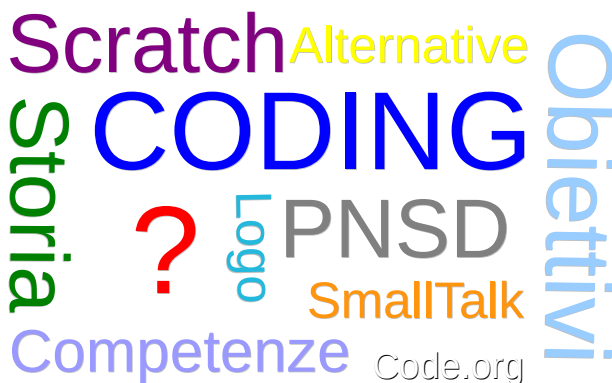


# Dietro il Coding



Significato, storia, obiettivi  
e alternative possibili

ver.0.66 del 01/10/16

“Je suis fracas quand la foule est tranquille”  
(Zebda, 1998)

© Stefano Penge 2016  
<http://steve.lynxlab.com>  
Creative Commons BY/SA

---



## Indice generale

Come nasce questo testo.....	5
1. Che cos'è il Coding.....	9
2. Perché è così importante?.....	17
3. Una valutazione storica.....	21
4. Come si parla del Coding?.....	25
Primo Intermezzo: che significa opensource?.....	31
5. A che serve il Coding?.....	35
6. Chi può insegnare il Coding?.....	47
Secondo Intermezzo: le differenze tra linguaggi.....	51
7. Che linguaggio?.....	55
8. Il modello didattico dietro Scratch.....	75
9. Come andrà praticato il Coding.....	83
10. Un po' di storia.....	89
11. E oggi?.....	101
12. Come potrebbe funzionare davvero.....	103
Suggerimenti di lettura.....	105



## Come nasce questo testo

Questo testo è stato scritto in occasione di due seminari organizzati dall'Associazione Gessetti Colorati, a Ivrea e a Torino, nel settembre 2016, su invito di Rodolfo Marchisio. Il titolo dei seminari era "Non di solo coding..." e univa riflessioni generali sul tema dei diritti/doveri digitali ad analisi di artefatti, come gli algoritmi e gli ambienti.

Raccoglie in realtà anche degli scritti precedenti, pubblicati sul mio blog (<http://steve.lynxlab.com>) dove da qualche anno cerco di capire il fenomeno del Coding. Di qui il titolo: andare "dietro il coding" significa cercare di capire cosa significa, cosa c'è sotto, cosa c'era prima, a cosa punta. Non è, quindi, né una difesa del Coding né una sua condanna.

In realtà è un interesse di ritorno: in un'era precedente – direi almeno venticinque anni fa - ho insegnato informatica a dei ragazzini delle medie in una scuola privata di Roma. E non essendoci un progetto didattico definito, avevo deciso di insegnare anche programmazione. Mi ricordo mattinate divertenti, entusiasmanti, ma anche blocchi e crisi. L'esperienza è durata alcuni anni, poi sono passato ad altro (per modo di dire: a creare software didattico o ambienti per l'apprendimento digitale); ma

alcune riflessioni fatte allora me le sono portate dietro fino ad oggi.

Ad esempio: il momento più difficile per i ragazzi non è mai stato la scrittura del codice del programma, ma piuttosto l'impostazione iniziale (l'ideazione) e la correzione. La prima fase perché richiedeva una visione molto astratta di ciò che si voleva fare, e che ancora non esisteva (un videogioco che simula una corsa di cavalli: quali sono i dati? Quali i vincoli? Come si potrà interagire col gioco?). La seconda, perché scoprire un errore di progettazione è molto più difficile che trovare un errore di sintassi nel codice o una variabile scritta male.

Nel tempo ho poi scoperto che queste difficoltà non diminuiscono tra i programmatori professionisti, anzi; forse spaventano solo un po' meno. Gli errori più gravi, anche nei programmi che usiamo, sono quelli concettuali (come il famoso bug dell'anno 2000), non quelli sintattici.

A forza di scrivere codice, e soprattutto di leggere codice scritto da altri, piano piano mi sono accorto di quante differenze ci sono tra il codice scritto da uno e quello scritto da un altro; di quanta parte delle abitudini mentali e linguistiche delle persone si riflettono poi nel modo di programmare. Dove cominciano queste differenze? Sono le stesse che ci portano a scrivere lettere e racconti in modo così personale? C'è un rapporto tra i primi programmi e i primi componimenti scolastici?

Chi legge di più, scrive anche meglio il codice? E la differenza tra ragazze e ragazzi? E l'ambiente sociale, la cultura della famiglia, la lingua madre: quanto incidono sul risultato finale?

E poi: venticinque anni fa esistevano i videogiochi, ma certo non erano così diffusi, non stavano su ogni smartphone, non era normale vedere una signora di mezza età giocare mentre viaggia in metropolitana. Che significa, per un bambino che è nato con un videogioco in mano, programmarne uno? Che succede nel suo immaginario quando può mettersi dall'altro lato del palcoscenico, come regista, attore, padrone di quell'universo?

Eccetera eccetera.

Quando leggo di Coding nelle scuole primarie, mi vengono in mente tutte queste questioni, e mi domando come le affrontano oggi gli insegnanti utilizzando Scratch. Avvicinandomi un po', ho scoperto però che...





## 1. Che cos'è il Coding

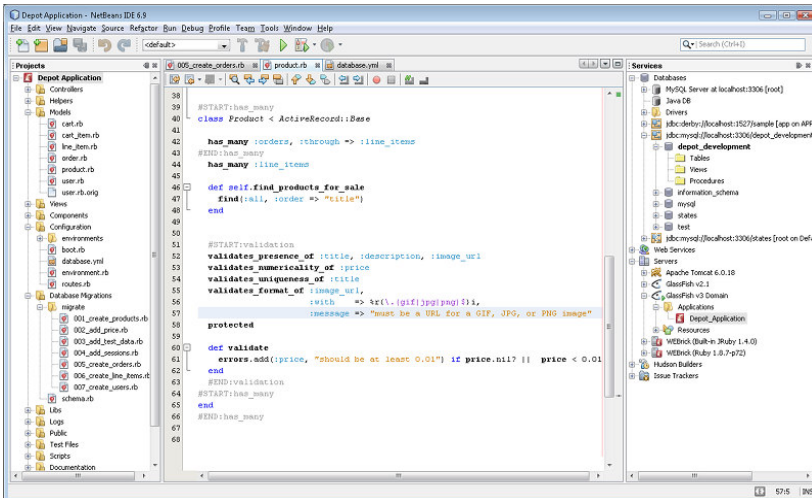
Per i pochi che non ancora lo sapessero: la parola "coding" significa letteralmente “l’attività di scrivere codice sorgente”, che è uno dei quasi-sinonimi di “programmare”.

Quasi, perché programmare può significare anche analizzare, progettare, verificare, integrare un codice sorgente, mentre coding fa riferimento solo alla scrittura del codice.

Infatti il codice sorgente si scrive, esattamente come un romanzo, o meglio come una sceneggiatura o una partitura musicale.

Bisogna conoscere la grammatica e il lessico di un linguaggio, ma non basta: bisogna avere in testa il plot, i personaggi, l’ambiente. O se preferite: bisogna avere in testa le funzioni di Propp.

Siccome è un’attività complessa, ci sono dei software per aiutare a farlo: si chiamano "editor", o più pomposamente "Ambienti Integrati per lo Sviluppo" (IDE). Come ad esempio NetBeans, che vedete qui di seguito.



NetBeans controlla la sintassi, evidenzia automaticamente i costrutti, permette di riusare frammenti, mostra l'aiuto contestuale, aiuta a seguire i collegamenti tra le centinaia di librerie che compongono un programma.

Con “Coding”, in questo momento e in Italia, ci si riferisce però alle attività di introduzione dei bambini alla programmazione, attraverso ambienti online di programmazione visuale, a partire dalla scuola primaria.

Cioè in cui paradossalmente non serve (anche se è possibile) scrivere il codice, ma è sufficiente posizionare oggetti simboli-

ci che stanno al posto di operatori, variabili, condizioni. Si utilizza un ambiente di programmazione in cui la metafora è quella dei "blocchi", un po' come se si trattasse del Lego: ci sono mattoncini di tipo diverso (di colori diversi), ognuno con un suo ruolo. Certi mattoncini si incastrano facilmente con altri. Questo "incastrarsi" rappresenta il vincolo della sintassi. Attenzione: non è un *linguaggio visuale*, ma un'interfaccia grafica per disporre linea di codice sorgente una sotto l'altra, nell'ordine giusto.

Non si scrive, perché scrivendo si possono commettere errori; scrivendo bisogna ricordarsi regole e termini.

Non si scrive perché scrivere è difficile.

Il Coding in questa accezione nasce probabilmente nel 2013 in Gran Bretagna, con un finanziamento ministeriale di 500.000 sterline. E' ancora online il sito dove veniva annunciata la nascita di una associazione nonprofit (Yearofcode) che promuoveva l'apprendimento della programmazione tra i bambini.

In Italia il MIUR, in collaborazione con il CINI e con un buon numero di "partner tecnologici" (TIM, Microsoft, Facebook, CISCO, Engineering, Samsung,...), all'interno del programma "La buona scuola" ha spiegato come e perché va intro-

dotto il Coding nella scuola in un sito dedicato, "Programma il Futuro".<sup>1</sup>

L'obiettivo dichiarato è “fornire alle scuole una serie di strumenti semplici, divertenti e facilmente accessibili per formare gli studenti ai concetti di base dell'informatica”.

Iniziativa numericamente di grande successo: in due anni sono stati raggiunti oltre 1 Milione di studenti con una media di 8 h di attività per ciascuno.



---

1 <http://programmailfuturo.it/come/ora-del-codice>

Nella versione aggiornata del progetto,<sup>2</sup> si chiarisce che "l'obiettivo non è quello di far diventare tutti dei programmatori informatici, ma di diffondere conoscenze scientifiche di base per la comprensione della società moderna". Un obiettivo quindi di livello culturale.

Inoltre, "la conoscenza dei concetti fondamentali dell'informatica aiuta a sviluppare la capacità di risoluzione di problemi e la creatività". E questo, invece, è un obiettivo metacognitivo.

La modalità base di partecipazione, definita L'Ora del Codice, consiste nel far svolgere agli studenti un'ora di avviamento al pensiero computazionale usando Scratch (ma si può fare anche senza computer). A quest'ora possono eventualmente seguire percorsi articolati e personalizzati.

Oltre alle attività centralizzate, esistono delle “varianti locali” del programma di Coding gestite nelle scuole direttamente da personale Microsoft, Samsung, Telecom all’interno di accordi-quadro con il MIUR.

L’iniziativa ministeriale ripropone corsi e ambienti di lavoro creati e gestiti da una associazione no profit statunitense, Code.org, che ha come partner Microsoft, Google e tanti altri.

---

2 <http://www.programmailfuturo.it/notizie/il-secondo-anno-del-progetto/avvio-secondo-anno>

L'Italia è il secondo Paese al mondo dopo gli USA per utilizzo dei materiali proposti.

Code.org ha in realtà obiettivi più ampi: punta all'integrazione razziale e a diminuire il gap di genere, a modificare i curricula delle scuole elementari e medie negli Stati americani, a soddisfare la richiesta di più informatica da parte dei genitori.

Obbiettivi, come si vede, abbastanza diversi da quelli Italiani.

Oltre alle lezioni introduttive per i più piccoli (su Code.org si parte da 4 anni) organizzate come puzzle da risolvere, immerse in universi molto conosciuti (Flappybird, Starwars, Minecraft) ci sono lezioni avanzate di Computer Science, materiali per i docenti (in inglese) e c'è un ambiente di apprendimento per ragazzi +13 che è basato su Javascript.

Ci sono poi liste di corsi offerti da terze parti (liberi o a pagamento).

Tutti i materiali sono distribuiti con licenza CC BY/NC/SA 4.0. Se non sapete cosa significa, non vi preoccupate e continuate a leggere: è spiegato poco più avanti.

*Nota: sul sito ministeriale invece non ci sono molti materiali di supporto per gli insegnanti. Per la verità, ne ho trovati solo due: la scansione di un libro di Carlo Batini<sup>3</sup> del 1984 (la via italiana al computational thinking) e un link ad un bell'ebook<sup>4</sup> del 2008 sugli aspetti culturali del digitale, Blown to bits.*

*Più avanti parleremo dell'iniziativa MIUR per costruire un curriculum sul Coding.*

---

<sup>3</sup> <http://www.programmailfuturo.it/media/docs/approfondimenti/Batini-basi-dell-Informatica.pdf>

<sup>4</sup> [http://www.bitsbook.com/wp-content/uploads/2008/12/B2B\\_3.pdf](http://www.bitsbook.com/wp-content/uploads/2008/12/B2B_3.pdf)





## 2. Perché è così importante?

Perché ritorna improvvisamente anche in Italia l'idea che si possa non solo *usare* la tecnologia digitale a scuola, ma anche *produrla*. E questa è una novità.

Dopo le prime esperienze dei progetti ministeriali in cui l'informatica tradizionale veniva inserita in vari modi nel curriculum, l'idea di far costruire programmi agli studenti (quella del Piano Nazionale Informatica) era stata abbandonata. La tecnologia andava usata, studiata, ma non prodotta.

Ora, sulla spinta di movimenti internazionali presenti in USA, GB, Francia, e sostenuti esplicitamente dai rispettivi Governi, si torna in qualche modo indietro (ma con vari anni di ritardo), da un uso "soft" dell'informatica ad un uso "hard". Ritornano di moda i robot, l'elettronica (Arduino), le stampanti 3D.

Come mai quest'inversione di direzione?

Sembra che sia diventata indiscutibile l'idea che per cavarcela in un mondo sempre più digitale occorra sapere non solo come funziona l'app che usiamo per chattare, ma anche sapere come si sviluppa.

Ora, al di là della condivisibilità dell'idea in sé – che pure andrebbe analizzata meglio, come faremo più avanti – è il ruolo della scuola pubblica che è in gioco.

La scuola, viene detto, si deve occupare di questa esigenza, deve investire risorse non per educare nelle materie tradizionali anche tramite l'uso di tecnologie ma per salvare i bambini da un futuro di cui non avranno la possibilità di essere attori.

Questo è ben diverso da quanto ipotizzato nei vari Multilab, PNTD, Scuola 2.0 etc.

Lì si parlava di didattica multimediale o di alfabetizzazione digitale, di allineare la scuola agli stili comunicativi, alle dinamiche e ai contenuti del resto del mondo – cioè in qualche modo del presente.

Qui si parte dal futuro: si vogliono sviluppare delle competenze che saranno utili più tardi, professionalmente ma non solo.

*Nota: è un passo importante, a cui potrebbe corrispondere un analogo interesse per il ruolo che avranno nel futuro le competenze linguistiche, matematiche, storiche che vengono acquisite oggi.*

*Serviranno? Saranno fondamentali? Salveranno i giovani?  
Vanno aggiornate sulla base di come sarà la società tra dieci,  
venti anni?*

*Belle domande. Ma qui ci occupiamo di altro.*



### 3. Una valutazione storica

Partire da futuro, sì. Sarebbe però altrettanto importante riesaminare tutti questi progetti ministeriali e farne una valutazione a posteriori. Sono stati investite risorse ingenti: prima di lanciarsi in un ennesimo programma non sarà il caso di sviluppare una critica onesta delle azioni passate?

E non solo iniziative guidate dall'alto: ci sono stati centinaia di insegnanti che da soli o riuniti in gruppi e associazioni hanno sperimentato per anni con i computer in classe o in laboratorio

Quali obiettivi sono ancora condivisibili tra quelli dichiarati allora e quali strategie si sono rivelate sostenibili (e quali invece tragicamente fallimentari)?

Fra l'altro, magari si scopre che qualcosa si può ancora recuperare.

Tra gli esempi di coding riportati nella pagina Facebook<sup>5</sup> del progetto o nell'area del concorso Codi-amo<sup>6</sup> tra storie e giochi e ci sono moltissimi disegni: girandole, stelle, spirali, etc.

---

<sup>5</sup> <https://www.facebook.com/programmailfuturo>

<sup>6</sup> <http://www.programmailfuturo.it/progetto/concorso/partecipanti>

Che assomigliano tanto ai migliaia di lavori fatti a partire almeno dagli anni '80 da insegnanti di buona volontà con il Logo nelle classi elementari e medie. In cui però il codice veniva scritto (magari in una variante italiana del linguaggio).

Vogliamo provare a riprendere quelle esperienze e a capire cosa ha (o non ha) funzionato?

I bambini e le bambine che hanno giocato con la tartaruga oggi hanno forse trenta o persino quarant'anni: vogliamo provare a vedere se quelle attività sono state utili ad avere un diverso approccio al digitale (oltre che una migliore comprensione della geometria piana)? Oppure sono rimaste un ricordo gradevole insieme alle partite di calcio a ricreazione?

A proposito di Logo: forse non tutti sanno che c'è un Logo che se ne sta nascosto dentro Libre Office. E' un'estensione sviluppata da due programmatori ungheresi, László Németh e András Tímár.<sup>7</sup>

L'idea è semplice: senza bisogno di un vero ambiente di sviluppo, si scrive codice direttamente nella pagina di Writer (si salva, si modifica, etc). E poi si esegue: il disegno risultante è un'immagine che si può copiare, stampare etc.

---

<sup>7</sup> <http://www.numbertext.org/logo/librelogo.pdf>

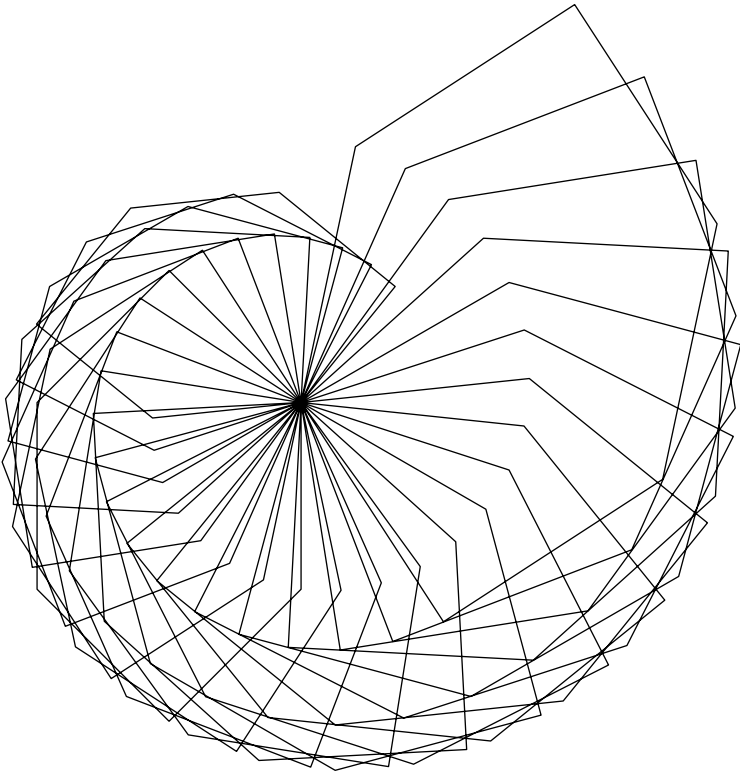
Esiste pure un manuale italiano,<sup>8</sup> scritto da Andreas R. Formiconi.

Per esempio, l'immagine della pagina accanto è stata generata proprio qui dentro, con qualche riga di codice scritta direttamente nel testo. L'interessante, a mio avviso, è che viene sottolineata la parentela tra la scrittura in una lingua naturale e la scrittura in un linguaggio di programmazione, almeno nel senso che si può usare lo stesso strumento digitale (un word processor) e le stesse semplici modalità di conservazione, condivisione e modifica dei sorgenti (i documenti ODT).

Un passo ulteriore, a mio avviso utile, sarebbe quello di cercare parentele più strette tra le diverse forme di scrittura. Di questo, più avanti.

---

<sup>8</sup> <http://iamarf.ch/unifi/Piccolo-manuale-LibreLogo.pdf>





## 4. Come si parla del Coding?

Parlare del Coding in Italia dovrebbe significare un ripensamento di quello che abbiamo fatto con i computer nelle scuole, di quello che stiamo facendo e di quello che vorremmo fare.

E scusate se è poco.

In attesa di questo ripensamento, almeno auspicato, ho cercato qua e là dei pareri oggettivi, delle valutazioni delle esperienze fatte, dei progetti a medio termine. Mi pare però che si trovi in giro soprattutto grande entusiasmo oppure fiera opposizione.

Ecco un elenco di opinioni sparse che ho raccolto:

- Ma io che c'entro, io insegno latino...
- E' tutta una manovra far entrare l'aziendalismo nella scuola e per sottomettere i docenti alle logiche del mercato.
- Che ti devo dire? Con Scratch ci divertiamo un sacco, l'ora passa in un attimo
- La "scuola digitale" non esiste e non sono gli strumenti che possono indurre cambiamenti significativi anche se conditi di (pseudo) metodologia.

- Si passa troppo tempo al computer tralasciando i rapporti umani!!!
- Mi raccomando ragazzi: fatevi tutti un account su Gmail e poi registratevi su [www.lamiaclasseduepuntzero.it](http://www.lamiaclasseduepuntzero.it)
- E' inutile studiare: tanto gli studenti ne sanno sempre più di noi, loro sono nativi
- Probabilmente anche tu hai già utilizzato strategie tipiche del metodo computazionale ma non ne sei consapevole
- Ma vuoi mettere il Pascal? Ai miei tempi si che si facevano cose serie coi computer
- A che serve? I computer sono stupidi, Wikipedia è piena di fatti ma bisogna insegnare la capacità critiche

Insomma, punti di vista molto diversi tra loro, ma che dicono di più sull'ideologia di chi parla che non sull'oggetto specifico del discorso.

Ho cercato di raggruppare queste opinioni e alla fine ho trovato tre partiti schierati:

- quelli che sono incondizionatamente a favore del “coding in classe”, dei Dojo, della settimana, il giorno e l’ora dei codice (gli **entusiasti**);
- quelli che sono contrari per principio all’insegnamento della programmazione ai bambini (i **luddisti**);
- quelli che sono delusi e tornano sui propri passi (i **disillusi**).

Non faccio parte di nessuno di questi tre partiti, e nel seguito proverò a spiegare perché. Mi piacerebbe però che – magari anche grazie alla lettura di testi come questo - si infoltisse la sparuta schiera di un quarto partito:

- quelli che provano a capirci qualcosa (i **critici**).

## Gli Entusiasti

Dei primi, non condivido l’entusiasmo acritico, senza radici nella storia e senza una visione degli aspetti culturali, sociali, economici che sono intorno al coding.

Non capisco come ci si possa dedicare integralmente a Scratch come se non ci fosse mai stato altro, o come se non esistessero alternative.

Non capisco come si possa copiare un modello educativo statunitense senza adattarlo al contesto italiano, o almeno senza capirlo.

## I Luddisti

Dei secondi, non condivido l'idiosincrasia verso tutto quello che sa di macchina e ancora di più la netta separazione tra mondo digitale e mondo “vero” (complesso, affettivo, etc).

Sarà perché la mia vita professionale si è giocata tutta nello spazio tra questi due mondi: non solo perché sono nato in uno e mi sono trasferito nell'altro, ma perché il dialogo tra questi due mondi mi sembra interessante, anzi fondamentale.

O ancora: perché finisco, oggi, per non vederla più, questa differenza. C'è un mondo solo e noi ci siamo dentro.

## I Disillusi

Non faccio nemmeno parte dei Disillusi, quelli che rimpiangono i bei tempi andati, quando in classe c'erano i Commodore 64. Che non si ritrovano nelle "nuove tecnologie" (social, mobile, video) perché non sono più quelle che gli erano familiari.

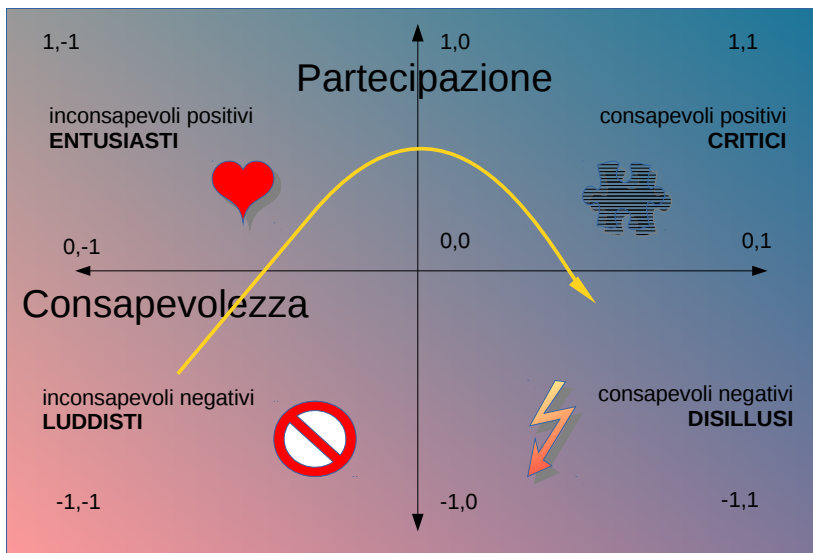
Penso che mantenere un atteggiamento di curiosità critica – insieme a una disponibilità di fondo ad imparare – sia fondamentale in ogni momento della propria vita professionale.<sup>9</sup>

Nel disegno che vedete più avanti, ho schematizzato queste quattro posizioni secondo due direttive: la consapevolezza e la partecipazione.

Naturalmente è solo uno schema per pensare. Non sono partiti così chiusi: aggiungo una curva che disegna il passaggio da un partito all'altro.

---

<sup>9</sup> Su questo vedi: <http://steve.lynxlab.com/?m=201608>



Provo a dettagliare le ragioni di questa posizione esterna esaminando alcuni elementi della proposta di Coding corrente, partendo dagli obiettivi. Ma prima, un intermezzo.

## *Primo Intermezzo: che significa opensource?*

Tutti i programmi che usiamo (da quelli sugli smartphone a quelli sul web) e che ci forniscono servizi (dalla gestione del traffico telefonico al controllo dei satelliti) sono simili: sono insiemi di istruzioni che i computer eseguono. Come ricette o sceneggiature.

La versione del programma che viene eseguita dai computer è, di solito, ottimizzata per essere veloce e senza errori. Questo processo (che si chiama "compilazione") genera una versione del programma che può essere letto dai computer, ma non dagli esseri umani.

Ma ogni programma è stato in origine scritto da qualcuno, in un certo linguaggio di programmazione (ce ne sono migliaia, come lingue umane). La prima versione del programma, quella originale, non ottimizzata, si chiama "codice sorgente". E' un testo, e come tutti i testi in teoria si può leggere – avendo le competenze per farlo - , modificare, copiare. Ma non necessariamente queste operazioni sono permesse dall'autore, o meglio da chi detiene il copyright sul programma. Normalmente, insieme al programma si riceve una licenza che specifica cosa non si può fare con esso: non si può copiare, non si può analizzare, non si può modificare. E' il caso, ad esempio, di Microsoft

Windows o di MS Office, ma anche della maggior parte delle app sui nostri smartphone, anche su quelli Android.

Alcuni programmi invece sono distribuiti con una licenza che permette esplicitamente di fare queste operazioni. Sono i programmi con licenza "aperta". Un esempio è Linux, che è un sistema operativo come Windows, o Libre Office, che è l'omologo di Microsoft Office.

Quando si parla di software libero si fa spesso confusione tra concetti diversi, come per esempio la gratuità e la libertà.

Avrete senz'altro letto da qualche parte il termine Creative Commons (ad esempio, i materiali didattici di Code.org sono rilasciati con questa licenza).

**Creative Commons** è una licenza che si applica a documenti (testi, audio, immagini, video). E' un'idea nata all'inizio di questo millennio da Lawrence Lessing, un professore di diritto dell'Università di Stanford, e si ispira ai "commons", cioè ai terreni che nella tradizione normativa anglosassone sono "comuni", dove tutti possono pascolare o tagliare legna. Questo concetto è trasferito dalla terra all'informazione.

L'altra caratteristica delle licenze CC è che sono modulari, cioè l'autore può decidere quali diritti concede all'utilizzatore.



**GPL** (General Public License), BSD, MIT, Apache, Mozilla, EUPL etc sono invece licenze che si applicano principalmente al codice sorgente. All'interno delle licenze aperte per il codice sorgente ci sono due grandi classi: le licenze ufficialmente approvate dalla Open Source Initiative e quelle che soddisfano i requisiti della Free Software Foundation .

**Public Domain** si riferisce a software (o a contenuti) che non hanno un autore conosciuto, o per cui l'autore ha rinunciato ai propri diritti.

**Freeware** invece si riferisce esclusivamente a software gratuito, ma non dice nulla sulla licenza o sulla disponibilità del codice sorgente. Molti programmi didattici sono freeware, ma non hanno una licenza "open". Significa che sono gratuiti, ma non che sono aperti. Se fossero stati rilasciati con licenza GPL qualcuno avrebbe potuto migliorarli o tradurli in altri linguaggi, per altri sistemi operativi.

Ma anche un linguaggio può essere "open"?

Sì, se:

- La sua grammatica e il suo lessico sono disponibili per lo studio

- L'interprete e/o il compilatore sono aperti e modificabili (rilasciati con una licenza che lo permette)
- Non si richiedono programmi non liberi per il suo utilizzo

## 5. A che serve il Coding?

Tutti d'accordo, il Coding è divertente. Ma a che serve? Un'attività didattica non può essere raccomandata solo perché è “carina” e facile, perché docenti e discenti si divertono. Deve avere uno o più obiettivi. Quali?

Ne ho raccolta una lista disordinata prendendo qua e là; alcuni secondo me sono condivisibili, altri meno. Alcuni sono molto alti, e forse poco raggiungibili; altri del tutto pratici ma chissà quanto sostenibili nel tempo. Non sono omogenei e anzi sullo sfondo si intravedono ideologie molto diverse.

Ma è fondamentale che chi organizza l'attività ce li abbia chiari, altrimenti si muove casualmente, spreca tempo ed energie e rischia pure di fare danni.

### 1. Quello di sviluppare il Computational Thinking

La questione è spinosa. Intanto non è facile definire il pensiero computazionale. Il termine è stato usato da Papert ma è diventato famoso in seguito ad un breve scritto di Jeannette Wing,<sup>10</sup> Microsoft Research / Carnegie Mellon University.

---

10

<http://www.cs.cmu.edu/afs/cs/usr/wing/www/publications/Wing06.pdf>

Computazionale significa “calcolabile”. Una funzione è computabile se esiste un algoritmo – cioè una serie di passaggi predefiniti, eseguibili da una macchina senza necessità di intervento esterno – che la calcola in un tempo finito. Per chi volesse saperne di più, il libro di Batini citato più sotto lo spiega con un dettaglio maggiore e con termini corretti.

Ma insomma è chiaro che il pensiero computazionale non è sinonimo di “pensiero logico” o di “concetti base dell'informatica”. Il pensiero computazionale ha a che fare con obiettivi, risorse e vincoli. Imparare il pensiero computazionale serve ad affrontare problemi con un approccio molto concreto che tende a evitare quelli irrisolvibili, a individuare le soluzioni che presumibilmente siano praticabili con le risorse disponibili oggi, ma anche quelle sostenibili nel tempo. Se si impara a valutare i problemi, a stimare le risorse e a definire i vincoli con i computer, poi si potranno applicare le competenze apprese anche in situazioni diverse.

Però a volte sembra che si faccia confusione tra un'interpretazione che abbiamo chiamato “culturale” e una “metacognitiva”:

“[i]l lato scientifico-culturale dell'informatica, definito anche pensiero computazionale, aiuta a sviluppare competenze logiche e capacità di risolvere problemi in modo

creativo ed efficiente, qualità che sono importanti per tutti i futuri cittadini”.

Non è proprio quello che dice Wing. Ma in ogni caso, c'entra poco con la programmazione.

Non si scrivono programmi solo per risolvere problemi, e programmare non è solo implementare algoritmi in un certo linguaggio di programmazione. L'ideazione (appunto, la creatività) serve quanto il rigore, altrimenti staremmo da anni a risolvere gli stessi problemi in maniera più efficiente.

Inoltre confesso che mi pare discutibile l'idea che sia davvero desiderabile introdurre il pensiero computazionale come competenza base, come maniera generale di affrontare la vita. Affidarsi solo al pensiero computazionale mi pare indizio di una visione davvero molto funzionalista del mondo.

In ogni caso: tanti anni fa un ricercatore del CNR, Giovanni Lariccia, proponeva nelle scuole un approccio didattico all'informatica senza computer (“carta e matita”).

Lo faceva sia perché di computer, onestamente, ce n'erano pochi, sia perché così era possibile collegare l'informatica all'esperienza quotidiana. Varrebbe la pena di andarsi a rileggere i suoi scritti (li trovate in fondo a questo testo).

## **2. Quello di preparare le nuove generazioni ai nuovi lavori?**

Questa è una delle argomentazioni che sembra più difficile da confutare.

Negli USA, entro il 2022 ci saranno 2.600.000 posti di lavoro nel settore dell'informazione; di questi, 750.000 per i programmatori, con una crescita del 22,8 %.<sup>11</sup> Con le parole di un articolo di qualche anno fa di Rebecca Lindegren<sup>12</sup>

“This means that U.S. companies would be forced to outsource valuable coding jobs to India, China, Eastern Europe, and other countries with growing IT sectors, while thousands of Americans remain unemployed or stuck in low-skilled, low-wage positions”

Quindi bisogna cominciare subito a preparare le future generazioni di programmatori. Purtroppo (o per fortuna) non c'è nessuna connessione diretta tra un ambiente educativo come Scratch e gli ambienti di programmazione professionali (come quello mostrato prima). E vista la distanza che c'è tra concetti, modelli, strumenti di dieci anni fa e quelli di adesso, la pretesa di insegnare oggi quello che sarà utile tra dieci anni è davvero troppo ambiziosa.

---

11 <http://www.bls.gov/news.release/pdf/ecopro.pdf>

12 <http://opensource.com/education/13/4/teaching-kids-code>

Fra l'altro: l'informatica non è solo programmazione. Ci sono (forse) milioni di posti di lavoro in attesa nel comparto informatico, là nel 2022, ma solo un terzo circa è riservata ai programmatori propriamente detti, mentre la maggioranza è lasciata a tutti gli altri lavoratori del settore (analisti, progettisti, sistemisti, grafici, esperti di reti, di sicurezza, commerciali, docenti, installatori, ...).

Ci possono anche essere ragioni un po' meno "nobili". Per esempio, una generazione di ragazzini che sono in grado di produrre in poche ore un'app con un certo linguaggio, su un certo sistema operativo, significa da un lato un serbatoio immenso da cui andare a pescare i migliori developers senza doversi assumere l'impegno e la responsabilità di formarli adeguatamente; e dall'altro un enorme mercato futuro per quelle app, per quel sistema operativo...

### **3. Quello di far costruire ai ragazzi delle storie in maniera alternativa alla scrittura?**

Bene, è un'attività molto utile e interessante, ma che si può fare benissimo in altri modi, dal disegno al teatro, dai fumetti al video. Perché proprio col coding? In qualche modo, però, que-

sta è la motivazione che condivido di più, ma per un motivo diverso.

Programmare – progettare e scrivere qualsiasi programma – è un altro modo di raccontare una storia: inventare un contesto, degli attori, un plot.

Sapere creare delle storie (o saperle smontare) è una competenza significativa in tanti campi, e mostrare come questi campi non siano poi così lontani è un obiettivo sensato.

Un po' paradossalmente: se a scuola si insegnasse diffusamente e coerentemente a costruire storie, allora farlo scrivendo un programma sarebbe un'applicazione utile.

Però di attività di costruzione in classe di problemi di matematica, o di scenari storici alternativi, non ne vedo tante in circolazione.

"Inventare le storie" mi fa subito pensare alla Grammatica della Fantasia di Gianni Rodari:

"[...] sebbene il Romanticismo l'abbia circondato di mistero e gli abbia creato attorno una specie di culto, il processo creativo è insito nella natura umana ed è quindi, con tutto quel che ne consegue di felicità di esprimersi e di giocare con la fantasia, alla portata di tutti"

Qui la creatività viene riconosciuta come universale, naturale (ma educabile) e soprattutto possibile fonte di soddisfazione



personale. Siamo in un altro mondo, spaziale e temporale, rispetto alla creatività come "furbizia", come ingegno applicato a risolvere un problema. Tuttavia questi due sensi di creatività sono evidentemente apparentati e bisognerebbe ricordarsi di tenere conto di entrambi.

#### **4. Quello di vaccinare i giovani?**

Contro quale virus? Contro lo sfruttamento indebito dei loro dati raccolti da un'app o da un social network system, o contro la pratica di spacciare come utilities gratuite programmi e siti che hanno scopi diversi da quelli dichiarati.

Ad esempio, sempre con le parole di Rebecca Lindegren:

“Children’s personal and professional lives will increasingly be shaped by computer programs. Without the ability to code, they will become passive consumers at the mercy of programmers working for technology giants, unable to construct or meaningfully interact with the virtual reality that surrounds them”.<sup>13</sup>

Questo passaggio dell’articolo è, a mio avviso, il più interessante. Senza la capacità di programmare, i bambini diventeranno passivi consumatori etc etc. Con la capacità di programmare

---

<sup>13</sup> <http://opensource.com/education/13/4/teaching-kids-code>

invece saranno vaccinati e potranno interagire significativamente con il mondo virtuale che li circonda.

Siamo d'accordo. Ma va definito cosa intendiamo per “capacità di programmare”. Un’attitudine? Un’esperienza, anche limitata? Una competenza specifica e verificata da terzi?

Stiamo parlando della buona abitudine di leggere il codice sorgente di ogni programma che si utilizza? Della curiosità verso ogni nuova soluzione che viene presentata, curiosità che non si contenta di un’etichetta o di una descrizione ma vuole arrivare a capire come funziona oggi e come funzionerà domani? O della capacità di progettare, sviluppare e mantenere soluzioni alternative?

Sono “capacità” completamente diverse. Si raggiungono, e si perdono, in tempi diversi e in modi diversi. Alcune di queste non sono generiche, ma possibili solo in connessione con certi contesti tecnologici e legali, primo fra tutti quello dell’apertura del codice sorgente (cioè l’opensource).

La seconda cosa da notare è che la possibilità di interagire pienamente con la realtà (virtuale, nel senso dell’insieme di dispositivi, reti, server, ...) non sembra dipendere solo da queste competenze. Anche qui, andrebbe forse ricordato che, oggi molto più di ieri, ognuno di noi ha comprato già preinstallati o consentito a installare sui propri dispositivi digitali – pc, tablet, smartphone, televisori, frigoriferi,... – centinaia di programmi

del cui funzionamento effettivo non possiamo sapere quasi nulla, se non quello che esplicitamente ci dicono i produttori.

Il codice sorgente di questi programmi (che a volte chiamiamo “applicazioni” o amichevolmente “app” per farceli sembrare meno complessi e e pericolosi) non è disponibile per la lettura o la modifica. Sapere programmare non aiuta minimamente a evitare che raccolgano i nostri dati e ne facciano un uso non previsto (da noi). Sapere programmare non ci permette di evitare di usarli: alzi la mano chi si può permettere di non avere un account gmail o una pagina FB. Senz’altro non ci aiuta a modificarli, a impedire che svolgano azioni se non illecite, almeno non gradite.

Interagire significativamente con gli altri tramite app e reti, ricevere e fornire dati – filtrandoli – richiede delle competenze, che oggi fanno sicuramente parte di quelle di base di ogni cittadino. Ma allora non è sufficiente un pomeriggio di manipolazione di Scratch, serve anche qualche informazione in più. Informazione che in effetti né la scuola dell’obbligo, né quella superiore, né l’università consegnano. Occorre parlare anche di altro: di codice aperto e licenze, di privacy, di diritti, di mercato dei dati, di equilibrio tra gratuità e sostenibilità. C’è da affrontare un discorso vasto sul ruolo egemone di Google e di Facebook, sul diritto all’anonimato o all’oblio.

Niente che non si possa spiegare ai bambini, con qualche attenzione; ma vorrei almeno vedere qualche passo in questa direzione critica *mentre* si parla di Coding.

### **5. Quello di dare ad ognuno le competenze minime per scriversi un programma per risolvere i propri problemi?**

Che so: una app per ricordarsi di prendere le medicine. Una che mi dice come devo riciclare il tetrapak. Questo sarebbe un bell'obiettivo pratico, utile almeno quanto far sì che uscendo dalla scuola un ragazzo sappia sostituire un interruttore, curarsi una ferita, fare la dichiarazione dei redditi o cambiare l'olio alla macchina (temo che non sia così). Un recupero delle Applicazioni Tecniche. Una riappropriazione dell'universo di macchine in cui siamo immersi.

Ma allora il linguaggio da proporre dovrebbe essere qualcosa in grado di produrre programmi che girano su PC, smartphone, tablet, su sistemi operativi diversi. E di nuovo, occorre quanto meno spiegare perché con certi elettrodomestici si può "parlare" (perché adottano uno standard di comunicazione aperto) e con altri no.

## **6. Quello di affrontare la didattica di qualsiasi disciplina in maniera costruttiva?**

Nessuno più di me sarebbe d'accordo, e tra l'altro questo è uno degli obiettivi dichiarati di Scratch. Qui però il centro non è la tecnologia, è la didattica. Che un ambiente di apprendimento debba essere aperto, modificabile, che si capisca meglio qualcosa se si sperimenta e si costruisce, dovrebbero essere tutte premesse che una didattica moderna accetta senza troppi problemi (ma probabilmente nei fatti le cose stanno diversamente).

Ma allora deve essere chiaro che il punto non è né divertirsi, né imparare le iterazioni, ma comprendere il funzionamento di una cellula o la struttura di un romanzo. Cioè: l'obiettivo è la didattica della biologia, non quella della programmazione.

E direi anche che non serve per forza partire sempre da zero: si potrebbe iniziare interagendo con simulazioni già preparate e poi modificarle, estenderle, riapplicarle altrove. Ci sono una quantità di siti web dove è possibile provare programmini del genere.

Come si vede, di obiettivi possibili ce ne sarebbero molti. Alcuni raggiungibili, altri meno.

Ma servono revisioni dei programmi (quelli scolastici) e risorse umane preparate.

Ci sono?

## 6. Chi può insegnare il Coding?

Questo è un tema delicato. Si va a toccare non solo la preparazione dei docenti, ma in generale il rapporto col digitale, gli aspetti sociali, la reputazione, la divisione tra umanistico e scientifico.

Sul sito di “Programma il futuro” si dice che gli strumenti adottati sono “[...] progettati e realizzati in modo da renderli utilizzabili in classe da parte di insegnanti di qualunque materia. Non è necessaria alcuna particolare abilità tecnica né alcuna preparazione scientifica”. Non si parla degli studenti, ma dei docenti.

Strumenti scelti perché sono facili da insegnare, non da imparare. Cioè: invece di formare i docenti, si sceglie un ambiente che non necessita di nessuna attività previa, in modo che tutti i docenti possano utilizzarlo. Come si arriva a proporre un metodo e un set di strumenti in funzione non della specificità e dell'adeguatezza a uno scopo ma in funzione del numero di operatori che sono in grado di gestirli? Certo così si fa prima e si spende di meno.

Ma questo è anche un passo all'interno di un processo di "democratizzazione" dell'informatica molto più generale.

In breve, mentre gli insegnanti di matematica sono per qualche motivo i naturali alfieri di ogni introduzione dell'informatica a scuola, gli insegnanti di materie umanistiche si sono sempre sentiti esclusi dal mondo digitale. Perché era troppo complesso per loro.

Creare un programma con un linguaggio come C o Java, o anche con il meno blasonato Javascript, o persino creare a mano una pagina HTML+CSS, è un'attività che richiede troppe competenze.

Alcuni ne hanno sofferto, altri se ne sono fatti una ragione. Molti hanno alzato come una bandiera la loro impermeabilità al digitale (“ah io di queste cose... poi insegno Inglese”).

Poi sono arrivati i CMS, e chiunque oggi può creare un sito web – magari con WordPress – senza avere la più pallida idea di cosa sta facendo. E l'onnipresente MS PowerPoint, con la possibilità di creare degli slideshow per mostrarli in classe, complice la LIM. Poi è arrivato Code.org.

Il Coding (inteso come attività all'interno di un ambiente visuale come Scratch) è qualcosa che può fare, e insegnare, chiunque. I concetti informatici da comprendere sono pochi: le variabili, le istruzioni condizionate, i cicli.

E' la grande rivincita del docente non informatico.



Ora io vorrei guardare un po' in avanti.

Cosa succede dopo che si è costruita la storia del lupo e dell'agnello con i bambini? Ci si ferma qui?

O questo è solo il primo passo – come sembrerebbe a giudicare dai contenuti didattici presenti in Code.org – di un percorso che dalla programmazione visuale porta a guardare il codice sorgente, e poi a scriverlo, e poi alla conoscenza di altri linguaggi ed ambienti diversi?

Se è l'inizio di un percorso, allora “non avere abilità tecniche e preparazione scientifica” ma essere capaci di fare da tutor non basta. Occorre, ad esempio, sapere che significa codice sorgente, conoscere la differenza tra compilare e interpretare, saper assegnare una licenza o dare un permesso d'autore...

Affidereste l'insegnamento della letteratura italiana ad un docente che ha solo vaghe conoscenze in questo campo? No? Allora anche i docenti di Coding devono essere adeguatamente formati.

Sempre che ne abbiano voglia.

Prima che partano delle obiezioni facili: non sto dicendo che basta essere informatici per insegnare il coding ai bambini, o che solo gli informatici possono farlo.

Sto dicendo che oltre a tutte le altre competenze, che do per scontate (pedagogiche, organizzative, psicologiche, valutative) servono anche quelle informatiche, e queste come quelle non si improvvisano, vanno imparate.

E' faticoso? Senza dubbio. E' più facile far finta di niente? Altrettanto.

Ma dovendo imparare un linguaggio, quale?

## *Secondo Intermezzo: le differenze tra linguaggi*

Leonardo Fibonacci è il matematico pisano che ha fatto conoscere la numerazione araba in occidente. Ma è più noto per la "successione di Fibonacci", cioè la serie di interi:

1,1,2,3,5,8,13,21,34,55,89,144,...

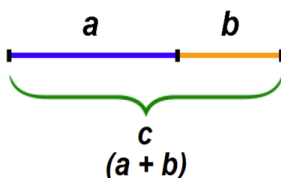
La formula per il calcolo dell'ennesimo numero di Fibonacci è:

$$F_n = \frac{\phi^n}{\sqrt{5}} - \frac{(1 - \phi)^n}{\sqrt{5}} = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}.$$

con:

$$\phi = \frac{1 + \sqrt{5}}{2} = 1,6180339887 \dots$$

in cui  $\phi$  è la sezione aurea



Un po' complicato? Beh, utilizzando un linguaggio di programmazione si può costruire la sequenza senza fare uso della sezione aurea e della formula matematica, semplicemente simulando il processo passo dopo passo.

$$F_n = F_{n-1} + F_{n-2}$$

$$F_1 = 1$$

$$F_2 = 1$$

$$F_3 = F_1 + F_2 = 1 + 1 = 2$$

$$F_4 = F_2 + F_3 = 1 + 2 = 3$$

$$F_5 = F_3 + F_4 = 2 + 3 = 5$$

...

Così la programmazione serve a risolvere un problema anche quando non se ne possiede una visione perfetta. In questo senso, programmare è un ausilio formidabile per l'apprendimento per scoperta.

E ci sono tanti modi diversi per arrivare ad una soluzione.

Alcuni esempi scritti in linguaggi diversi, tratti da <http://www.scriptol.com/programming/fibonacci.php>, possono servire a dare un'idea della differenza tra linguaggi di programmazione.

Basic	<pre>x = 1 y = 1 n = 100 FOR x = 1 to n   z = x + y   x = y   y = z   PRINT z + 1 NEXT x</pre>
C	<pre>int fib(int n) {   int first = 0, second = 1;   int tmp;   while (n--) {     tmp = first+second;     first = second;     second = tmp;   }   return first; }</pre>
LUA	<pre>fibs = { 1, 1 } setmetatable(fibs, {   __index = function(fibs,n)     fibs[n] = fibs[n-2] + fibs[n-1]     return fibs[n]   end })</pre>

RUBY	<pre>def fibonacci( n )   return n if n &lt;= 1   fibonacci( n - 1 ) + fibonacci( n - 2 ) end</pre>
PERL	<pre>sub fibo; sub fibo {\$_ [0] &lt; 2 ? \$_ [0] : fibo (\$_ [0] - 1) + fibo (\$_ [0] - 2)}</pre>
LOGO	<pre>to fibonacci :N   ifelse :N &lt; 3 [output 1] [output sum fibonacci :N - 1 fibonacci :N - 2] end</pre>

Come si vede, pur essendo tutte rappresentazioni dello stesso algoritmo, ci sono differenze superficiali enormi: dalla sintassi all'uso delle parentesi e della punteggiatura, ai termini usati per introdurre la funzione (*def*, *sub*, *to*). Però c'è chi sostiene che una volta imparati i primi due o tre linguaggi, poi è tutta discesa.

Come per le lingue naturali. E tutto sommato, una parentela tra apprendimento delle lingue naturali e dei linguaggi artificiali si può tentare di tracciare. Più in generale: ci sarebbe solo da guadagnare nello studiare i linguaggi di programmazione con gli stessi concetti con cui si studiano le lingue: evoluzione, derivazione, numero di parlanti, letteratura, stili...

## 7. Che linguaggio?

Dicevamo: quale linguaggio scegliere? Mi pare che in Italia ci sia una totale, incondizionata adesione alla (validissima) proposta del MIT:

Coding = Scratch

Il sito di Scratch<sup>14</sup> ospita ad oggi oltre 16 milioni di progetti.

La domanda è: ma sappiamo che cos'è Scratch, da dove viene, come funziona, che limiti ha?

Scratch nasce al MIT nel 2003 con un finanziamento di ricerca della National Science Foundation, in collaborazione con diverse altre università statunitensi (Pennsylvania, Harvard, Washington e altre).

Nei ringraziamenti, vengono citati Seymour Papert e Alan Kay, ovvero l'autore del LOGO e uno degli inventori della programmazione orientata agli oggetti e delle finestre, nonché ideatore di Squeak / Etoys.

Perché Scratch è stato scelto da Code.org (e quindi dal MIUR per fondarci Programma il futuro)? Cosa ha di particolare?

---

<sup>14</sup> <https://scratch.mit.edu/>

Gli elementi che saltano agli occhi sono almeno due: il fatto che Scratch vada usato online e l'uso dei blocchi grafici che rappresentano istruzioni per pilotare un robot. Non sono caratteristiche uniche di Scratch, anzi vedremo che sono piuttosto comuni, ma sicuramente hanno contribuito al suo successo. Ed entrambe meritano analisi e discussione.

1. Scratch è una “online community where children program and share interactive stories, games, and animations”. Bisogna quindi essere necessariamente connessi per programmare. Significa che per iniziare ad usarlo non serve scaricare, installare, aggiornare, ma basta aprire un browser e collegarsi ad un sito, dove si trovano modelli, esempi, lavori di altri gruppi.

In realtà esisteva un editor offline, basato su Adobe Air. E' una scelta strana per il MIT; né Adobe Air né l'editor offline sono OpenSource<sup>15</sup> (a differenza di quasi tutti gli altri ambienti dello stesso genere). Non ho idea di quanti usino la versione offline, talmente è più facile utilizzare la versione online. Ma certo questa spinta alla connessione continua, in un Paese in cui la banda larga per tutti - soprattutto in provincia - è ancora un obiettivo lontano, qualche difficoltà la crea.

---

15 A dire il vero, su Github esiste una versione OpenSource dell'editor offline, che però è comunque basato su Flash, che non lo è.



In ogni caso c'erano, e ci sono ancora, anche altre comunità dello stesso genere. Quasi ognuno degli ambienti di programmazione per bambini ha un team di sviluppo che mantiene un sito con esempi, suggerimenti, guide. La comunità intorno a Scratch è la più recente, probabilmente la più attiva, non necessariamente la più interessante. Potrebbe valere la pena andare a curiosare anche nelle altre.

2. Il fatto che non sia necessario scrivere codice sorgente (il che, come abbiamo notato sopra, è piuttosto curioso all'interno di un progetto di "Coding").

La questione non è irrilevante. Uno dei passaggi chiave nella storia degli ambienti di programmazione per bambini è proprio il passaggio dalla scrittura di codice alla programmazione visuale. Significa che invece di scrivere "if questo then quest'altro" il programmatore deve trascinare oggetti grafici, disporli in un certo ordine, e scrivere solo dati all'interno di buchi (le variabili). La sintassi è affidata alla posizione degli oggetti, la semantica alla scrittura. Ci sono enormi vantaggi: il rischio di commettere errori di sintassi è annullato, la struttura del codice è rappresentata in forma visuale ed è più comprensibile con uno sguardo d'insieme.

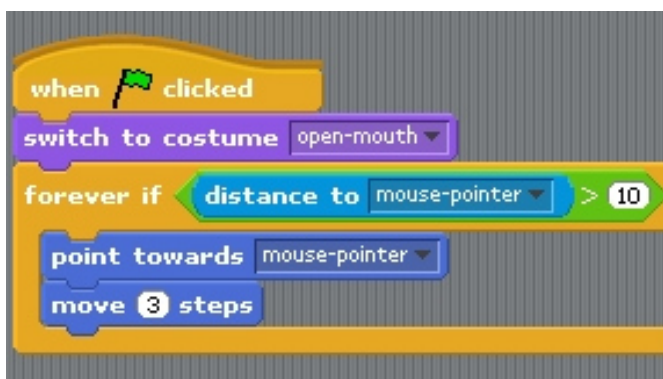
Si introduce però uno iato forte tra l'attività presente (visuale) e quella futura (scrittura). Infatti gli ambienti di programmazione visuale professionali sono piuttosto pochi, dedicati di solito

a domini specifici come il multimedia e, nel 99% dei casi, un programmatore scrive o modifica codice scritto.

Ci sono degli evidenti vantaggi nell'uso della scrittura: un codice scritto in qualsiasi epoca e in qualsiasi linguaggio può essere modificato con un banale editor di testo (se la licenza lo permette). E abituarsi a leggere e rileggere - vale per un testo scritto in una lingua naturale come per un codice sorgente - non è un'attività poco utile nella vita.

In ogni caso, le caratteristiche principali di Scratch sono:

- Gratuito
- Interfaccia grafica pensata per bambini
- Linguaggio imperativo, guidato dagli eventi
- Occorre registrarsi per utilizzarlo
- Non del tutto opensource



Attenzione: Scratch non è pensato per bambini che non sanno leggere, perché i blocchi hanno comunque un nome. Esistono però dei progetti collegati (ScratchJr,<sup>16</sup> Scratch Blocks<sup>17</sup>) di ambienti davvero preverbali. O quasi: come vedete qui sotto, i numeri ci sono.



Curiosando all'interno dello stesso sito di Scratch<sup>18</sup> o con una breve indagine su Wikipedia<sup>19</sup> si scopre che di ambienti di programmazione “facili”, educativi, visuali, giocosi, ne sono esistiti, e ne nascono ogni giorno, una marea.

Non servono tutti esattamente allo stesso scopo, sono diversi per tipologia di linguaggio, espandibilità, possibilità di eseguirli su dispositivi diversi, licenze d'uso.

Alcuni sono dichiaratamente alternativi a Scratch (come ToonTalk), altri ne sono estensioni. Molti si interfacciano col

---

16 <https://www.scratchjr.org/about.html>

17 <https://github.com/LLK/scratch-blocks/wiki>

18 [https://wiki.scratch.mit.edu/wiki/Alternatives\\_to\\_Scratch](https://wiki.scratch.mit.edu/wiki/Alternatives_to_Scratch)

19 [https://en.wikipedia.org/wiki/Visual\\_programming\\_language](https://en.wikipedia.org/wiki/Visual_programming_language)

mondo esterno (Lego, Arduino, Raspberry PI) nello spirito della tartaruga-robot del Logo.

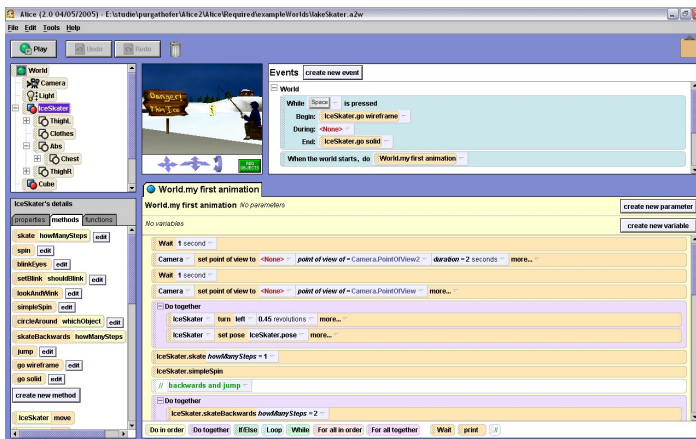
Hanno tutti in comune l'obiettivo e il target: rendere possibile la programmazione anche a bambini che non hanno nessuna conoscenza teorica dell'informatica.

Tutti questi ambienti condividono l'idea che sia più facile iniziare a programmare senza scrivere codice, e che questa maggiore facilità sia un vantaggio anche in termini dell'acquisizione di competenze più avanzate nel futuro. E su questo presupposto dovremo tornare più avanti. Per ora limitiamoci a osservarne qualcuno.

*Nota: sarà un caso se esistono progetti alternativi a Scratch che vengono dall'Austria, dalla Francia, dalla Germania, dall'Ungheria, dall'India... e non dall'Italia?*

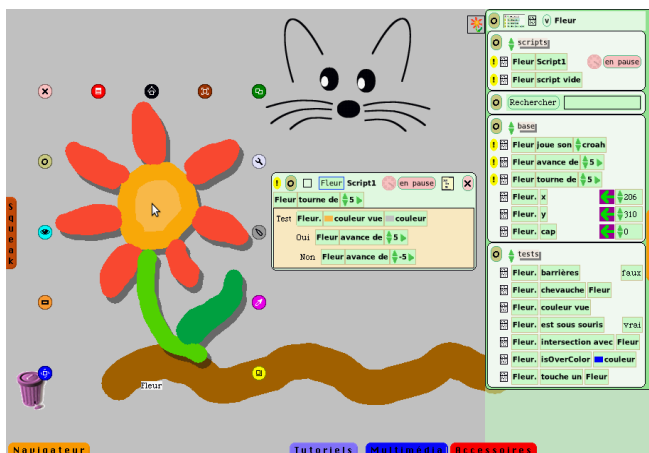
## Alice

- Alice è un ambiente per la creazione di storie e giochi in 3D, scritto in Java
- Creato presso la Carnegie Mellon University a partire dal 1999
- Drag and drop; la versione 3 rappresenta un'introduzione verso linguaggi più "seri", come Java
- Free ma non opensource



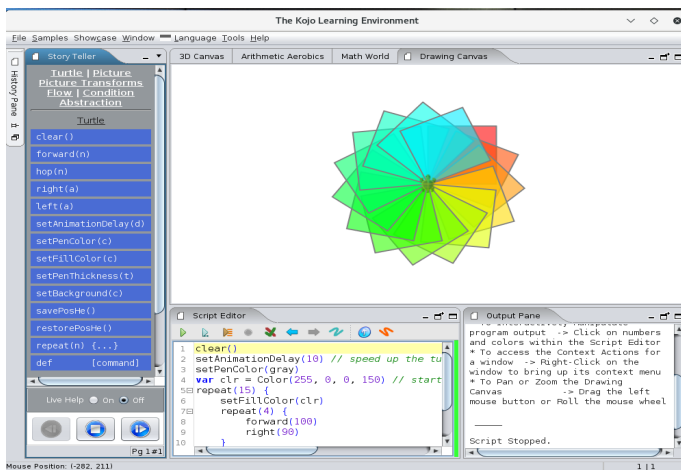
## Squeak! /Etoys

- Squeak! è un moderno interprete SmallTalk scritto in SmallTalk
- Nato nel 1996 da un gruppo di sviluppo Apple
- Etoys è stato progettato da Alan Kay presso la Disney
- Nel 2006 Etoys è stato incorporato nei computer del progetto OLPC (One Laptop Per Child)
- E' (oggi) opensource



## Kojo

- Kojo è proposto da una fondazione indiana, Kogics<sup>20</sup>
- E' ispirato fortemente da Logo ma anche da altre idee (il linguaggio Processing, Geometer Sketchpad)
- Il linguaggio usato è Scala<sup>21</sup> che è un moderno linguaggio funzionale
- E' opensource e scaricabile liberamente

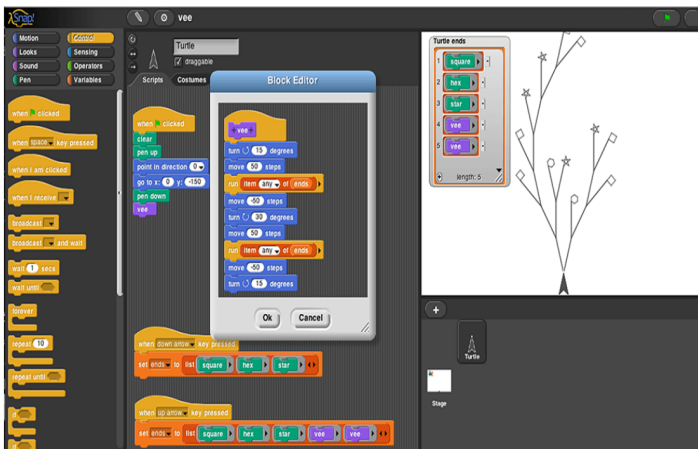


<sup>20</sup> <http://wiki.kogics.net/kogics-foundation>

<sup>21</sup> <http://www.scala-lang.org/>

## Snap!

- Snap!<sup>22</sup> è una riscrittura di Scratch fatta da Jens Möning e Brian Harvey all'Università di California a Berkeley
- Basato sul linguaggio Scheme<sup>23</sup> (derivato da LISP)
- Fino alla versione 3 è stand-alone, dalla 4 online
- E' opensource (AGPL)
- I programmi possono essere riusati in vari modi (per esempio, come app mobile <http://snapp.citilab.eu/>)
- Si può passare dalla modalità visuale a quella codice; il linguaggio può essere esteso aggiungendo nuove strutture



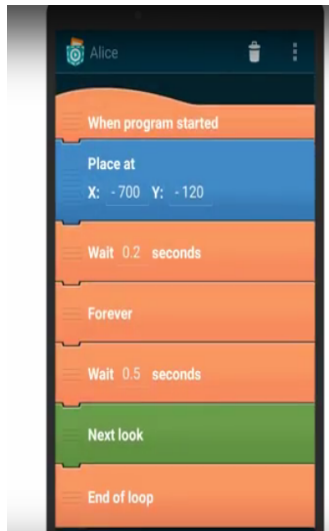
<sup>22</sup> <http://snap.berkeley.edu>

<sup>23</sup> <https://it.wikipedia.org/wiki/Scheme>



## Catrobat

- Catrobat /Pocket Code<sup>24</sup> è un'iniziativa ispirata da Scratch nata presso l'Università di Graz, Austria, con ampio supporto di Google e poi di Samsung.
- Catrobat è un ambiente visuale per creare app, che gira dentro gli smartphone.
- Permette di controllare Arduino, Raspberry PI, Lego NXT, NFC
- E' opensource

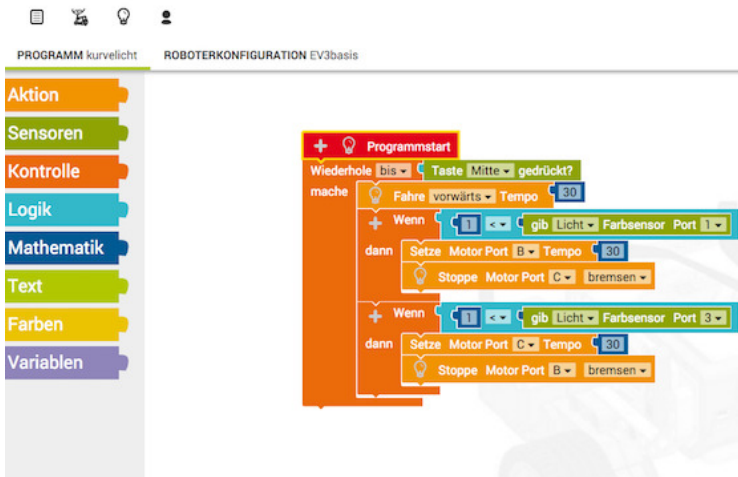


---

<sup>24</sup> <http://www.catrobat.org/>

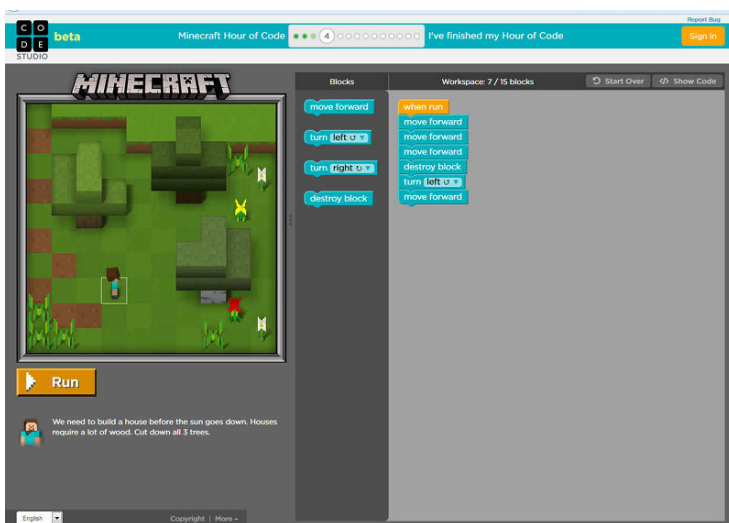
## OpenRoberta

- Sviluppato dal Fraunhofer Institute for Intelligent Analysis and Information Systems (IASI) a Sankt Augustin (Germania)
- Fondi Google (1 Milione di € !) e sostegno Lego
- Ambiente online per programmare il robot EV3 della Lego con il linguaggio NEPO (New Easy Programming Online)
- E' opensource



## Minecraft: Education Edition

- Versione online<sup>25</sup> del gioco Minecraft, sviluppato da una società svedese comprata da Microsoft nel 2014
- L'interfaccia di programmazione è Google Blockly, che permette di passare dai blocchi a Javascript
- Permette di collegarsi ad altre classi
- Gratis fino al 1 Novembre, poi a pagamento (5 \$ anno)
- E' oggetto di una campagna di formazione di Microsoft e (probabilmente) di un accordo con il MIUR
- Proprietario<sup>26</sup>



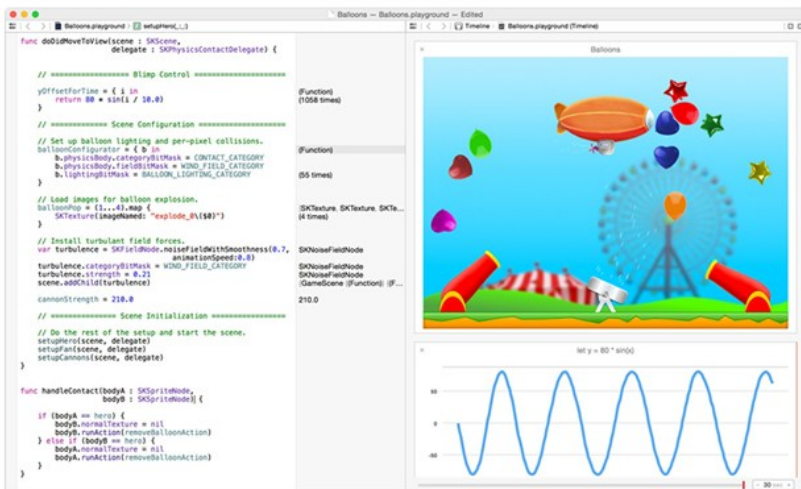
---

<sup>25</sup> <http://education.minecraft.net/>

<sup>26</sup> Da qualche giorno ne esiste una versione OpenSource, francese:  
<https://framinetest.org/>

## Apple Swift Playground

- Ambiente educativo basato su Swift,<sup>27</sup> linguaggio introdotto da Apple nel 2014
- Gratuito
- Opensource (Apache 2.0)



<sup>27</sup> <https://www.apple.com/swift/playgrounds/>

## ToonTalk

- Linguaggio visuale (l'unico tra quelli citati finora) ideato da Ken Kahn<sup>28</sup> che lo presenta come un figlio delle idee del Logo
- Modello di programmazione per vincoli, concorrente
- Ripreso recentemente e disponibile come libreria Javascript
- Opensource (GPL 3)

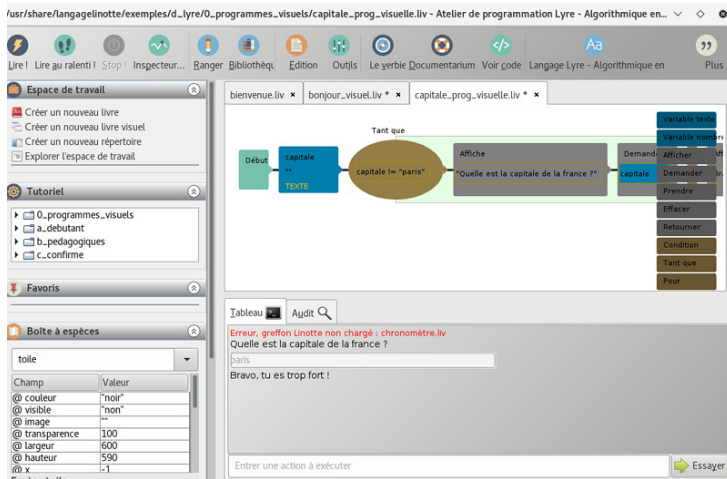


---

<sup>28</sup> <https://github.com/ToonTalk/ToonTalk/wiki>

## Linotte

- Orientato agli oggetti<sup>29</sup>
- Possibilità di passare dalla modalità visuale a quella testuale
- Estensibile
- Lessico e sintassi francese
- Opensource (GPL 3)



<sup>29</sup> <http://langagelinotte.free.fr/>

A proposito di lingue e Paesi: anche i linguaggi visuali utilizzano delle etichette ("repeat", "if/then/else"). Che nella versione originale sono in Inglese, come sono ricalcate sull'Inglese molte delle istruzioni dei linguaggi di programmazione tradizionali.

Per fortuna, praticamente tutti gli ambienti che abbiamo citato possono essere proposti ai bambini nella loro lingua naturale (Italiano compreso). Il che è una caratteristica degli ambienti educativi: devono essere flessibili, adattabili agli stili degli apprendenti.

Tuttavia, forse non tutti sanno che esistono dei linguaggi di programmazione NON basati sull'inglese. Sono linguaggi che hanno lessico e sintassi ispirati a lingue diverse (Cinese, Islandese, Arabo,...). La loro sola esistenza dimostra che la connessione tra lingua naturale e linguaggio di programmazione, tra come si pensa e come si programma, è molto forte.

Ne esistono anche di educativi, come quel Linotte ideato e sviluppato da Mounès Ronan, insieme con Lyra, il suo "atelier de programmation". Esistono quattro versioni di Linotte (originaria, semplice, più completa, con sintassi francese) ma si può passare facilmente da una all'altra.

Anche qui, la disponibilità del codice sorgente secondo una licenza libera permette, a chi ne ha voglia, di tradurre le eti-

chette e i messaggi anche in altre lingue oltre a quelle già disponibili.

Sarebbe bello che, una volta decisi gli obiettivi e preparate le risorse umane, una volta fissati i parametri per la scelta, si andasse a scegliere il linguaggio più adatto, tra tutti quelli disponibili, invece di scegliere sempre e solo quello più conosciuto. Per farlo, ovviamente, occorre saper distinguere tra modelli, linguaggi, licenze.

E perché bisogna registrarsi, andare di qua e di là sul web? Non si potrebbe mettere su un ambiente di apprendimento del coding sul sito della propria scuola?

Sì. A patto che la soluzione scelta sia scaricabile e opensource. Ad esempio Google Blockly,<sup>30</sup> che permette appunto di inserire un editor visuale di codice in una qualsiasi pagina web.

Un'ultima domanda: come si spiega la presenza, accanto alle maggiori università statunitensi (ma non solo) di industrie dell'IT com Microsoft e Apple tra quelli che propongono ambienti di Coding per i più piccoli?

---

30 <https://developers.google.com/blockly/>



Con le parole di Riccardo Meggiato, Wired:<sup>31</sup>

"Ma è il settore dell'educazione a formare, più dei giochi, i gusti tecnologici delle future generazioni. Uno strumento utilizzato a scopo educativo entra a far parte di una forma mentis. È il motivo per cui, chi ha qualche anno sulle spalle, da piccolo si massacrava con Sega Megadrive e ora non sfiora un videogame nemmeno da lontano, eppure continua a usare Word al posto di LibreOffice perché è sempre stato abituato a utilizzare il pacchetto Microsoft. I gusti in fatto d'intrattenimento cambiano, quelli in fatto di educazione e lavoro, semmai, evolvono."

Chiaro, no?

---

31 <http://www.wired.it/gadget/computer/2016/06/14/apple-rivoluzione-swift-playground/>



## 8. Il modello didattico dietro Scratch

L'autorità scientifica principale dietro Scratch, Mitch Resnick (LEGO professor al MIT Media Lab), parla degli obiettivi di Scratch in questi termini: “Quando qualcuno impara a programmare con Scratch impara allo stesso tempo importanti strategie per risolvere problemi, creare progetti e comunicare le proprie idee.” Le stesse idee sono espresse nella guida del 2011 pubblicata dall'Università di Harvard.<sup>32</sup> La guida è una miniera di idee e contenuti, declinati in 20 sessioni didattiche. Nell'introduzione si dice fra l'altro:

“Engaging in the creation of computational artifacts prepares young people for more than careers as computer scientists or as programmers. It supports young people's development as computational thinkers – individuals who can draw on computational concepts, practices, and perspectives in all aspects of their lives, across disciplines and contexts.”

Il pensiero computazionale è quindi visto come una maniera di affrontare i problemi della vita, attraverso concetti come se-

---

32 <http://scratched.gse.harvard.edu/sites/default/files/CurriculumGuide-v20110923.pdf>

quenze, cicli, parallelismo, eventi, condizioni, operatori e dati. Le pratiche che vengono incoraggiate come più adatte sono “essere iterativi e incrementali, verificare e correggere, riusare e mescolare, astrarre e modulare”.

Da un lato questa visione mi attrae, dall'altra devo confessare un po' di spavento. Davvero i problemi reali vanno affrontati in termini di algoritmi formalizzabili? Davvero le azioni nei complessi contesti quotidiani vanno regolate in funzione di condizioni, operatori e dati? I bambini devono giocosamente imparare a comportarsi nella vita come automi perfettamente informati?

Persino l'idea dell'introduzione giocosa alla programmazione, per preparare i futuri sviluppatori di cui avremo bisogno nei prossimi dieci anni, andrebbe esaminata un po' di più prima di essere accettata come una verità incontestabile. Non solo perché nel comparto informatico ci serviranno molte figure professionali diverse dagli sviluppatori; e non solo perché la programmazione richiede tanta creatività nell'inventare quanta abilità nel portare a termine l'idea.

Il punto centrale, il fulcro, è la strategia didattica.

Che “chi impara prima, impara meglio”, sembra essere un proverbio uscito dalla saggezza popolare, e quindi poco discutibile. Per diventare un musicista, un ballerino, un calciatore, un cantante, bisogna cominciare da bambini. Vale anche per le lingue straniere, vale anche per la matematica. Lo sappiamo per esperienza. Vale anche per la programmazione dei computer? Beh, qui troppi dati statistici non ci sono, dobbiamo procedere per analogia; e le analogie vanno tenute sotto controllo perché tendono a sfuggire.

Come si insegna ad un bambino una materia complessa, composta di tecnica e conoscenze, senza che le difficoltà impediscano i progressi?

Tradizionalmente, abbassando il livello della qualità richiesta, semplificando gli obiettivi, ma lasciando intatti gli elementi e le regole. Non si insegna ad una bambina di sei anni a suonare il violino con uno strumento con due sole corde, ma con un violino  $\frac{3}{4}$ , che ha la stessa complessità di uno  $\frac{4}{4}$ . Ma c'è una sterminata letteratura didattica composta per guidare lo studente dal facile al difficile. Non si semplifica il contesto, si modulano le richieste. La strategia didattica è lineare: c'è una gradazione infinita di modi di far vibrare una corda, e chi apprende procede lungo questo sentiero infinito. Limiti: la bambina si

annoia, soprattutto se non capisce dove la porterà questa lunga strada, perché i risultati iniziali non sono troppo incoraggianti. Finirà probabilmente per abbandonare.

C'è un altro modo, più moderno: si crea un ambiente “didattico”, semplificato, in cui elementi e regole sono ridotti rispetto all'originale. Un flauto con i tasti invece dei fori; un toy piano diatonico. Qui i risultati gradevoli si raggiungono presto, la motivazione è rafforzata. Non si inizia con la grammatica, ma con la semantica.

Questi giocattoli educativi non sono in continuità con il mondo reale. C'è una cesura netta: da un lato il giocattolo, dall'altro lo strumento vero, da un lato gli spartiti colorati, dall'altro i pentagrammi.

Questa strategia è senz'altro più efficace per iniziare, è più “democratica”, ha successo nella quasi totalità dei casi. Non serve a preparare musicisti professionisti, ma a comunicare l'amore per la musica (per esempio, io ho cominciato a suonare su un pianoforte giocattolo, e da allora amo disturbare il vicinato con ogni tipo di strumento – per questo vivo in campagna). E' molto chiaro che non c'è un passaggio graduale dal modo “semplificato” a quello “avanzato”. Sono due mondi diversi, che si assomigliano in alcune parti, ma che non sono in connessione diretta. Alcuni degli studenti arrivati ai confini di uno si

affacceranno sull'altro, si accorgeranno che la complessità è enormemente più elevata, ma decideranno di entrare lo stesso; altri si fermeranno nel primo mondo, come ho fatto io.

Tra gli ambienti di programmazione per bambini, però, alcuni sono stati pensati proprio come un raccordo, una via di mezzo tra i giocattoli e le cose serie. Sono ambienti dinamici, modificabili, evolutivi, in cui il bambino può iniziare in un contesto semplice e poi aggiungere complessità fino ad arrivare all'editor testuale di codice sorgente che è identico a quello del programmatore professionista. Questo è uno dei motivi per cui gli oggetti digitali sono più potenti di quelli fisici.

Ci sarebbe anche da affrontare il tema della pedagogia dell'errore. L'adagio "sbagliando si impara" non è solo un modo di consolarsi quando si è alla prime armi. Quando ci si imbatte in un errore si mobilitano molte più risorse di quando tutto procede per il meglio: si alza il livello di attenzione, si cerca di rivedere i passaggi, si sposta il focus dal particolare al generale e viceversa. Avere a disposizione un ambiente di apprendimento dove non è possibile sbagliare ha sicuramente l'effetto di tranquillizzare chi apprende, ma anche quello di impedire tutti questi processi di livello "meta".

Ora, quali sono gli obiettivi principali di Scratch? Almeno due: preparare futuri programmatori e iniziare i bambini al computational thinking.

E la strategia didattica dietro Scratch, di quale delle due categorie delineate sopra fa parte? Direi della seconda, visto che l'ambiente che si propone è giocoso, facile, divertente, ma non ha le stesse regole del “mondo adulto” della programmazione. Non si scrive e legge codice. Non ci si preoccupa della correttezza sintattica. Non ci si preoccupa dell'efficienza, della velocità, della comprensibilità, della standardizzazione, dello stile, etc etc. Non c'è niente di male, come non c'è niente di male in un toy piano. Ma solo nei fumetti Schroeder suona Beethoven con quello.

Il problema nasce proprio quando si immagina che entrambi gli obiettivi possano essere raggiunti con una sola strategia, ovvero quando questa differenza di approcci strategici non viene proprio percepita, cioè quando si pensa che iniziare a programmare con Scratch sia solo il gradino più basso di una scala che porta, gradualmente, alla produzione dei software che usiamo ogni minuto. E questo è tipicamente un errore di chi non ha una grande esperienza della programmazione.

Programmare non è solo *un po' più difficile* di creare un gioco con Scratch: è enormemente più complesso; come realizzare un



film o registrare un concerto non è solo un po' più difficile di girare un video con uno smartphone.

Personalmente ho visto tanti bambini (studenti, figli, figli di amici) giocare con ambienti di programmazione fatti apposta per costruire giochi. Di tutti questi, solo uno, che io sappia, ha avuto la voglia e la costanza di continuare. Gli altri, quando hanno capito che programmare è difficile, si sono arresi e sono tornati a giocare.



## 9. Come andrà praticato il Coding

Dicevamo che la maniera di organizzare il Coding può variare molto tra le attività proposte su Code.org, quelle proposte dagli sponsor e quelle organizzate da volontari (ad esempio nei Coder Dojo). Questa situazione è destinata a cambiare: verranno date delle indicazioni precise, anzi verranno proposti alle scuole dei veri e propri curricula didattici, con obiettivi, tempi, modi, contenuti. Chi svilupperà questi curricula per le scuole? Le scuole stesse, o almeno alcune di esse.

Un bando MIUR del 26 Settembre 2016 destina 4,3 milioni di euro alle reti di scuole per produrre 25 curricula su 10 tematiche del “digitale”. Le tematiche sono sicuramente interessanti, a partire da quelle definite "Fondamentali", ovvero:

- diritti in internet
- educazione ai media (e ai social)
- educazione all'informazione

Seguono le tematiche “Caratterizzanti”, che comprendono anche il coding:

- STEM (competenze digitali per robotica educativa, making e stampa 3D, internet delle cose)
- big e open data

- coding
- arte e cultura digitale
- educazione alla lettura e alla scrittura in ambienti digitali
- economia digitale
- imprenditorialità digitale

Più avanti nel bando vengono descritti degli esempi dei contenuti per ogni area. Per il Coding, in particolare per la scuola primaria, troviamo questa descrizione:

“Educazione e sviluppo del pensiero computazionale sia tramite attività unplugged (senza calcolatore) sia tramite linguaggi di programmazione visuali (scuola primaria) [...].

Siamo all'interno dello stesso orizzonte di Code.org. Nel seguito, però, si danno indicazioni più precise sugli argomenti:

- coding by gaming, percorsi di apprendimento condivisi in classe; uso di strumenti di coding by gaming online;
- competenze computazionali di base; il **codice binario**; identificare e scrivere istruzioni sequenziali;
- esecuzione di sequenze di istruzioni elementari; programmazione visuale a blocchi;
- capire lo sviluppo e l'utilizzo di strumenti informatici per la risoluzione di problemi;

- calcolare **espressioni logiche** con gli operatori AND e OR;
- riconoscere nel procedimento di soluzione algoritmica di un problema gli elementi strutturali fondamentali: sequenza, scelta condizionata, iterazione;
- conoscere e saper applicare nella vita quotidiana metodologie di **ricerca sequenziale, dicotomica e hash** e comprenderne i limiti di applicazione e il grado di efficienza;
- conoscere le strategie per l'**ordinamento di oggetti** (selezione/inserimento, partizionamento) e comprendere, in modo intuitive, l'efficienza della strategia adottata;
- saper eseguire semplici **algoritmi su grafi** di ridotte dimensione, quali la ricerca di **cammini e di matching**;
- saper rappresentare i dati o i risultati di un problema mediante l'uso di **tabelle, alberi o grafi**; oggetti programmabili; **verifica e correzione del codice**;
- conoscere il concetto di ipertesto, il suo ruolo nel world wide web, e la struttura a rete di calcolatori su cui esso è basato”

Come si vede, a parte le due righe in cui si citano il “coding by gaming” e la “programmazione visuale a blocchi”, qui si sta parlando dei contenuti classici di un corso di introduzione all’informatica (come quello del testo del professor Batini cita-

to). Il che lascia un po' perplessi rispetto agli obiettivi dichiarati, soprattutto perché si sta parlando di scuola primaria.

Ma soprattutto, è chiaro che il Coding appartiene all'area delle pratiche, non a quella delle riflessioni. Quando si gioca con Scratch, quando si pilota un robot, non si deve riflettere, ad esempio, su che ruolo avranno i robot nella società di domani?

Questa separazione tra riflessione e tecnica, tra aspetti etici e pratica, è indizio di un'altra grande separazione che è più difficile cogliere: quella tra aspetti cognitivi e affettivi.

In generale, il rapporto affettivo con le macchine digitali (computer, tablet, robot) e con gli artefatti digitali (programmi) è davvero poco studiato. Un po' perché l'informatica si colloca dal lato delle scienze dure, un po' perché non siamo proprio abituati a dare spazio agli aspetti affettivi nell'apprendimento.

Ne può essere esagerata l'importanza (Papert) o può essere stigmatizzato come simulacro pericoloso dei rapporti con le persone (Turkle); ma è questo che fa funzionare il Coding (e che fa amare ai programmatori il proprio lavoro).

Non è detto che docenti e famiglie condividano questa tonalità affettiva particolare. Ma non è un aspetto che si può trascurare.

Però questo non significa che occorre che tutte le attività di Coding siano giochi.

Mi sembra che ci sia un errore pedagogico alla base della maniera di proporre, e di pensare, gli ambienti di apprendimento come Scratch. E' come se ci fosse un sillogismo:

1. L'apprendimento tramite un gioco è più divertente
2. Bisogna insegnare il pensiero computazionale
3. Quindi facciamolo con un ambiente giocoso (“coding by gaming online”), così non sembra nemmeno scuola.

L'errore, a mio modo di vedere, sta nel fatto che apprendere, nel senso di procedere gradualmente in un percorso di maggior comprensione del mondo, è gratificante *di per sé*.

Creare un (piccolo) mondo, assegnargli delle regole e modificarle, poi dargli vita e vederlo evolvere, è un modo molto efficace – e piacevole - di imparare.

Questa creazione di un piccolo mondo è, appunto, la programmazione. Che è parente della creazione narrativa e di quella musicale.

Visuale o verbale, non fa molta differenza. Anzi: paradossalmente: più si aggiungono “gadget”, meno il bambino è concentrato sul processo di costruzione vera e propria del mondo arti-

ficiale. L'identificazione del bambino con l'avatar (tartarughe o pupazzetti), tipica del videogioco, è solo uno dei possibili rapporti che si possono istituire.

Il programmatore, junior o meno, è il Demiurgo del suo universo, non semplicemente uno degli eroi.



## 10. Un po' di storia...

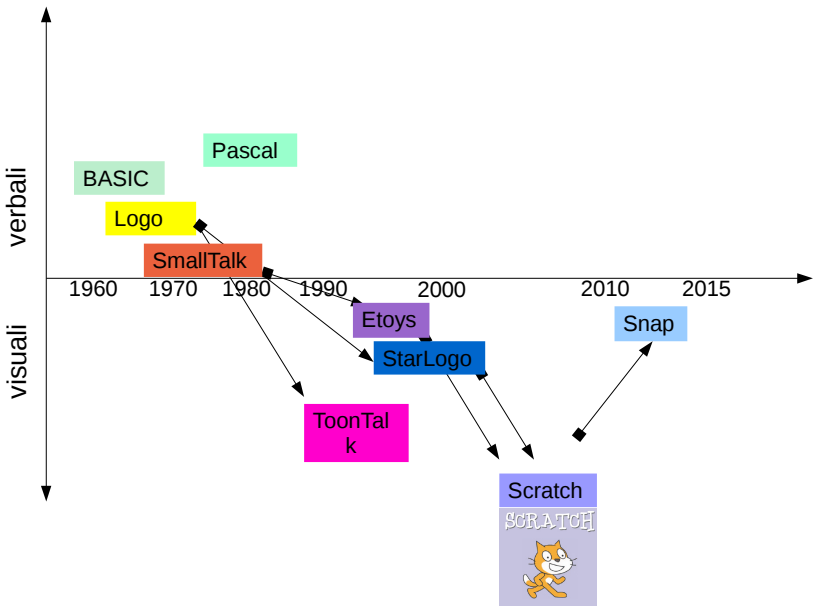
Siccome chi non conosce la storia è destinato a ripeterne gli errori, torniamo un po' indietro.

Scratch non è nato dal nulla: deriva da Squeak!, che deriva a sua volta da SmallTalk ma innestandoci le idee pedagogiche del Logo.

Ma per essere precisi, tutto è iniziato con il bistrattato BASIC, negli anni sessanta del millennio scorso.

Per dare un'idea: non erano ancora nati né il C (1973), né il PERL (1987) per non parlare di Java (1995).

In quella caverna preistorica c'erano degli uomini forse primitivi, ma con delle idee luminose.



## B.A.S.I.C.

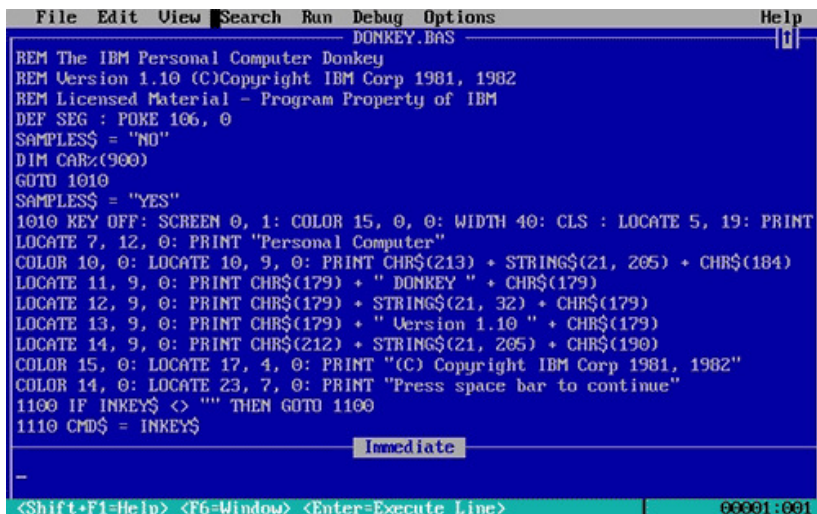
Quando Kemeny e Kurtz alla Dartmouth University nel maggio del 1964 inventano il BASIC (Beginners' All-purpose Symbolic Instruction Code) l'idea era quella di proporre uno strumento con cui chiunque potesse programmare. Un linguaggio talmente facile da poter essere imparato senza conoscenze pregresse in matematica e logica.

Perché? Perché a quell'epoca programmare era un'attività riservata a pochi super esperti, che avessero accesso a macchine costose. E perché c'era l'idea che sapere programmare fosse una competenza che avrebbe migliorato la vita di chiunque.

Nasce perciò un linguaggio con una sintassi semplice, con delle istruzioni che assomigliano all'inglese, pensato in funzione dell'utente.

Un'idea che avrebbe dovuto aspettare ancora dieci anni prima di essere davvero realizzata, con la diffusione degli home computer e con un interprete BASIC capace di fare grafica e suonare.

Incidentalmente, l'interprete che ha contribuito di più alla diffusione del linguaggio è stato scritto da Paul Allen, Monte Davidoff e Bill Gates.



```
File Edit View Search Run Debug Options Help
DONKEY.BAS
REM The IBM Personal Computer Donkey
REM Version 1.10 (C)Copyright IBM Corp 1981, 1982
REM Licensed Material - Program Property of IBM
DEF SEG : POKE 106, 0
SAMPLE$ = "NO"
DIM CAR$(900)
GOTO 1010
SAMPLE$ = "YES"
1010 KEY OFF: SCREEN 0, 1: COLOR 15, 0, 0: WIDTH 40: CLS : LOCATE 5, 19: PRINT
LOCATE 7, 12, 0: PRINT "Personal Computer"
COLOR 10, 0: LOCATE 10, 9, 0: PRINT CHR$(213) + STRING$(21, 205) + CHR$(184)
LOCATE 11, 9, 0: PRINT CHR$(179) + " DONKEY " + CHR$(179)
LOCATE 12, 9, 0: PRINT CHR$(179) + STRING$(21, 32) + CHR$(179)
LOCATE 13, 9, 0: PRINT CHR$(179) + " Version 1.10 " + CHR$(179)
LOCATE 14, 9, 0: PRINT CHR$(212) + STRING$(21, 205) + CHR$(190)
COLOR 15, 0: LOCATE 17, 4, 0: PRINT "(C) Copyright IBM Corp 1981, 1982"
COLOR 14, 0: LOCATE 23, 7, 0: PRINT "Press space bar to continue"
1100 IF INKEY$ <> "" THEN GOTO 1100
1110 CMD$ = INKEY$
Immediate
-
<Shift+F1=Help> <F6=Window> <Enter=Execute Line> 00001:001
```

## Logo

Il Logo, altro elemento chiave del discorso, nasce pochi anni dopo l'invenzione del BASIC, cioè nel 1967.

L'obiettivo è completamente diverso: in origine era stato pensato da un gruppo ricercatori nel campo della nascente Intelligenza Artificiale (del calibro di Bobrow, Feurzeig e Salomon) per insegnare la programmazione in LISP agli studenti.

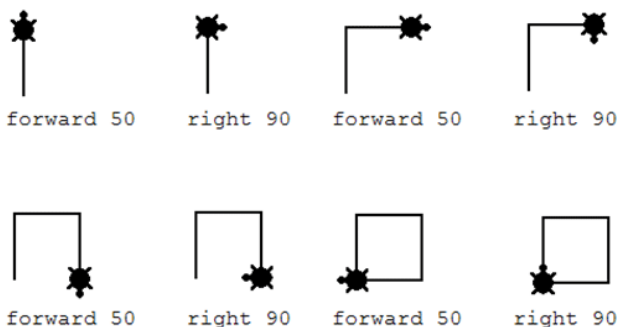
Con l'intervento di un matematico (Seymour Papert) Logo diventa una maniera alternativa di imparare la geometria che non passasse per la lettura di teoremi ma per la costruzione automatica di figure in soggettiva, dal punto di vista dell'utente, che si identifica in un avatar digitale: la famosa tartaruga.

Dunque con un obiettivo didattico, non legato alle tecnologie. Usare Logo non serve a diventare programmatori.

La base di partenza scelta è un po' più nobile del BASIC, in termini di "generazioni dei linguaggi": viene scelto il LISP che è un linguaggio funzionale, non imperativo.

Il modello di interazione è quello dell'insegnamento ad un robot. Che all'inizio è un robot vero, poi diventa solo un avatar sullo schermo.

Quando – molto più tardi – vengono realizzati degli interpreti per Apple e per PC IBM, cominciano a essere disponibili delle traduzioni nelle lingue nazionali, cioè in cui le istruzioni fossero parole comuni della lingua madre di chi lo utilizzava (SE... ALLORA, RIPETI...FINCHE), con l'idea di facilitare la scrittura di programmi da parte di bambini che non conoscono l'inglese.



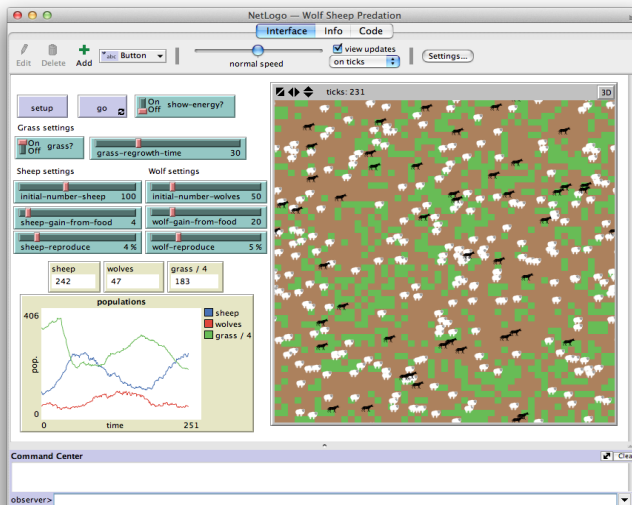
© 2000 Logo Foundation

Dal Logo originale sono gemmate altre versioni che hanno lasciato da parte la geometria piana e si sono concentrate sulla concorrenza (cioè sulla possibilità di "lanciare" molte tartarughe contemporaneamente per simulare il comportamento di un alveare o del traffico all'ora di punta) o sulla multimedialità (la possibilità di "vestire" la tartaruga con immagini, animazioni e suoni), fino a versioni in cui non è più necessario scrivere i programmi, ma è sufficiente selezionare le azioni da un menù e comporle, esattamente come in Scratch.

Nelle pagine seguenti ne mostriamo due tra i più noti: NetLogo e StarLogo.

## NetLogo<sup>33</sup>

- Ideato da Uri Wilenski presso il Center For Connected Learning and Computer Based Modelling, Northwestern University, a partire dal 2005
- Scritto in SCALA, la versione web in Javascript
- Permette di connettere più istanze per effettuare simulazioni complesse (Netlogo Hubnet)
- Opensource (GPL)

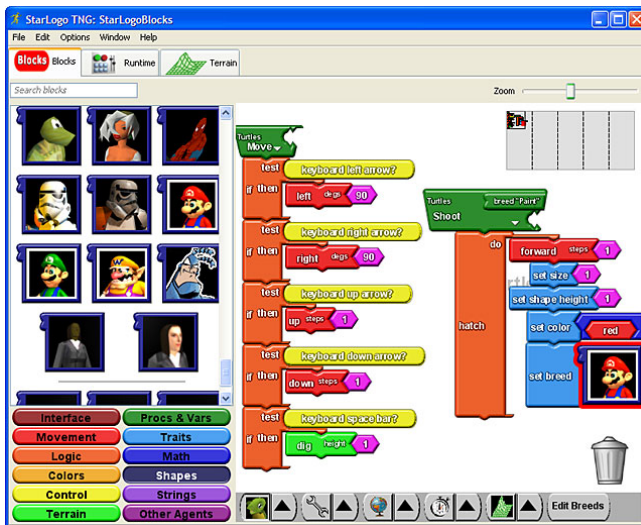


<sup>33</sup> <http://ccl.northwestern.edu/netlogo/>



## StarLogo<sup>34</sup>

- Ideato da Mitchel Resnick ed Eric Klopfer presso il MIT Media Lab a partire dal 2000
- Scritto in Java (la versione più recente però usa Flash)
- Non è del tutto opensource, perché non può essere utilizzato per fini commerciali !



---

34 <http://education.mit.edu/projects/starlogo-tng>

## SmallTalk

Con un finanziamento considerevole dell'ARPA, mirato a ripensare il mondo dell'interazione uomo-macchina, Alan Kay e altri colleghi del Learning Research Group allo Xerox Parc nel 1971 creano il linguaggio SmallTalk.

Per dire, quel gruppo di lavoro e quelle ricerche sono all'origine di innovazioni come finestre, mouse, ipertesti, tablet, programmazione ad oggetti e altre quisquiglie.

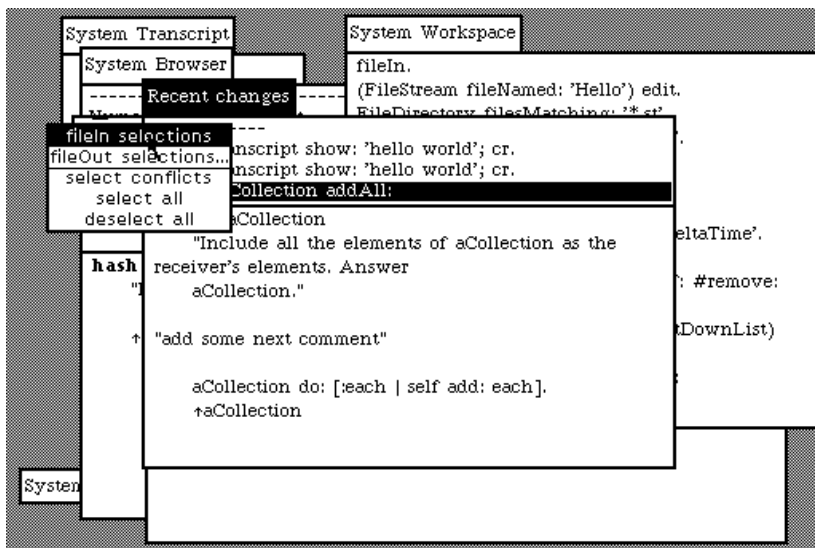
Nato inizialmente come linguaggio educativo, nel tempo SmallTalk diventa anche uno strumento professionale.

E' semplice, ha una sintassi elegante e coerente. SmallTalk introduce un nuovo modo di programmare, in forma di dialogo tra oggetti. In SmallTalk tutto è oggetto, compreso l'interprete stesso; e tutti i comandi sono messaggi.

$7 + 7$

significa inviare il messaggio '+' all'oggetto 7.

SmallTalk mette in scena degli attori, gli assegna le battute e assiste allo spettacolo.



Scratch è figlio di queste tre idee potenti nate negli anni sessanta:

- un linguaggio semplice adatto a tutti
- un modello di interazione pensato per l'apprendimento
- un ambiente grafico e una logica ad oggetti

Ma che ne è stato di quelle idee potenti?



## 11. E oggi?

Se oggi ci guardiamo intorno, vediamo una situazione completamente diversa da quella che avevano immaginato i mitici precursori.

I programmatori sono aumentati moltissimo di numero, ma non è l'uomo qualunque che programma, è solo chi ha intrapreso un percorso formativo specializzato.

I programmi sono ovunque, ma quasi nessuno sa/può modificarli (o almeno sceglierli con consapevolezza di quello che fanno).

Tutti sanno cos'è una app, ma nessuno ha un'idea anche vaga di come funziona, quali dati gestisce, a chi li invia. Con gli evidenti rischi per la privacy, e con l'arricchimento velocissimo di chi costruisce e vende profili e pubblicità, eccetera eccetera.

Tutti hanno in bocca il termine “opensource”, usandolo magari a sproposito e confondendolo con “gratis”, ma dimenticano che quasi nessuna delle app che hanno felicemente installato sul proprio smartphone lo è.

Il sogno di Kemeny dell'uomo qualunque in grado di programmare non si è realizzato, ma anche quello di Papert di

cambiare radicalmente la didattica tramite la tecnologia digitale non sembra passarsela molto meglio.

SmallTalk non è più utilizzato da molti, anche se ha dato origine a quasi tutti i linguaggi moderni; le interfacce grafiche sono ovunque, ma non permettono di guardare dentro gli oggetti.

Cos'è che non ha funzionato?

Perché il BASIC è stato snobbato (pure se esistono implementazioni moderne, per Windows e per Linux) e il Logo dimenticato nelle scuole? Eppure abbiamo visto che Scratch deve tanto a questi due antenati, anzi tutto sommato ci aggiunge poco.

E se non c'è grande differenza tra le caratteristiche dei primi linguaggi “educativi” e quelli di oggi, perché stavolta dovrebbe andare meglio?

Come facciamo ad assicurarci che non succeda di nuovo?

Non sarà il caso, stavolta, di fare attenzione alla situazione globale e di assicurarci che le condizioni di successo per l'iniziativa (non solo quelle tecniche) ci siano tutte?

## 12. Come potrebbe funzionare davvero

Ma allora, perché stavolta funzioni, quanto tempo ci va dedicato? 8 ore l'anno non sono sufficienti?

No. Ma non perché bisogna farne 10.

Perché insieme al coding vanno affrontate altre questioni.

1. Ci deve essere un piano didattico più esteso che comprenda le competenze di cittadinanza digitale, i nuovi diritti e i doveri “digitali” – la privacy, la condivisione, l'attenzione all'uso consapevole delle risorse digitali, la libertà di leggere e riusare codice. Non sono temi che si possono rimandare (anche perché non capiterà mai di affrontarli, qualsiasi percorso accademico si scelga).

2. In questo piano devono essere prese in carico (o smontate e rimontate) le altre “materie”, senza le quali il Coding perde di senso a scuola. Perché non può essere un'attività ricreativa incastrata tra l'ora di matematica e quella di storia, ma deve fare i conti con entrambe.

3. E non si possono prendere in prestito proposte didattiche validissime, sì, ma in contesti educativi molto differenti. Vanno adattate tenendo conto della nostra cultura e del nostro ambiente – in negativo e in positivo.

4. Deve essere rivalutata la componente linguistica del Coding, la sua parentela con le altre forme di scrittura. In fondo non si insegna a scrivere in Italiano trascinando blocchetti...

5. Il Coding deve essere affrontato nel quadro di un ripensamento della didattica che prepari non solo alla soluzione di problemi dati, ma anche alla posizione di nuovi problemi, all'invenzione di narrazioni in tutti campi.

6. Vanno esaminati e gestiti gli aspetti affettivi dell'apprendimento tramite costruzione di artefatti digitali

7. Devono essere scelti un ambiente e un linguaggio adatti, in funzione dell'età degli studenti, di obiettivi didattici concreti, dei dispositivi e degli ambienti disponibili. Ma con uno sguardo d'insieme, con una prospettiva che copra tutto l'arco dell'educazione, non solo qualche anno.

9. Soprattutto, in attesa di occasioni formative non randomiche e superficiali, bisogna darsi da fare: domandarsi, cercare, studiare, provare.

Insomma: coding sì, ma sul serio.



## Suggerimenti di lettura

1. G. Lariccia: *Le radici dell'informatica. I fondamenti di una "informatica povera e cognitiva" riscoperti nel funzionamento della mente umana e nelle sue proiezioni sulla organizzazione sociale*. Firenze, Sansoni, 1988
2. C. Batini, *Le basi dell'informatica*, Editori Riuniti, 1984  
<http://www.programmailfuturo.it/media/docs/approfondimenti/Batini-basi-dell-Informatica.pdf>
3. S. Papert: *Mindstorms. Children, computers and powerful ideas*, Basic Books, 1980 (*Mindstorms. Bambini computer e creatività*. Emme Edizioni, 1984)
4. S. Papert: *The Children's Machine. Rethinking school in the age of the Computer*. Basic Books, 1993 (*I bambini e il computer*, Rizzoli, 1994)  
<http://www.papert.org/articles/ChildrensMachine.html>
5. D. Rushkoff: *Program or be programmed*, ORbooks, 2010  
<http://www.orbooks.com/catalog/program/#>

6. H. Abelson, K. Ledeen, H. Lewis: *Blown to bits. Your Life, Liberty, and Happiness After the Digital Explosion*, 2008

<http://www.bitsbook.com/excerpts/>

7. AAVV: *ScratchJr: Computer programming in early childhood education as a pathway to academic readiness and success*

<http://web.media.mit.edu/~mres/proposals/ScratchJr-draft.pdf>

8. AAVV: *New Pathways into Data Science. Extending the Scratch Programming Language to Enable Youth to Analyze and Visualize Their Own Learning*

<http://web.media.mit.edu/~mres/proposals/NSF-data-proposal.pdf>

9. K.Kahn: *ToonTalk and Logo.Is ToonTalk a colleague, competitor, successor, sibling, or child of Logo?*

<http://el.media.mit.edu/logo-foundation/resources/papers/pdf/toontalk.pdf>