2 La pratica didattica

2.1 Competenze "computazionali"?

Ammesso dunque che si possa parlare di lingua e letteratura in una classe attrezzata con LIM e computer, cosa c'entra tutto questo con il coding? E come va applicato l'uno all'altra?

La situazione attuale in Italia vede (ancora) una contrapposizione: da un lato chi inneggia al "coding" senza se e senza ma, e vede nel progetto ministeriale "Programma il futuro" l'inizio di una nuova era a prescindere dai modi della sua applicazione; dall'altro i detrattori del "coding", divisi tra pionieri pentiti e analogici integrali. Tra i primi, di solito più giovani, si osserverà che non ha molta importanza stare a scegliere un modello pedagogico e dei riferimenti teorici prima di avviare un'attività di coding che è comunque gratificante per tutti; tra i secondi, qualcuno dirà che queste discussioni e questi ragionamenti si sarebbe potuti fare tranquillamente *senza un computer*, magari usando carta e matita; oppure utilizzando un programmino già fatto da qualche docente volenteroso e reperibile in rete senza eccessiva fatica.

E' proprio dalla resistenza di questi ultimi che bisogna partire se si vuole che i vantaggi del coding non restino circoscritti nei giardini felici degli entusiasti e si facciano prassi. La prima obiezione credo che non sia valida: senza uno strumento pratico per costruire un modello funzionante di una situazione (quello che nel seguito chiameremo "un automa") non è possibile sperimentare davvero quel modello. Se si usa solo la lingua rappresentata staticamente alla lavagna multimediale - ad esempio con una tabella -, ai ragazzi si deve chiedere di fidarsi del fatto che le regole insegnate funzionino davvero e coprano tutti i casi possibili. La lingua viene presentata come un proprietà di qualcun altro – mentre è patrimonio di tutti.

Ugualmente non mi convince la seconda obiezione (meglio usare software didattico già pronto): una cosa è costruire un modello, un'altra è utilizzarlo senza sapere come è fatto "dentro". Non credo sia il caso di dilungarsi qui sul perché una modalità didattica basata sulla costruzione sia più efficace di una basata sull'uso. Si può, e si deve, discutere invece sul punto di partenza e di arrivo del processo di costruzione: non si può partire da zero e non si può pretendere di costruire un programma perfetto.

Detto questo, gli aspetti strettamente informatici in queste proposte sono volutamente messi in secondo piano. Il motivo è che non sono il centro dell'attività: non si tratta di un'attività il cui scopo è quello di imparare, ad esempio, l'uso delle liste e della ricorsività, ma un'attività *in cui* l'uso delle liste permette di costruire un modello funzionante della struttura di una poesia. Questa proposta non è mirata all'apprendimento del cosiddetto "pensiero computazionale" (qualsiasi cosa voglia dire), ma all'apprendimento del funzionamento della *lingua* e della struttura di un *testo* tramite l'applicazione di competenze computazionali.

Ma quindi queste competenze devono essere già possedute dagli studenti? Non è detto; anzi, l'attività proposta è anche un modo per mostrare come questi strumenti siano potenti e funzionali al progetto scelto e quindi per imparare l'uso in maniera significativa e motivante.

E se i bambini (o ragazzi) giù conoscono Scratch? Forse è meglio, forse no. Intanto cominciano col dire che l'ambiente principale scelto per esemplificare il codice non è un ambiente visuale come Scratch, Snap! e tanti altri, ma un ambiente di editing testuale; ma anche il linguaggio scelto è di un tipo leggermente diverso. Qualche parola per spiegare perché.

Scratch[1] è un ambiente conosciutissimo e usatissimo per le attività di coding nelle scuole di tutto il mondo. Ha una storia interessante alle spalle ed è opera di un gruppo di ricercatori del MIT. Snap![2], sviluppato a Berkeley, è forse meno noto, ma ancora più interessante e più completo.

L'idea di base di Scratch è che per realizzare delle storie multimediali, dei piccoli giochi, non sia necessario scrivere codice ma sia sufficiente scegliere un blocco-istruzione in un pannello, trascinarlo nell'area del codice e incastrarlo con gli altri.

Non è vero che non sia necessario leggere: infatti i blocchi riportano scritta la loro funzione. La principale differenza, a mio parere, è che la forma dei blocchi permette solo certi incastri e non altri. Ad esempio, all'interno di un blocco se-allora-altrimenti c'è uno spazio per inserire la condizione logica. Questo spazio vuoto ha una forma esagonale, e tutti i blocchi che possono avere solo valore vero o falso hanno appunto una forma esagonale. In questo modo è letteralmente impossibile inserire nello spazio delle condizioni logiche un valore numerico o un'istruzione. In un ambiente più tradizionale, oltre ai suggerimenti, all'autocompletamento, bisogna essere in grado di interpretare il messaggio di errore che viene riportato quando si cerca di scrivere qualcosa di sintatticamente inappropriato.

E' un po' come se si avesse un programma di videoscrittura che impedisce di scrivere "ieri sono andato ha scuola". Non un programma che segnala l'errore (come ormai fanno tutti), ma proprio impedisce di commetterlo.

E' una funzione utile? Ad ogni età? Ad ogni livello di competenza? Aiuta o impedisce il formarsi di una consapevolezza della sintassi? Gli studi in proposito non sono arrivati ad una risposta univoca. Da un lato il bando delle calcolatrici dalle aule, dall'altro la pedagogia dell'errore di rodariana memoria.

In ogni caso, la scelta di non utilizzare qui un ambiente visuale è stata fatta, a dispetto della contraddizione apparente, per *facilitare* la lettura del codice sorgente, il suo commento, la sua modifica, il riuso di parti, insomma la consapevolezza linguistica. Gli ambienti di scrittura del codice sono dotati di funzionalità che comprendono quelle di un wordprocessor (taglia, sposta, copia, cerca, sostituisci) ma in alcuni casi le superano. Una per tutte: la possibilità di visualizzare la struttura sintattica del codice tramite degli artifici tipografici, come il colore, il carattere, l'allineamento.

1 https://scratch.mit.edu/
2 https://snap.berkeley.edu/

In particolare, abbiamo scelto Kojo,[3] che è un ambiente didattico basato sul linguaggio funzionale Scala[4] e ricco di strumenti interessanti, tra cui un sottoinsieme di istruzioni per muovere una tartaruga sullo schermo ripreso dal Logo, un ambiente per la geometria costruttiva e un linguaggio per produrre musica.

E' possibile, se necessario, tradurre gli esempi di codice in qualsiasi altro linguaggio, compreso lo stesso Scratch. Tuttavia qui si è scelto un linguaggio *funzionale* (e non imperativo, come sono i linguaggi che vengono usati normalmente per introdurre i bambini alla programmazione) perché a parere dell'autore questo modello si presta meglio a trattare l'argomento linguistico, perché è in generale più potente e, perché no?, più elegante. Si tratta evidentemente anche di una sfida: i costrutti imperativi ("fai questo") sono facili da interpretare come comandi impartiti ad un robot, e questa interpretazione è comprensibile anche per bambini più piccoli. Gli altri modelli di programmazione (funzionale, orientata agli oggetti, logica) richiedono probabilmente un lavoro iniziale maggiore, che però vale la pena di essere almeno valutato prima di decidere di scegliere un linguaggio o un altro.

Il modello *imperativo* dei linguaggi è costruito a partire dall'idea che ci siano solo due attori: il programmatore e un robot che ne esegue gli ordini. Ma non è il solo possibile: si potrebbe partire da un dialogo tra "personaggi" (come nella programmazione *orientata agli oggetti*), o dalla costruzione di una dimostrazione (come nella programmazione *logica*), o dal flusso di trasformazione di dati di partenza (come in quella *funzionale*). Tra l'altro, è probabile che il presente – e il futuro – dell'informatica stiano andando piuttosto in queste direzioni. Se fosse vero che il coding a scuola prepara alle professioni del futuro (cosa della quale non sono affatto certo), allora converrebbe puntare ai modelli, se non addirittura ai linguaggi, che nel futuro hanno maggiori probabilità di essere ancora utilizzati.

In coda alle prime due attività, comunque, presentiamo la traduzione di una versione del codice sorgente in altri due linguaggi diversi, Prolog e LibreLogo. Le tre versioni presentano molte somiglianze e qualche differenza, alcune più evidenti (di tipo grammaticale e lessicale), alcune meno (di tipo strutturale). Anche la comparazione dei codice – che in fondo è un compito di letteratura comparata - è un'attività interessante che può originare discussioni sul concetto di traduzione. In fondo al testo trovate, per ognuno dei tre linguaggi, un breve tutorial che ne spiega le caratteristiche essenziali, o per lo meno quello che servono per capire i codici sorgenti presentati come esempi. Chi è interessato davvero è invitato ad andare oltre, seguendo le piste di approfondimento suggerite.

2.2 Caso come educatore

Tra le motivazioni citate da persone di cultura umanistica per giustificare la loro avversione per i computer e i programmi mi è capitato spesso di trovare l'affermazione che i computer sono "stupidi", che ripetono sempre e soltanto quello che gli è stato detto. Sembra di sentire Platone

3 http://kogics.net/kojo
4 http://www.scala-lang.org/

parlare del testo scritto, figlio illegittimo dell'autore che non è in grado di difenderlo se non ripetendo all'infinito le stesse cose.[5]

Ci sono almeno due ragioni per cui questo non è vero: la prima è perché i programmi *possono imparare*, nel senso di aggiungere conoscenze a quelle inizialmente contenute nel programma stesso; la seconda è perché i computer possono *eseguire istruzioni casuali* e quindi produrre cose nuove.

La prima ragione è quella che fa discutere di più, perché apprendimento confina con intelligenza. E' vero apprendimento? C'è consapevolezza dell'apprendimento? Per il momento la trascuriamo (una breve ripresa del tema la trovate nell'attività sulla morra cinese e nel tutorial dedicato al Prolog).

La seconda sembra banale: cosa c'è di innovativo nel caso? E cosa c'è di educativo?

Qui l'equilibrio generale del progetto editoriale non ci permette di approfondire il tema come meriterebbe. Facciamo però qualche accenno a delle piste di riflessione che potrebbero anche essere esplorate con gli studenti.

Cosa significa eseguire istruzioni casuali? Certo non comporre casualmente un programma a partire da istruzioni pescate a caso, perché non funzionerebbe. Lo spazio per il caso è molto ristretto: si può scegliere una parola a caso da inserire in una struttura, o una struttura di frase a caso, ma l'uso che se ne fa sarà determinato. Da un punto di vista più generale, la vera domanda è: ma come è possibile che un computer faccia un'azione a caso? Se il computer è una macchina completamente deterministica (come tutte le macchine), come fa a compiere un'azione non determinabile a priori? Casuale, qui, significa che nessuno è in grado i prevedere il risultato di un'operazione. Il termine che si usa in informatica è *pseudocasuale*, proprio a evidenziare questo aspetto soggettivo.

E invece gli umani? Se chiedete ad uno studente di dirvi un numero a caso tra uno e dieci, sarà davvero casuale, o sarà invece un numero legato in qualche modo alla sua esperienza precedente (una data importante, il numero della maglia del giocatore preferito)?

Insomma, districarsi tra azioni casuali, determinate, prevedibili in base a delle premesse, non è facile. Ma nemmeno necessario per il nostro discorso. Quello che ci interessa è che i computer possono generare esempi di strutture che (ci appaiono come) imprevisti.

Per esperienza diretta, uno dei momenti più emozionanti nel "coding" è sicuramente quando, dopo aver scritto insieme ai ragazzi un programma che in base a valori casuali varia angoli, dimensioni e colori di uno schema, si assiste all'esecuzione del disegno.

Quello che viene fuori è quello che ci aspettavamo? Ci assomiglia almeno un po'? O è completamente diverso?

E poi: ci sono delle regolarità che emergono? Se si, da dove vengono?

Facciamo un esempio con LibreLogo.

```
TO quadrato
       PENCOLOR ANY
      FILLCOLOR ANY
```

5 Fedro, 275d

```
        FILLTRANSPARENCY (RANDOM 50)
        PENDOWN
        lato = RANDOM 50
        angolo = 90
        REPEAT 4 [
                FORWARD lato RIGHT angolo
        ]
        FILL
        PENUP
END

TO triangolo
        PENCOLOR ANY
        FILLCOLOR ANY
        FILLTRANSPARENCY  (RANDOM 50)
        PENDOWN
        lato = RANDOM 100
        angolo = 120
        REPEAT 3 [
                FORWARD lato RIGHT angolo
       ]
        FILL
        PENUP
END

TO disegnoCasuale
          CLEARSCREEN
          HOME
          REPEAT 10 [
                      spostaACaso
                      quadrato
                      spostaACaso
                      triangolo
              ]
END

TO spostaACaso
        PENUP
        FORWARD (RANDOM 100)
        RIGHT (RANDOM 90)
END

disegnoCasuale
```

Stefano Penge: Lingua, Coding e creatività – Copyright Anicia 2017

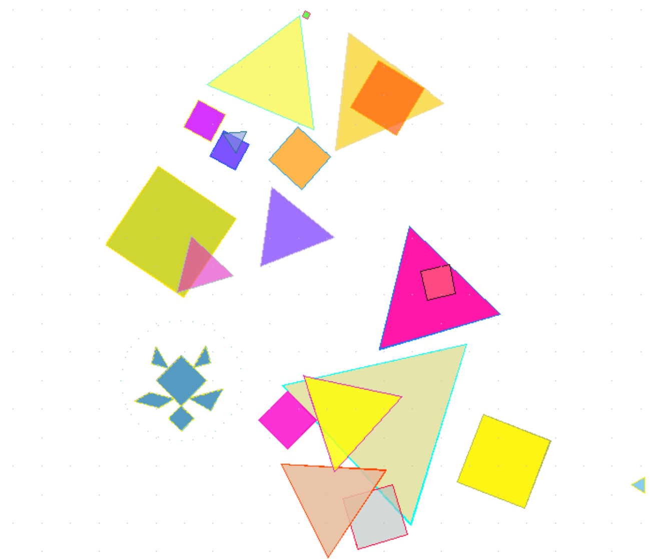

Illustrazione 1:

Questo codice disegna dieci quadrati e dieci triangoli di dimensioni e colori casuali in punti casuali dello schermo. Al di là del valore estetico del risultato, c'è tanto da imparare da un programma come questo. Guardate il codice sopra riportato: è fatto di parole e numeri. Sperimentando con i valori numerici ci si può avvicinare ad una struttura regolare di sapore Kandiskiano, o alla mappa di una città. Si possono modificare *tutti* i numeri? Se lo si fa con alcuni, quelli che seguono la parola REPEAT, scompaiono le figure regolari (o appaiono molto più raramente). Se invece dove appare la forma (RANDOM …) si sostituisce un numero, si fissano alcuni valori che sono casuali e si riduce la variabilità, ottenendo sempre lo stesso disegno. C'è insomma uno spettro continuo che va da un comportamento perfettamente controllato e prevedibile a uno che, pur restando controllato, è imprevedibile. Imparare a muoversi in questo spazio è una delle competenze, a mio avviso, davvero trasversali tra le discipline. Il mondo, per come lo conosciamo, non è né del tutto frutto di processi casuali, né completamente prevedibile con delle equazioni. La vita è possibile solo in questo campo intermedio.

Quest'utilizzo del caso come sorgente di sorpresa e stimolo alla riflessione sulla regolarità non è limitato al disegno; anzi, nel resto di questo testo non troverete più (se non eccezionalmente) quadrati e triangoli, ma frasi, filastrocche e problemi generati casualmente. E' una differenza importante che segna la distanza tra le nostre attività e i problemi di coding che potete trovare sulla rete, e che hanno una sola soluzione: qui ogni esecuzione del programma è, e deve essere, diversa dalle altre; solo la competenza del gruppo classe potrà valutarne l'adeguatezza al progetto originale.

2.3 Dal modello alla pratica

Cominciano a essere disponibili dei testi sul coding, rivolti per lo più alla scuola primaria, che propongono degli esercizi basati tipicamente su Scratch e sulle sequenze proposte in Code.org. Avrete ormai capito che il testo che state leggendo è qualcosa di diverso.

Anche se ogni attività è corredata di una presentazione, di una descrizione della proposta, di suggerimenti, di piste di discussione (e naturalmente, del codice sorgente relativo), non ci si aspetti di poter *sempre* svolgere le attività *esattamente come sono descritt*e. Questo non è un manuale da usare immediatamente in classe senza un minimo di adattamento, in cui ci siano materiali giù pronti per l'uso e organizzati nel consueto modo (prerequisiti, obiettivi didattici, schede di valutazione).

Certo sarebbe stato possibile organizzarlo in questo modo, riducendo il campo di applicazione ad un certo grado scolastico, ad un'età di riferimento, ad un solo linguaggio, ad un solo tema. In questo modo, però, avremmo ristretto i destinatari e probabilmente si sarebbe perso il carattere di stimolo alla riflessione e alla personalizzazione, che è invece il fulcro del progetto editoriale.

Le attività descritte vanno intese come *modelli* da adattare e contestualizzare, non come schemini pronti da recitare in classe. Al docente si richiede di capire il senso di ogni attività, di *digerirla* e solo dopo di provare a sperimentarne una o più in classe, nell'ordine che le/gli sembra più opportuno. Le versioni di codice sorgente presentate sono di difficoltà crescente, in modo che si possa scegliere quella più consona al livello della classe, oppure lavorare a partire dalla prima e andando avanti, fino a cambiare qualche elemento per agganciarlo ad attività già svolte o in programma. E' anche perfettamente pensabile tradurre i codice in un linguaggio diverso (Scratch, Snap! o altro ancora) per fare un'attività corrispondente negli obiettivi ma diversa nelle modalità e nei risultati.

A mio modesto avviso, questo è l'unico modo possibile per evitare che il "coding" si trasformi da ricca opportunità in ripetizione meccanica di azioni di cui non si capisce bene la valenza didattica.

Soprattutto nelle prime attività, ogni volta che viene introdotto un nuovo termine del linguaggio Scala questo viene segnalato e spiegato. Poi negli ultimi capitoli trovate tre brevi tutorial relativi ai linguaggi usati, corredati di riferimenti per approfondirne lo studio. La domanda che probabilmente vi sarete posti è: ma da dove bisogna partire? Prima dallo studio del linguaggio per poi passare al suo uso nelle attività, oppure prima dalle attività per poi andare ad approfondire con i tutorial?

Non so dare una risposta univoca. Dipende dalle competenze del lettore. Se è in grado di capire, grosso modo, quello che viene proposto e i frammenti di codice presentati, il secondo modo è più utile, perché studiare una funzione senza avere idea di a cosa possa servire è frustrante a tutte le età, ma soprattutto per un adulto. Se invece appartiene alla tribù di quelli che leggono le istruzioni e preferisce avere prima almeno un'infarinatura del linguaggio, allora può andare subito a leggere il tutorial (anche per sapere come si fa a installare l'ambiente di lavoro sul proprio computer). Ma dovendo scegliere, quale dei tre?

Scala è un linguaggio serio, da grandi, che però si può utilizzare in un ambiente facilitato (Kojo) che consente anche di muovere una tartaruga qua e là. Potrebbe essere una buona scelta per chi ha una certa familiarità con Snap! o Scratch e vuole procedere oltre. Per chi invece comincia veramente da zero, la scelta migliore (a mio avviso) è Prolog, che non assomiglia a nient'altro, ma per un umanista può essere più semplice e produttivo.

Per chi lavora nella scuola primaria, la scelta migliore è probabilmente Logo. La versione del linguaggio utilizzata è quella che è "nascosta" dentro LibreOffice, che non è perfettamente compatibile con la versione standard UCB Logo. Tuttavia se necessario non dovrebbe essere difficile, una volta capite le piccole differenze, adattare il codice alla propria versione di Logo.

La mia personale opinione è che sono interessanti tutti e tre, e che potendo vale la pena di di dare un'occhiata a tutti. Per il linguaggi di programmazione vale lo stesso detto che circola sull'apprendimento delle lingue naturali: dopo le prime tre, è tutta discesa.

2.4 Linguaggi e stili

Scala e Logo sono linguaggi *funzionali*. Significa che il modello di programmazione che incarnano (la famiglia cui appartengono) è quello della scrittura di funzioni che restituiscono valori. E' un modello che risale agli anni Cinquanta ma ultimamente che sta tornando in auge con Haskell, Ocaml, F#. Sono linguaggi eleganti, molto robusti (cioè resistenti all'errore umano) perché minimizzano gli effetti collaterali dovuti alla creazione di miriadi di variabili che possono essere modificate involontariamente da altre parti del programma. Un programma scritto in stile funzionale non ha bisogno di variabili per "appoggiare" i risultati intermedi, ma è una catena di applicazioni di funzioni.

Prolog invece appartiene al paradigma della programmazione *logica*. Qui scrivere un programma significa definire regole e fatti, e a partire da questi cercare di dimostrare teoremi. Chi scrive in Prolog non si preoccupa del *come*, ma del *cosa*. Sembra la negazione del pensiero computazionale.

Sono differenze che sembreranno poco significative per un neofita, ma che sono fonte di diatribe infinite tra programmatori. Esistono circa duemila linguaggi di programmazione diversi, da quelli seri a quelli giocosi, fino a quelli impossibili da usare,[6] e ne nascono di nuovi ogni giorno. Per ragioni che possono essere casuali, o culturali, o personali, ogni programmatore sceglie, ad un certo punto della sua vita professionale, il suo paradigma preferito e il suo linguaggio di elezione. Raramente un programmatore conosce un solo linguaggio, ma spesso pontifica sulla evidente superiorità di uno sull'altro. Altrettanto spesso si discute sugli stili, sugli errori da *newbies*, sulla eleganza di un costrutto e sulla becera efficienza di un altro pure equivalente.

Tutto questo per sfatare il mito che la programmazione sia un arte minore, un mestiere governato solo dalla logica e dall'efficacia. Chi scrive programmi ha gli stessi problemi e le stesse fisime di chi

6 Il termine esatto è "linguaggi esoterici": linguaggi formalmente corretti, ma assurdi, come Brainfuck https://it.wikipedia.org/wiki/Brainfuck.

scrive romanzi: appartiene ad una scuola, venera modelli, rifinisce mille volte un passaggio solo perché non è esteticamente soddisfacente.

Questo, in fondo, è il motivo principale per cui in questo testo non è stato scelto un linguaggio (o meglio: un ambiente di programmazione) visuale. Perché al di là della supposta maggiore facilità dello spostare blocchi anziché scrivere parole, c'è una parentela forte tra la scrittura di un codice sorgente e la scrittura di un testo di altro genere; rinunciare in partenza a esplorare questa parentela, soprattutto per chi si occupa di lingue e letterature, mi sembra un'occasione persa.

Chi progetta un programma ha in testa qualcosa che assomiglia ad un plot di un romanzo: immagina personaggi, scenari, colpi di scena, epiloghi felici o infelici. Mi è capitato di ragionare con mio figlio Luca sul concetto di *suspense* all'interno di un codice sorgente: una conclusione non dichiarata, di cui però si sviluppa l'attesa man mano che il codice va avanti. A dispetto della compartimentazione attuale dei saperi e degli insegnamenti, programma meglio chi ha un'esperienza personale di storie e di narrazioni. C'è addirittura chi ritiene che malgrado il cosiddetto *gender gap* (cioè la cronica mancanza di donne tra i programmatori), le donne siano meglio attrezzate per scrivere programmi, perché più dotate in termini di creatività e più capaci di tradurre in termini narrativi uno schema statico. Tutta da verificare, ma sicuramente un'ipotesi affascinante.

Se è vero – e non è ancora dimostrato – che tutti dovrebbero saper programmare, è almeno altrettanto vero che chi programma dovrebbe saper immaginare e scrivere una storia. Altrimenti non farebbe che mettere meccanicamente in sequenza istruzioni; e questo, ahimé, non ha mai portato da nessuna parte.

Infine: tutto il codice sorgente, per quel che vale, è rilasciato con licenza GPL 3.0. Significa che potete riusarlo, modificarlo, pubblicarlo. Credo che sia una buona abitudine dichiararlo.

Grazie alla collaborazione di Massimo Ghisalberti, che mi ha consigliato di riscrivere intere parti, il codice sorgente in Scala dovrebbe essere corretto e funzionante. Lo stesso vale per il codice Prolog, grazie al supporto di Andrea Sterbini.

Gli eventuali errori rimasti sono opera mia.

3. Le attività proposte

Di seguito trovate alcune proposte di attività didattiche, tutte legate al tema della lingua e la letteratura, partendo dalla grammatica ma finendo per toccare la poesia e le arti visive. Non seguono un ordine tematico particolare (ma in alcuni casi si rimanda dall'una all'altra), anche se le prime attività presentano codici sorgenti più corti e semplici da capire.

I titoli delle attività sono questi:

1. **E perché non un'automa?** grammatica e articoli
2. **Limericks digitali**: poesie, versi e rime
3. **Leggibile da un computer**: aspetti quantitativi dei testi
4. **Cento mila miliardi di problemi**: struttura linguistica e struttura matematica
5. **Arte e parole**: rappresentare visivamente un testo
6. **Braille, ASCII e co.**: trasformare un testo per renderlo accessibile
7. **Carte, forbici… e sassi**: giochi come dialoghi
8. **Sulle orme di Tolkien**: creare una pseudo-lingua
9. **Coniugo, coniugas, coniugat:** dal paradigma alla forma e viceversa
10. **Musica a programma**: generare musica e immagini

La struttura di ogni attività è la seguente:

a) un'introduzione al tema
b) una descrizione della progetto e della proposta didattica
c) una prima versione del codice sorgente in Kojo
d) una discussione del codice, con l'evidenziazione dei limiti e dei possibili sviluppi
e) altre versioni e altre discussioni
f) una valutazione finale
g) (una versione del codice in LibreLogo)
h) (una versione del codice in Prolog)

I sorgenti presentati sono stati testati, ma sono sicuramente migliorabili e, soprattutto, modificabili sia scegliendo degli esempi diversi, sia puntando a ottenere risultati diversi. Non sono l'unica soluzione possibile ad un problema, ma sono uno dei possibili approcci per affrontare un progetto. Se nel corso dell'attività si decide di prendere un'altra strada, può essere un'ottima idea,

Stefano Penge: Lingua, Coding e creatività – Copyright Anicia 2017

purché l'insegnante abbia almeno una vaga idea di dove si sta andando e non si trovi alla fine in un vicolo cieco che avrebbe l'effetto di spegnere l'entusiasmo degli studenti (ed il suo).

I costrutti sintattici nuovi non vengono sempre spiegati in dettaglio (per questo occorre fare riferimento ai tutorial), ma sono stati scelti in modo da essere il più possibile autoevidenti. In ogni caso, è fondamentale *provare* il codice prima di pensare di utilizzarlo.

Stefano Penge: Lingua, Coding e creatività – Copyright Anicia 2017