁸ del C++ influenzata da Java, Python e Ruby (su questi ci torneremo).

Objective-C e C-Sharp

- La combinazione del C con il paradigma object oriented è stata particolarmente fertile di nuovi linguaggi di successo:
- **Objective-C** è una estensione object oriented del C ispirata allo Smalltalk e sviluppata a metà degli anni ‘80 da **Brad Cox**, la cui compagnia fu poi assorbita dalla **NeXT Computers** (fondata nel 1985 da Steve Jobs durante il suo esilio dalla Apple).
- La NeXT fu poi acquistata dalla Apple nel 1997: Jobs rientrò in Apple e Objective-C è diventato uno dei principali linguaggi usati per lo sviluppo di applicazioni negli ambienti **OS X** e **iOS** della Apple
- **C-Sharp**, o meglio ancora **C#**, è il linguaggio ad oggetti sviluppato dalla Microsoft tra il 1999 e il 2000 come parte della piattaforma di sviluppo software **.NET** (si legge “dotnet”). Il C# combina caratteristiche del C, del C++ e di Java.

Gli anni '90 e l'era Internet: Java

- L'origine del linguaggio **Java** è piuttosto curiosa, e in qualche modo ricorda quella dello Smalltalk.
- Il linguaggio fu sviluppato sotto la guida di **James Gosling** a partire dal 1990 alla **Sun Microsystems**: fondata nel 1982, fu una delle aziende della Silicon Valley di maggior successo negli anni '80 e '90.
- La Sun ha il suo posto nella storia dell'informatica: oltre ad aver dato i natali a Java, alla Sun è stato sviluppato il sistema operativo **Solaris** (una variante di Unix ampiamente diffusa), e l'architettura software del **Network File System (NFS)**.
- Di grande successo fu la linea dei **Sun Server** e **workstation**, che contribuirono tantissimo all'avanzamento delle architetture RISC.
Nel 2010 la Sun è stata assorbita dalla **Oracle**.



Java

- Alla fine del 1990 alla Sun parte un progetto basato sull'uso di un **set top box** per **TV via cavo**. Un apposito telecomando e il televisore avrebbero costituito le interfacce di input e output del sistema.
- Il televisore sarebbe divenuto così un sistema interattivo, eseguendo applicativi legati alle trasmissioni televisive. Ad esempio, la pubblicità di un'auto poteva essere trasformata in una visita virtuale all'interno dell'auto stessa. (Oggi giorno, esempi di set top box sono i dispositivi per la ricezione della TV digitale e satellitare)
- Dunque, c'era bisogno di un linguaggio di programmazione nel quale sviluppare applicazioni che potessero essere scaricate dalla rete a cui era connesso il televisore ed eseguite sul processore poco potente del dispositivo.



Java

- Il linguaggio più in voga in quel periodo era il C++, che però era troppo pesante per girare su una CPU di basse prestazioni, e anche inaffidabile per scrivere programmi che non dovevano essere particolarmente veloci o complessi, ma non dovevano mai dare errore.
- Il team di Gosling pensò allora di modificare opportunamente il C++ per le esigenze del progetto. Inizialmente denominato “**C++ ++ --**”, ad indicare che al C++ erano state aggiunte alcune cose e tolte altre, il nuovo linguaggio in via di sviluppo fu poi ridenominato **Oak** (dalla finestra del suo ufficio Gosling poteva vedere una pianta di quercia).
- Ma a lavori ormai in notevole stato di avanzamento, nel 1993 un evento epocale rischiò di mandare all'aria l'intero progetto della Sun: il **World Wide Web** debuttava fra il grande pubblico, rendendo improvvisamente meno attraente l'idea di una TV interattiva. 142

Nota a margine: il World Wide Web

- Si tende spesso ad identificare il termine **World Wide Web (WWW)** con la rete Internet, ma sono due cose distinte, benché collegate.
- Internet è il risultato di un lungo lavoro per permettere a reti di comunicazione sviluppate indipendentemente In USA ed Europa di comunicare fra loro usando un protocollo comune (il termine Internet nasce nel 1974, FTP ed e-mail esistono dall'inizio degli anni '70).
- Ma è dagli anni '90 che Internet diventa popolare, con lo sviluppo di **HTTP**, **HTML** e i primi **Browser**, che trasformavano Internet in uno spazio aperto e omogeneo di risorse digitali identificate da *indirizzi* (gli **URL**) e interconnesse fra loro da ciò che chiamiamo **hyperlink**.
- Internet è dunque il substrato di reti (risorse hardware) e protocolli di comunicazione (risorse software) sul quale “vive” il WWW. 143

Nota a margine: il World Wide Web

- I primi passi del WWW furono compiuti in sordina. HTTP, HTML e il primo browser sono stati messi a punto nel 1990 al **CERN di Ginevra** da **Tim Berners Lee**. Il primo sito Web risale al dicembre del 1990.
- Ma nel 1993 si verifica quello che è considerato da tutti il punto di svolta: al **NCSA** dell'università dell'Illinois viene sviluppato **Mosaic**, il primo browser capace di combinare testo e immagini.
- L'interfaccia grafica di Mosaic rendeva le pagine web interessanti per divulgare al grande pubblico (ormai i personal computer erano ampiamente diffusi) una (potenzialmente) infinita quantità di informazioni in una modalità grafica attraente.
- E l'idea di TV interattiva diveniva improvvisamente obsoleta...



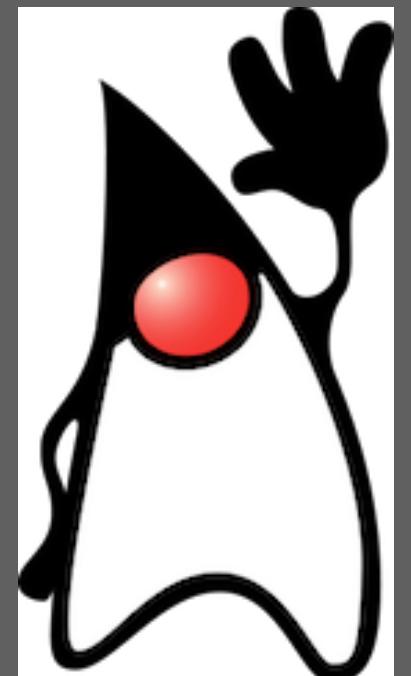
Java

- Ma alla Sun furono bravi nel convertire un progetto che rischiava di fallire in una idea di enorme successo: applicazioni scaricate da Internet insieme al codice HTML ed eseguite localmente sul browser in modo da **aggiungere capacità interattive alle pagine Web**: le **applet**.
- Il linguaggio delle applet doveva essere indipendente dal browser e dall'hardware su cui girava, secondo la filosofia **WORA** lanciata dalla stessa Sun: **Write Once Run Anywhere** (o **WORE**, E=Everywhere).
- La sintassi simile al C++ fu conservata, e il nome del linguaggio, dopo essere stato cambiato di nuovo in *Green* (dal nome con cui era partito il progetto, nel 1990), divenne infine **Java** (dall'omonima varietà di caffè indonesiano).



Java

- Java fu rilasciato nel 1995 al culmine di un grosso sforzo di marketing e con grandi aspettative da parte di tutto il mondo informatico (una aspettativa simile solo a quella di quasi quaranta anni prima in occasione del lancio del FORTRAN).
- Oltre a Java veniva reso disponibile anche il browser **HotJava** (scritto in Java, inizialmente avrebbe dovuto chiamarsi *WebRunner*, in onore del film *Blade Runner*), in grado di eseguire le applet scritte in Java.
- Ben presto però, visto il successo dell'idea, tutti i principali browser vennero aggiornati incorporando una **Java Virtual Machine** per poter garantire l'esecuzione delle applet Java su qualsiasi piattaforma.



Java

- E molto rapidamente, le tante qualità di Java ne decretarono il successo al di là della semplice scrittura di applet per pagine Web.
- Grazie alla sintassi simile, Java poteva facilmente essere appreso da chi conoscesse già C e C++, ed era più semplice e affidabile. Negli anni ‘90 i programmi Java erano decisamente meno efficienti di quelli scritti in C++, ma il gap di prestazioni è diminuito negli anni.
- Ma soprattutto, il successo di Java si deve alla sua portabilità: un sorgente Java viene compilato in un codice intermedio, il **bytecode**, (un’idea già usata dal Pascal con il P-code) che viene poi interpretato da una Java Virtual Machine, della quale esistono ormai implementazioni per qualsiasi piattaforma Hardware/software.
- Grazie alla disponibilità di implementazioni gratuite, in dieci anni Java è divenuto il linguaggio di programmazione più diffuso e usato.

Nota a Margine: the Java Saga

- La nascita della tecnologia Java è stata raccontata in toni quasi epici in un articolo apparso sulla rivista **Wired** nel dicembre del 1995 che vale la pena di leggere: **The Java Saga**. (si trova facilmente in rete)
- Oltre ad una serie di antefatti divertenti, l'articolista, *David Bank*, racconta di come l'enorme investimento fatto da Sun su Java avesse origine nel desiderio del suo amministratore delegato del tempo, **Scott McNealy**, di distruggere l'egemonia della Microsoft (e forse c'era di mezzo anche una antipatia personale verso Bill Gates). Nelle parole con cui McNealy presentava Java al mondo:

“Java blows up Gates's lock and destroys his model of a shrink-wrapped software that runs only on his platform”



Verso il 21° secolo: i linguaggi di scripting

- Un **linguaggio di scripting** è un linguaggio per scrivere **script**: programmi di solito corti **eseguiti da un interprete**.
- Un classico esempio è la **Shell Unix** (o una sua qualsiasi variante), con cui si possono scrivere programmi interpretati dalla shell stessa e usati per automatizzare attività di routine del sistema operativo.

```
#!/bin/bash
FILES="$@"
for f in $FILES
do
    if [ -f ${f}.bak ]          # if .bak backup file exists, read next file
    then
        echo "Skiping $f file..."
        continue                # read next file and skip the cp command
    fi
    /bin/cp $f ${f}.bak         # no backup file, use cp command to copy file
done
```

I linguaggi di scripting

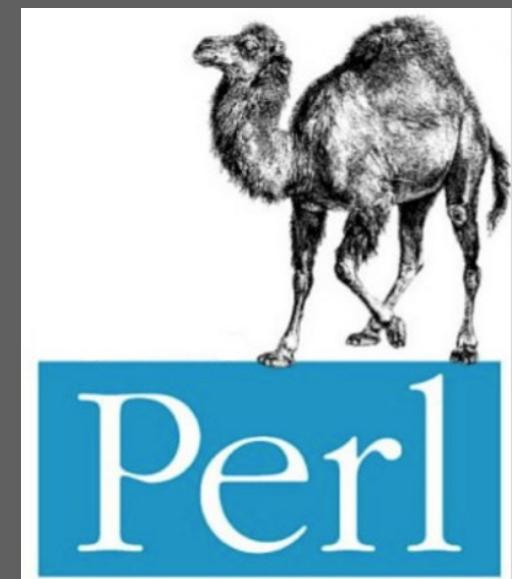
- Normalmente, un linguaggio di scripting viene usato nel contesto di un ambiente software all'interno del quale lo script può essere scritto e fatto girare quasi istantaneamente, poiché è l'ambiente software stesso a fornire tutto ciò di cui lo script ha bisogno per girare.
- Così, una qualsiasi shell Unix è l'ambiente nel quale possiamo scrivere e far girare immediatamente un shell script. Un Web browser è l'ambiente in cui possiamo scrivere e visualizzare immediatamente pagine web interattive.
- In linea di principio, qualsiasi linguaggio può essere un linguaggio di scripting, tuttavia alcuni linguaggi sono naturalmente orientati alla scrittura di script, mentre altri richiederebbero l'implementazione di un ambiente software (un interprete e librerie opportune per interagire con l'hardware sottostante) normalmente non disponibile.

I linguaggi di scripting

- Negli ultimi anni si è avuta una enorme diffusione di **linguaggi di scripting general purpose**, in grado cioè di coniugare:
- **1) caratteristiche dei classici linguaggi di programmazione** (ad esempio l'essere strutturati, object-oriented e più o meno indipendenti dal dominio di applicazione) e
- **2) caratteristiche dei classici linguaggi di scripting:** prototipizzazione veloce, controllo dei tipi a run time, esecuzione immediata attraverso un interprete (o alternativamente attraverso la modalità di compilazione **just-in-time**)
- Attualmente, i linguaggi di scripting *più o meno* general purpose di maggior successo sono **Perl**, **Python**, **PHP**, **Javascript** e **Ruby**, tutti sviluppati negli anni '90 (Perl a fine anni '80).

Perl

- Perl è stato ideato da Larry Wall nel 1987 come linguaggio di scripting che girava in ambiente Unix per manipolare testi e file. E in effetti, inglobava caratteristiche della Shell, di Awk e sed.
- A partire dagli anni '90 ha subito diverse evoluzioni fino a diventare un linguaggio general purpose, in ogni caso ispirato dall'ambiente in cui è nato: lo Unix e il linguaggio C.
- Una delle ragioni del successo di Perl è che è facilmente integrabile con altri linguaggi e ambienti di sviluppo, ed è supportato dalla maggior parte dei sistemi operativi.
- È usato soprattutto per scrivere script CGI, ossia programmi eseguiti su un server Web e il cui output viene poi inviato al browser lato client. È anche usato per lo sviluppo di interfacce grafiche e per la programmazione di sistema.



Il bubblesort in Perl

```
sub bubble_sort {  
    for my $i (0 .. $#_){  
        for my $j ($i + 1 .. $#_){  
            $_[[$j]] < $_[[$i]] and @_[$i, $j] = @_[$j, $i];  
        }  
    }  
}
```

Python

- **Python** nasce nel 1991 dal lavoro di **Guido von Rossum**, e si chiama così in onore del gruppo comico inglese dei *Monty Python*.
- Possiede caratteristiche object oriented e funzionali, e una sintassi semplice. Ad esempio, i blocchi di codice sono delimitati non da parentesi o keywords, ma dall'indentazione mediante spazi bianchi.
- L'implementazione più diffusa di Python, **CPython**, è scritta in C, ed è un interprete da linea di comando, come la Shell (di fatto l'ambiente implementa una macchina virtuale che esegue bytecode)
- Python viene usato estensivamente per lo sviluppo di applicazioni web e in campo scientifico, e ormai molti sistemi operativi includono un interprete per script scritti in Python.



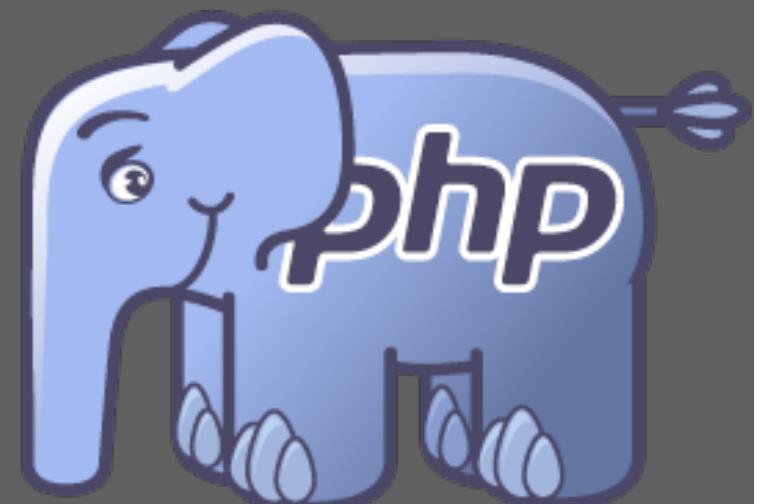
Il bubblesort in Python

```
def bubble_sort(seq):
    changed = True
    while changed:
        changed = False
        for i in xrange(len(seq) - 1):
            if seq[i] > seq[i+1]:
                seq[i], seq[i+1] = seq[i+1], seq[i]
                changed = True
    return None
```

```
if __name__ == "__main__":
    # sample usage and simple test suite
    from random import shuffle
    testset = range(100)
    testcase = testset[:]          # make a copy
    shuffle(testcase)
    assert testcase != testset    # we've shuffled it
    bubble_sort(testcase)
    assert testcase == testset   # we've unshuffled it back into a copy
```

PHP

- PHP (Personal Home Page, ma poi ridefinito come PHP: Hypertext Preprocessor) è stato creato da Rasmus Lerdorf nel 1994. È stato concepito per la scrittura di pagine web dinamiche, e poi come linguaggio di scripting per lo sviluppo di applicazioni lato server.
- Si è poi evoluto in un linguaggio general purpose interpretato, ad esempio usato per la scrittura di applicazioni grafiche *stand-alone*. Nelle versioni più recenti, PHP contiene anche funzionalità object oriented.
- PHP è free, e viene ormai supportato praticamente da qualsiasi piattaforma, sistema operativo, e web hosting provider. Anche molti database relazionali supportano PHP.

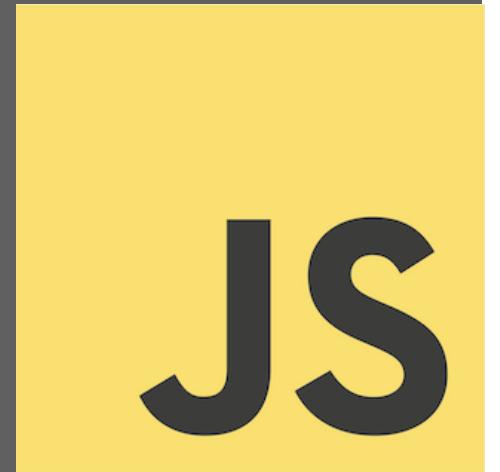


Il bubblesort in PHP

```
function bubbleSort( array &$array )
{
    do
    {
        $swapped = false;
        for( $i = 0, $c = count( $array ) - 1; $i < $c; $i++ )
        {
            if( $array[$i] > $array[$i + 1] )
            {
                list( $array[$i + 1], $array[$i] ) =
                    array( $array[$i], $array[$i + 1] );
                $swapped = true;
            }
        }
    }
    while( $swapped );
}
```

JavaScript

- **JavaScript** è stato sviluppato nel 1995 (in 10 giorni!) da **Brendan Eich** all'interno della **Netscape Communication Corp.** per scrivere applet, come alternativa più leggera e facile da usare di Java.
- Una vera e propria guerra tra i browser **Netscape** ed **Explorer** portò la Microsoft a sviluppare un proprio linguaggio per applet, il **Jscript**, che tuttavia non riuscì mai ad avere la meglio, e JavaScript è ormai divenuto una delle tecnologie fondamentali del Web.
- Nelle versioni più recenti, JavaScript è un linguaggio strutturato, con caratteristiche object-based e funzionali.
- È usato principalmente per scrivere applicazioni lato client che aggiungono funzionalità alle pagine Web, ma anche come linguaggio di scripting in molti altri contesti.

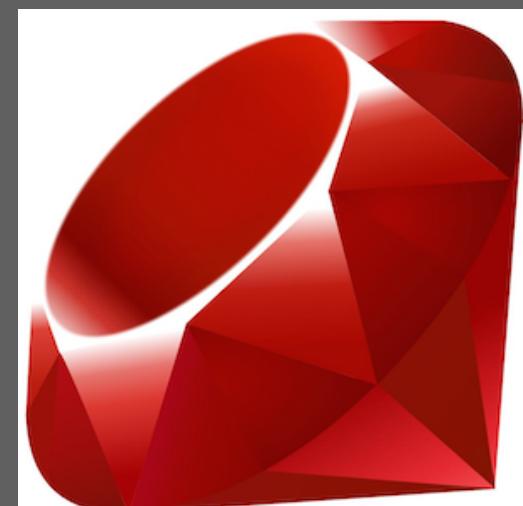
A large, bold, black "JS" logo is positioned on a yellow rectangular background in the bottom right corner of the slide.

Il bubblesort in JavaScript

```
Array.prototype.bubblesort = function() {
    var done = false;
    while (!done) {
        done = true;
        for (var i = 1; i<this.length; i++) {
            if (this[i-1] > this[i]) {
                done = false;
                [this[i-1], this[i]] = [this[i], this[i-1]]
            }
        }
    }
    return this;
}
```

Ruby

- Ruby è stato sviluppato da **Yukihiro Matsumoto** e reso disponibile a fine 1995. Il suo autore lo descrive come un linguaggio semplice come il Lisp, orientato agli oggetti come lo Smalltalk, e pratico come il Perl.
- Esistono ormai interpreti Ruby per la maggior parte degli ambienti e sistemi operativi, scritti in C, C++ o in Java. Ruby è particolarmente adatto per lo sviluppo di applicazioni Web, dove molti lo considerano il naturale successore di PHP.
- In particolare, molta della popolarità di Ruby si deve alla piattaforma **rails**, la piattaforma software open source scritta in Ruby per lo sviluppo di applicazioni Web (**rubyonrails.org**)



Il bubblesort in Ruby

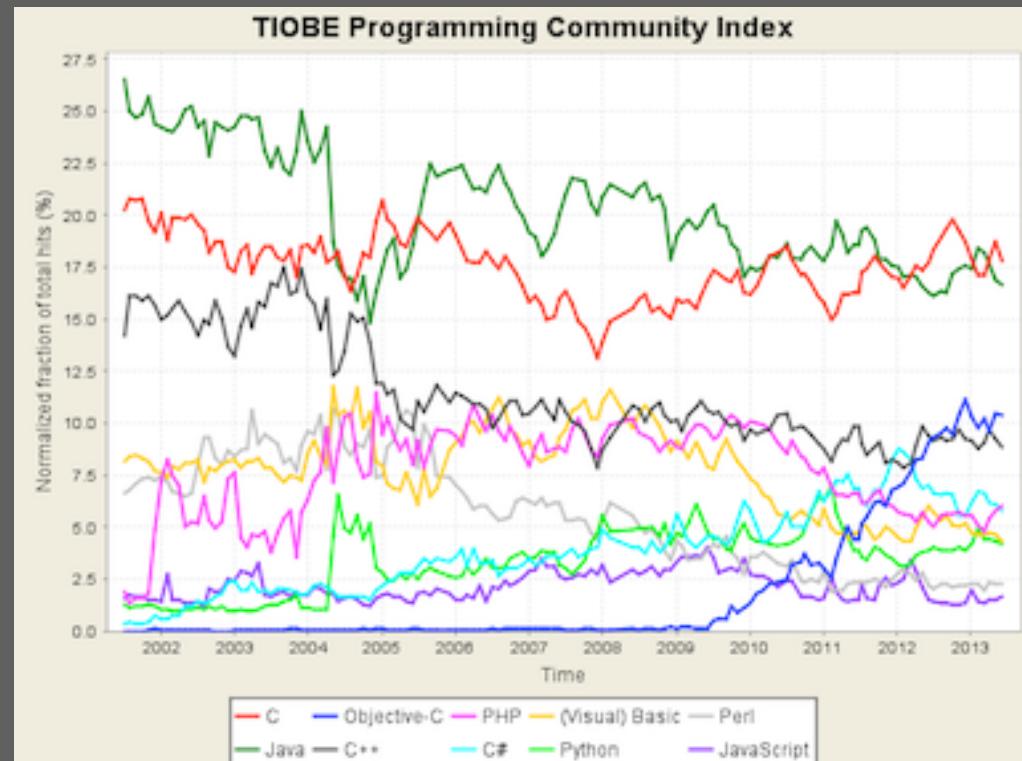
```
class Array
  def bubblesort!
    length.times do |j|
      for i in 1...(length - j)
        if self[i] < self[i - 1]
          self[i], self[i - 1] = self[i - 1], self[i]
        end
      end
    end
    self
  end
end
ary = [3, 78, 4, 23, 6, 8, 6]
ary.bubblesort!
p ary
#=> [3, 4, 6, 6, 8, 23, 78]
```

Linguaggi di programmazione più popolari

- Esistono diversi indici di popolarità dei linguaggi di programmazione oggi in uso, e a seconda di quale parametro venga considerato, si ottengono risultati diversi.
- Possiamo ad esempio contare il **numero di ricerche in rete** per un certo linguaggio, o il **numero di richieste di programmatori** del linguaggio, il **numero di manuali venduti** sul linguaggio, o semplicemente contare il **numero di pagine web** restituite da Google cercandone il nome.
- In ogni caso, i risultati non saranno diametralmente opposti fra di loro: come vedremo, i linguaggi più popolari rimangono quasi sempre gli stessi. Possono cambiare un po' le loro posizioni in classifica, con Java e C (o sue varianti evolute) più o meno sempre ai primi posti.
162

TIOBE Programming Community Index

- È basato sul numero di risposte a ricerche sul Web che contengono il nome di uno specifico linguaggio. Viene pubblicato periodicamente e permette quindi di analizzare la popolarità di un linguaggio nel tempo.
- Nel sito: http://www.tiobe.com/tiobe_index si possono trovare molte statistiche interessanti relative ai 100 linguaggi più usati, incluse le classifiche degli anni scorsi.
- In figura la popolarità dei primi dieci linguaggi dal 2002 al 2015 con Java e C ampiamente in testa rispetto a tutti gli altri. Nel prossimo lucido le prime dieci posizioni a marzo 2015 e 2016.



TIOBE Programming Community Index

Mar 2016	Mar 2015	Language	Ratings	Change
1	2	Java	20.528%	+4.95%
2	1	C	14.600%	-2.04%
3	4	C++	6.721%	+0.09%
4	5	C#	4.271%	-0.65%
5	8	Python	4.257%	+1.64%
6	6	PHP	2.768%	-1.23%
7	9	Visual Basic .NET	2.561%	+0.24%
8	7	JavaScript	2.333%	-1.30%
9	12	Perl	2.251%	+0.92%
10	18	Ruby	2.238%	+1.21%

IEEE Spectrum Top Ten (2015)

- La rivista **IEEE Spectrum** usa una valutazione basata su 12 metriche, e osserva giustamente che la classifica che se ne ottiene dipende da quale importanza si da a ciascuna metrica.
- Sul sito della rivista (**spectrum.ieee.org**) è anche disponibile un tool (a pagamento) che permette di variare i pesi assegnati a ciascuna misura in base alle proprie esigenze.
- Nel lucido successivo vediamo la classifica del 2015 (sx) secondo i pesi di default scelti dalla rivista, e il confronto col 2014 (dx). In questa classifica Perl e Visual Basic sono al 15° e 16° posto, mentre R è un linguaggio (e un ambiente) per calcoli statistici e data mining.

IEEE Spectrum Top Ten (2015)

- Types (of usage): Web, Mobile, Enterprise, Embedded

Language Rank	Types	Spectrum Ranking	Spectrum Ranking
1. Java	🌐📱💻	100.0	100.0
2. C	📱💻CHIP	99.9	99.3
3. C++	📱💻CHIP	99.4	95.5
4. Python	🌐💻	96.5	93.5
5. C#	🌐📱💻	91.3	92.4
6. R	💻	84.8	84.8
7. PHP	🌐	84.5	84.5
8. JavaScript	🌐📱	83.0	78.9
9. Ruby	🌐💻	76.2	74.3
10. Matlab	💻	72.4	72.8

The PYPL Index

- Il **PYPL PopularitY of Programming Languages Index** calcola la classifica contando il numero di ricerche fatte su google per tutorial di un certo linguaggio.
- L'idea è che poiché un tutorial è un buon punto di partenza per imparare un linguaggio, molte ricerche per un suo tutorial testimoniano la popolarità di quel linguaggio.
- Sul sito (**pypl.github.io/PYPL.html**) sono disponibili mappe interattive per confrontare l'andamento nel tempo di diversi linguaggi.
- Nella pagina successiva la classifica a marzo 2016: Ruby, Visual Basic e Perl sono rispettivamente al 12°, 13°, e 15° posto.
Swift è una versione evoluta di Objective-C sviluppata alla Apple per sviluppare applicazioni in ambiente iOS e OS X.
167

The PYPL Index

Mar 2016	Variation vs Mar 2015	Language	Ratings	Trend vs Mar 2015
1		Java	24.1 %	+0.2 %
2	+	Python	12.1 %	+1.4 %
3	-	PHP	10.6 %	-0.9 %
4		C#	8.8 %	-0.1 %
5		C++	7.5 %	-0.4 %
6		C	7.4 %	+0.2 %
7		Javascript	7.4 %	+0.4 %
8		Objective-C	5.0 %	-0.8 %
9	++	R	3.0 %	+0.4 %
10		Swift	3.0 %	+0.4 %

Programmers' Job Index

- E perché non contare il numero di offerte di posti di lavoro per programmatori in un certo linguaggio?
- Lo fa la **Coding Dojo** (www.codingdojo.com), un sito che si occupa di problematiche del mondo del lavoro in campo informatico. Nella prossima pagina i risultati, dove IOS in realtà corrisponde a Swift.
- **SQL** finisce al primo posto perché qualsiasi applicazione che debba usare informazione immagazzinata in un database quasi certamente usa SQL. Naturalmente SQL non è un vero e proprio linguaggio di programmazione, ma più propriamente un sofisticato linguaggio di interrogazione di basi di dati.
- Come mai in questa classifica il C manca del tutto?

Programmers' Job Index

Languages ranked by number of programming jobs

*Data from
Indeed.com
2016*

