Tutorial LibreLogo

Nota: questo testo è parte del volume di Stefano Penge, Lingua, programmi e creatività. Coding con le materie umanistiche, Anicia, 2017.

Come e perché LibreLogo

Librelogo è una versione del linguaggio Logo che è eseguibile all'interno di LibreOffice. E' stato scritto da un ungherese, László Németh come un *plugin* di Writer:[1] significa che non è un programma a sé stante, ma che vive all'interno di un word processor. Questa scelta progettuale, molto forte, ha vantaggi e svantaggi. Per scrivere un programma, salvarlo, stamparlo insieme al risultato, non si deve fare niente di particolare, visto che si sta già dentro un programma di scrittura. D'altra parte ogni tanto l'interprete si confonde con i vari tipi di virgolette e quando si salva un documento, il file contiene molte più cose del codice (ad esempio, i disegni realizzati), il che può essere scomodo.

Per prima cosa, quindi, se non avete ancora installato la suite di LibreOffice,[2] fatelo. E' un'alternativa libera a Microsoft Office, o a Office 365 come si chiama oggi. Libera non significa solo gratuita, significa che il codice è opensource e chiunque – con le adeguate competenze – è messo in condizione di personalizzarlo e modificarlo. Oltre a poter eseguire il codice sorgente qui riportato, potete usare Write, Calc, Impress per tutti gli usi che vi vengono in mente.

Un punto di forza, che potrebbe essere superfluo per i lettore di questo testo ma che è una porta aperta ad usi potenti, è il fatto che in LibreLogo è possibile eseguire codice scritto in Python, o estendere le capacità del programma utilizzando le sue librerie; oppure accedere alle funzioni di PyUNO, che è il linguaggio di automazione di LibreOffice. Per esempio questa funzione esoterica (che NON è scritta in Logo, ma funziona) cancella l'ultimo oggetto grafico appena disegnato da LibreLogo:

```
TO delete_last_object
_.drawpage.remove(_.drawpage.getByIndex(
_.drawpage.getCount()-1))
END
```

Logo

LibreLogo utilizza delle convenzioni leggermente diverse da quelle del Logo dell'Università della California a Berkeley, che in qualche modo è considerato lo standard per questo linguaggio. Comunque non è difficile passare da una versione all'altra, quindi quello che imparate qui potete facilmente riusarlo con qualsiasi altra versione.

Il Logo è associato in maniera indissolubile con la tartaruga, e su questo argomento si trova molto materiale sia formativo che didattico (vedi in fondo a questo tutorial per qualche pista). Qui di

1 https://help.libreoffice.org/Writer/LibreLogo_Toolbar/it
2 https://it.libreoffice.org/

seguito però ci concentriamo prevalentemente sul linguaggio in generale, e in particolare sulle funzionalità per lavorare con caratteri e parole.

Dalla tartaruga bisogna però partire per capire il modello di interazione che sta dietro il Logo originale. Il contesto è questo: esiste un essere intelligente, capace fondamentalmente di fare poche cose:

1. ricevere delle istruzioni (delle sequenze di parole e caratteri)
2. provare a capirle
3. se non le capisce, segnalare il problema
4. se le capisce, eseguirle
5. imparare nuove istruzioni
6. eseguirle

Logo è infatti nato come un interprete, cioè un programma che sta in attesa di istruzioni e le analizza ed esegue man mano che arrivano. Kojo e Prolog, invece, sono compilatori, cioè hanno bisogno di avere tutto il codice sorgente a disposizione per trattarlo internamente in modo da poterlo eseguire velocemente e senza errori.

Per date un'idea della potenza di questa metafora dell'essere intelligente in ascolto, e dell'attenzione con cui era stata realizzata, ricordo la sorpresa provata la prima volta in un ambiente Logo ho provato a scrivere un espressione aritmentica come 3+5. La risposta dell'interprete (che potete provare anche voi usando un interprete online[3]) era curiosa:

```
DON'T KNOW WHAT TO DO WITH 8.
```

Intanto era ben chiaro che c'è un soggetto intelligente dall'altro lato dello schermo. Il messaggio non è "ERRORE 404" o qualcosa di altrettanto misterioso, ma "Non so che fare": la differenza, in termini di proiezione da parte del bambino, è enorme. Questo è molto diverso dal messaggio che nella stessa situazione produce ad esempio KTurtle[4] ("Non puoi inserire 3 qui"), o dalla mancanza di messaggio da parte di LibreLogo.

In secondo luogo, pur rilevando che c'è qualcosa che non va, Logo dice qualcosa di significativo e utile: "non so che fare con 8". Logo in realtà ha perfettamente capito la richiesta, l'ha eseguita, ha a disposizione il risultato, ma chiede all'utente cosa vuole fare con quel risultato. Insomma, sta cercando a sua volta di insegnare qualcosa.

Perché non sa cosa fare? Perché Logo è un linguaggio funzionale. Ci sono due tipi di istruzioni in Logo: i comandi e le funzioni. I comandi sono istruzioni che hanno un effetto sul mondo della tartaruga: la spostano, la girano, ne cambiano il colore, etc. Le funzioni sono comandi che restituiscono un risultato, e che per lo più NON hanno un effetto sul mondo.

3 + 5 non è un comando, ma è un modo per richiamare la funziona "somma" con due valori, 3 e 5. Siccome ogni funzione restituisce un valore, Logo chiede cosa deve fare con quel valore. Potrebbe dire "Vuoi che lo metta da parte? Che te lo scriva sullo schermo? Insomma, parla." Se avessimo scritto invece:

```
PRINT 3 + 5
```

Logo l'avrebbe trovato perfettamente sensato, e non si sarebbe lamentato.

3 http://www.calormen.com/jslogo/ La sintassi è quella dell'UCB Logo.

4 E' una versione OpenSource di Logo per ambiente Linux, che è parte del progetto Education KDE https://www.kde.org/applications/education/kturtle/.

Funzioni e variabili

Un aspetto chiave che rende così utili le funzioni è che ogni funzione può ricevere un valore da un'altra funzione. Si possono creare delle catene di funzioni lunghe a piacere. Per esempio, `RANDOM` è una funzione che estrae un elemento a caso dalla lista di parole che la segue.

```
RANDOM ["Logo","Prolog","Kojo"]
```

Il risultato (ad esempio, "Logo") può essere passato ad un'altra funzione, così:

```
PRINT RANDOM ( RANDOM ["Logo","Prolog","Kojo"])
```

Qui stiamo chiedendo a Logo di prendere un carattere a caso dalla parola estratta a caso dalla lista, e di scriverla.

Notate che c'è una differenza tra l'ordine in cui si scrivono le istruzioni e quello in cui vengono eseguite. Logo comincerà da *destra*, costruendo la lista di parole, poi prendendo una parola a caso, poi una lettera a caso da questa parola, e poi scrivendola.

Questo modo di scrivere programmi è tipico dello stile "funzionale", cui appartengono sia Logo che Kojo. Tuttavia è possibile memorizzare temporaneamente i risultati di funzioni in variabili con un nome, per riusarle in seguito:

```
L = RANDOM ( RANDOM ["Logo","Prolog","Kojo"])
```

Le variabili sono proprio questo: un modo per tenere da parte dei valori restituiti da funzioni. A volte sono comode solo per noi umani che leggiamo il codice sorgente, ma non sarebbero necessarie. L'istruzione che prima abbiamo scritto su una sola riga si poteva anche scrivere così:

```
Parola = RANDOM ["Logo","Prolog","Kojo"]
Lettera = RANDOM Parola
PRINT Lettera
```

Una variabile si crea quando serve scrivendo la sua etichetta, il segno uguale e il suo valore (un numero, un carattere, una serie di caratteri).

Un punto su cui inciampano i ragazzi è quel segno uguale. Non è l'uguale dell'aritmetica, quello che segnala che le espressioni a sinistra e a destra hanno lo stesso valore. Quella funzione è assolta da un altro segno, cioè ==. Invece un segno = significa che al momento di leggere quella riga del programma Logo crea un nuovo oggetto che in futuro sarà reperibile con quell'etichetta) e gli assegna il contenuto che sta a destra dell'uguale. La versione originale di Logo, per chiarezza, non usava il segno = ma richiedeva di scrivere:

```
MAKE "NOME
```

e al momento dell'uso,

```
PRINT :NOME
```

differenziando l'etichetta dal suo contenuto. Altri linguaggi scimmiottano verbi inglesi diversi, come SET o LET.

L'etichetta di una variabile può essere quasi qualsiasi sequenza di caratteri, purché cominci con una lettera. In alcuni linguaggi deve cominciare obbligatoriamente con un carattere particolare ($), o con una maiuscola. In LibreLogo maiuscole e minuscole si possono usare a piacere; solo che una variabile che si chiama "Lettera" e una che si chiama "lettera" sono due cose diverse, il che può essere imbarazzante quando si cerca di capire perché le cose non funzionano.

Attenzione anche alla differenza tra etichette di variabili e stringhe (cioè sequenze di caratteri).

Maestro e "Maestro" sono due cose molto diverse.

La versione senza virgolette è il nome di una variabile, per cui ha senso scrivere:

```
Maestro = "Gianni" PRINT Maestro
```

mentre:

```
"Maestro" = "Gianni"
```

è un errore. Avrebbe invece senso scrivere:

```
PRINT ("Maestro" == "Gianni")
```

che però significa un'altra cosa: dimmi se è vero che "Maestro" e "Gianni" sono parole identiche. La risposta è ovviamente no (ovvero: FALSE).

Dicevamo prima che Logo è in grado di imparare istruzioni nuove. Come fanno ad essere davvero *nuove*? In realtà quello che si può insegnare a Logo è un *nome* nuovo per un particolare insieme di di istruzioni, con un certo ordine o una certa logica.

Quando trovate scritto

```
TO FareUnaCosaNuova
          IstruzioneNota
          AltraIstruzioneNota
          ...
END
```

potete leggerlo così: sto insegnando a Logo una nuova istruzione ("FareUnaCosaNuova"), che da quel momento in poi potrà essere usata direttamente, invece di dover scrivere tutto quello che è compreso tra:

```
TO FareUnaCosaNuova
```

e:

```
END
```

In effetti il `TO` iniziale è una abbreviazione di "TO DO …"; nelle varie versioni italiane del linguaggio, il `TO` era spesso reso con "`PER`", nel senso di "`PER FARE`"

Un effetto di questa metafora di insegnamento/apprendimento è che bisogna ricordarsi di definire una nuova istruzione prima di chiedere a Logo usarla (il che è ovvio nella vita; ma non è ovvio in altri linguaggi).

Liste e blocchi

In Logo, come in Lisp da cui Logo deriva, la maniera generale di mettere insieme elementi è la lista. Una lista in Logo è può essere una cosa come

```
[1,2,3,4]
[Alberto,Mario,Francesco]
```

ma anche

```
[FORWARD 100 RIGHT 90]
```

Cioè: il corpo di una funzione (il blocco di istruzioni che Logo impara ad eseguire quando viene chiamata quella funzione) non è altro che una lista, né più né meno che una lista di numeri o parole. Questa sostanziale omogeneità permetteva di scrivere in Logo programmi capaci di modificarsi da soli, per quanto questa possibilità sembri fantascientifica.

Invece in LibreLogo[5] occorre stare attenti: questa è una lista

```
["Logo","Prolog","Kojo"]
```

questo invece è un blocco di istruzioni:

```
[ FD 100 RT 90 ]
```

La differenza sta negli spazi subito prima e subito dopo le parentesi, che sono *obbligatori* per le istruzioni e *vietati* per le liste. Se li dimenticate, LibreLogo protesta.

Una lista è una cosa un po' più sofisticata di un elenco. Un elenco è una serie di elementi. Una lista è una serie di elementi in cui il primo punta al secondo, il secondo al terzo, e così via. E' come se ogni elemento non sapesse altro che qual è l'elemento che lo segue.

In molti linguaggi esistono delle funzioni apposite per trattare le liste (vedi per esempio in Prolog). In UCBLogo (ma non in LibreLogo) ne esistono varie:

FIRST che restituisce il primo elemento di una lista

BUTFIRST che restituisce la lista meno il primo elemento

LAST che restituisce il primo elemento di una lista

BUTLAST che restituisce la lista meno l'ultimo elemento

Per esempio, per prendere il secondo elemento di una lista si può scrivere:

```
PRINT FIRST BUTFIRST [pippo pluto paperino]
```

In LibreLogo, invece, le liste sono trattate come elenchi, e per accedere al secondo elemento bisogna indicarlo numericamente:

```
Lista = ["pippo","pluto","paperino"]
PRINT  Lista[2]
```

Il senso, e l'utilità, di queste funzioni è che permettono di manipolare una lista anche senza avere idea della sua lunghezza. Vedremo un esempio più avanti.

Parametri

Le istruzioni, che restituiscano valori o no, posso avere dei parametri. Un parametro è una promessa e un'attesa. Alcune istruzioni possono essere inviate a Logo in maniera semplice, per esempio:

```
HOME
```

o

```
PENUP
```

ma sono veramente poche. La maggior parte hanno bisogno di un'informazione aggiuntiva, come `FORWARD` che muove la tartaruga nella direzione in cui è girata attualmente: avanti, si, ma di quanto?

`FORWARD` si aspetta un numero, che indica di quanti passi deve avanzare (esattamente com `RIGHT` che abbiamo visto prima, che si aspetta un angolo).

Se invece siamo noi a inventare un'istruzione, potremmo essere nella stessa situazione. Ad

5 LibreLogo utilizza come linguaggio "ospite" il Python (ovvero: l'interprete Logo è scritto a sua volta in Python). Per questo motivo, la sintassi di LibreLogo è un po' ibrida tra i due linguaggi.

esempio, vogliamo scrivere un'istruzione che sostituisca le "o" della parola "Logo" con delle "a". Esiste una apposita funzione di LibreLogo che si chiama SUB (per "substitute", immagino) che fa già questa operazione. Potremmo scrivere così:

```
TO cambia
	PRINT SUB (“o”, “a”, "Logo")
end
```

Quella che abbiamo appena scritto è una funzione di dubbia utilità, ma soprattutto di uso ristrettissimo: funziona solo con quella parola e con quella sostituzione. Se siamo interessati ad una generalizzazione (ed è difficile non esserlo) possiamo invece promettere che la funzione cambia verrà chiamata sempre con tre valori: il carattere da sostituire, quello da mettere al suo posto, la parola su cui operare:

```
TO cambia car1 car2 parola
	PRINT SUB (car1, car2, parola)
end
```

Ora possiamo usare questa funzione così:

```
cambia “o” “a” “Logo“
```

ma anche così

```
cambia “o” “a” “Prolog“
```

o così:

```
cambia “K” “D” “Kojo“
```

Attenzione ad una particolarità di LibreLogo: mentre la funzione SUB, che fa parte del linguaggio, richiede i parametri separati da virgole, cambia, che abbiamo definito noi, invece se li aspetta separati da spazi. E' una delle tante idiosincrasie dei linguaggi che fanno impazzire i programmatori.

I blocchi di istruzioni etichettati (le funzioni) sono comodi, perché evitano di riscrivere lo stesso codice più volte; ma si può fare di meglio. Questa è una tendenza generale dei linguaggi di programmazione: se si può scrivere meno codice, è meglio. Non tanto per la pigrizia dei programmatori, quanto perché si riduce la possibilità di commettere errori.

Immaginiamo di voler cambiare le "o" in "a" in tutti i nomi di linguaggi visti sopra. Dovremmo scrivere:

```
TO cambiatutto
cambia “o” “a” “Logo“
cambia “o” “a” “Prolog“
cambia “o” “a” “Kojo“
END
```

Ma possiamo essere più concisi, il che è una questione di stile ma anche di minor rischio di errore: più righe si scrivono, più aumenta la possibilità di commettere errori. Se abbiamo bisogno di applicare la stessa funzione più volte, a tutti gli elementi di una lista, possiamo utilizzare FOR, un costrutto ripete un'istruzioneche velocizza le cose:

```
FOR linguaggio in [“Logo”,“Prolog”,“Kojo”][
	cambia "o" "a" linguaggio
]
```

Cosa abbiamo scritto? Qualcosa come:

prendi ad uno ad uno gli elementi della lista, mettili nella variabile "linguaggio" e passali alla

funzione "cambia"

Se invece sappiamo già in anticipo quante volte vogliamo ripetere un'istruzione, allora si può usare una forma più tradizionale, come questa:

```
linguaggi = [“Logo”,“Prolog”,“Kojo”]
i = 1
REPEAT 3 [
        linguaggio = linguaggi[i]
        cambia "o" "a" linguaggio
        i = i + 1
]
```

Abbiamo detto sopra che le funzioni restituiscono un valore: ma come si fa, quando si crea una funzione, a specificare quale valore va restituito, se ce n'è più di uno? Con la funzione OUTPUT, che in altri linguaggi si chiama RETURN:

```
TO pesca_a_caso lista
        Nome = RANDOM lista
        OUTPUT 'Il linguaggio migliore è…' +Nome
END
Lista = ["Logo","Prolog","Kojo"]
PRINT pesca_a_caso Lista
```

La tartaruga

Per completezza, parliamo ora della parte di LibreLogo strettamente al movimento della tartaruga, cioè del robot virtuale che LibreLogo disegna sul foglio. Intanto notiamo che la tartaruga, come tutto quello che disegna, è un oggetto grafico, che si può spostare con il mouse, copiare e incollare in altri documenti.

I comandi possono essere scritti uno alla volta nel campo che appare a fianco dei bottoni della Toolbar Logo ed eseguite premento il tasto Invio. Per programmi più complessi, però, c'è un sistema migliore: si può scrivere il codice direttamente nel foglio, in qualsiasi punto. Dopodiché è sufficiente selezionare il codice e premere il bottone con il triangolo verde.

I comandi base sono questi:

Comando	**Parametri**	**Significato**
FORWARD	Quanto (in unità di schermo o in millimetri)	Muove nella direzione attuale
BACK	Quanto (in unità di schermo o in millimetri)	Muove nella direzione inversa
RIGHT	Angolo (in gradi)	Cambia la direzione verso destra
LEFT	Angolo (in gradi)	Cambia la direzione verso sinistra
FILL		Riempi una figura chiusa
HOME		Riporta la tartaruga al centro del foglio

Comando	Parametri	Significato
CLEARSCREEN		Come HOME ma pulendo ogni disegno realizzato finora

A questi si aggiunge una serie di istruzioni che non muovono la tartaruga ma ne caratterizzano il comportamento visivo:

Comando	Parametro	Significato
HIDETURTLE		Nasconde la tartaruga
SHOWTURTLE		Mostra la tartaruga
PENDOWN		Attiva la modalità di tracciamento di linee
PENUP		Disattiva la modalità di tracciamento di linee
PENCOLOR	Colore (una lista di tre valori: Rosso, Verde e Blu)	Stabilisce il colore delle linee
PENWIDTH	Quanto, da 0 a 9	Stabilisce la larghezza delle linee
PENJOINT	rounded, miter, bevel o none	Stabilisce la forma dei vertici
PENCAP	square, round o none	Stabilisce la forma delle estremità delle linee
PENSTYLE	Solid, dotted o dashed	Stabilisce il tratteggio
FILLCOLOR	Un nome di colore, oppure una lista di tre valori(Rosso, Verde e Blu), più uno per la trasparenza	Stabilisce il colore di riempimento delle figure chiuse con FILL
FILLSTYLE	Lo stile, un numero da 1 a 10	Stabilisce lo stile di riempimento
POSITION	Riga e colonna	Muove (o restituisce) la posizione della tartaruga
HEADING	Angolo	Cambia (o restituisce) la direzione in cui punta la tartaruga

Altre piste

Su Logo[6] esistono molti testi e manuali, a partire dal manuale originale, del 1974, scrito da Abelson, Goodman e Rudolph e scaricabile da https://dspace.mit.edu/handle/1721.1/6226 .

Ci sono manuali che trattano versioni specifiche di Logo, come MSWLogo (http://www.softronix.com/logo.html), FMSLogo (http://fmslogo.sourceforge.net/), o KTurtle per Linux (https://docs.kde.org/stable5/it/kdeedu/kturtle/).

Diversi articoli sul LOGO sono disponibili sul sito della Logo Foundation: http://el.media.mit.edu/logo-foundation/resources/papers/titleindex.html

Su LibreLogo invece – per di più in Italiano - non c'è molto. Fondamentale il Piccolo Manuale di LibreLogo[7] scritto da Andreas Formiconi con chiara intenzione didattica, e rivolto in particolare ai suoi studenti di Scienze della Formazione Primaria.

Infine due risorse non tecniche, ma culturali.

La prima è un tentativo a molte mani di tracciare un albero completo di tutti gli ambienti e linguaggi derivati dal LOGO: http://www.elica.net/download/papers/logotreeproject.pdf

La seconda è una lista degli articoli scritti da Seymour Papert: http://papert.org/works.html. Vale la pena di leggerli perché sono sempre sorgente di illuminazione.

6 La storia di LOGO può essere letta sulla pagina di Wikipedia: https://en.wikipedia.org/wiki/Logo_%28programming_language%29

7 http://iamarf.ch/unifi/Piccolo-manuale-LibreLogo.1.0.pdf