3.2 Limericks digitali

Argomento

I limericks sono dei componimenti poetici resi famosi da Edward Lear con i suoi nonsense. Ne ha scritti anche Rodari per mostrare i legami tra gioco, creazione linguistica e apprendimento. Ci sono tanti esempi di lavori didattici di insegnanti elementari su questo tema.[1]

I limericks si prestano bene, nella loro semplicità, per ragionare produttivamente in classe di poesia, di metri, di rime. Inoltre, per la loro tradizione umoristica, permettono di affrontare la poesia senza l'alone di magia che circonda l'opera del poeta.

Naturalmente oltre alla poesia, si possono andare a studiare altre forme chiuse di testo (una lettera, una ricetta) per distinguere tra struttura e contenuto, tra sintassi e lessico, per impadronirsi del concetto di vincolo e della sua relazione con la creatività. In generale, trovo che sia un modo di presentare il significato di "grammatica" (nel senso in cui viene usata in ambito informatico: delle regole che permettono di costruire un testo) che è meno ostile di quello tradizionale, e si può riconciliare con quello.

L'attività è divisa in tre parti: la scrittura di limericks, la generazione casuale di limericks (a mano) e infine la generazione automatica di limericks (scrivendo un programma che lo faccia per noi). Il motivo di questa sequenza sta nel vecchio adagio: se si è in grado di spiegare a qualcuno come fare una cosa significa che la si è capita bene.

Ora immagino facilmente che qualcuno possa obiettare che così si insegna a trattare i testi con la sola logica combinatoria, che è un virus che ormai contagia la maggior parte dei ragazzi fin dalle elementari[2] E' anche facile associare la logica combinatoria ai computer e scaricare sull'una le malefatte dell'altro, o viceversa.

Il punto di questa proposta di attività è proprio questo: mostrare sensibilmente come un programma che genera testi in una forma chiusa a partire da elementi minimi presenti in un archivio deve essere molto sofisticato per riuscire a produrre qualcosa di somigliante ad un testo scritto da un umano. Sofisticato non significa intelligente: significa che deve possedere le competenze per categorizzare, cercare, sostituire, mettere insieme. Ad un grado molto semplice, permette di costruire slogan; ad uno più complesso, email, fino ai plot dei romanzi alla Dan Brown[3] o ai finti articoli scientifici per ingannare i valutatori delle riviste[4]

1 Come questi di una quinta http://maestrazicchetto.blogspot.it/2012/03/i-nostri-limerick.html o questi altri https://sonoilmaiestro.wordpress.com/tag/limerick/.

2 Una descrizione più lunga del problema la trovate in questo post di Mariangela Galatea Vaglio: http://nonvolevofarelaprof.blogautore.espresso.repubblica.it/2017/01/28/fategli-fare-i-riassunti-piccolo-vademecum-perche-gli-alunni-non-diventino-analfabeti-funzionali/.

3 Come questo: http://www.polygen.org/it/grammatiche/cultura/ita/danbrown.grm

4 Come questo: http://www.elsewhere.org/pomo/

E anche così, la piacevolezza del risultato deriva più dagli aspetti incongrui, dalle assurdità involontarie (involontarie?) che da una coerenza semantica, che non c'è. Ma questo è un modo per insegnare a vedere questa differenza. Per chi ha presente il test di Turing, si può immaginarne una versione meno pretenziosa: si raccolgono insieme limericks prodotti dal programma e limericks scritti dai ragazzi, e si chiede di distinguerli. E poi di spiegare la differenza. Trovo che questa modalità sia più efficace di continuare a ripetere "dillo con parole tue".

Anche questa attività è basata su Kojo. In appendice trovate una versione in LibreLogo, che è forse più semplice da capire e modificare, e una in Prolog.

La proposta

Cominciamo con qualche esempio di limerick d'autore (per la precisione, di Lear):

There was an Old Man of Apulia
Whose conduct was very peculiar;
He fed twenty sons
Upon nothing but buns,
That whimsical Man of Apulia.

Un esempio italiano di Gianni Rodari (dalla Grammatica della Fantasia):

Un signore molto piccolo di Como
una volta salì in cima al Duomo
e quando fu in cima
era alto come prima
quel signore micropiccolo di Como.

Facciamo un'analisi di un limerick (proprio nel senso di "smontaggio negli elementi più piccoli"). Cerchiamo le regolarità: "signore" nella prima e ultima riga; "Como" al termine della prima e ultima riga; "Duomo" che fa rima con "Como" e "cima" che fa riga con "prima"; la lunghezza delle righe; gli accenti. Proviamo a descriverne la forma astratta: sono cinque righe, hanno una certa lunghezza (di lettere? O di sillabe?), degli accenti che si ripetono, delle corrispondenze tra righe. Si può rappresentare con uno schema visivo, dal consueto AABBA a qualcosa di un po' più creativo con frecce e colori.

Poi (o prima?) guardiamo ai vincoli narrativi: di cosa parla? Cosa fa per prima cosa? E in mezzo? E alla fine?

I. Viene presentato un personaggio con un aggettivo e il luogo provenienza.

II Questo personaggio si trova in una situazione, o incontra qualcuno/qualcosa; oppure si descrive una sua caratteristica particolare (un desiderio, una mania)

III L'effetto dell'incontro o del suo tentativo di realizzare il suo desiderio (prima parte)

IV L'effetto dell'incontro (seconda parte)

V Il riassunto finale: di nuovo il personaggio, ma con un aggettivo (spesso inventato) che lo caratterizza sintetizzandone la stranezza.

Proviamo a modificarlo: al posto di "Como" mettiamo "Alghero" al posto di "Duomo" mettiamo "pero". Invece di "un signore molto piccolo" possiamo mettere "un chirurgo daltonico"; invece di "salì in cima al" possiamo scrivere "si addormentò tra i rami di". Eccetera. Se rispettiamo le rime e le lunghezze delle parole, tutto fila liscio. Altrimenti zoppica un po', ma pazienza.

Dopo aver fatto un po' di esercizio di sostituzione, capito il meccanismo, proviamo a scriverne qualcuno da zero. Da dove si comincia? Dalla prima riga? Dalle rime? O si descrive a parole quello che vogliamo raccontare e poi lo si "mette in versi"?

Quali sono le difficoltà maggiori? Individuare un protagonista, metterlo nella situazione, farci venire in mente un'espansione (Rodari: il sasso nello stagno) o trovare le rime e costruire il verso con gli accenti giusti? E quali sono gli effetti più divertenti?

Immaginiamo che per Natale dobbiamo scrivere mille limericks per appenderli alle pareti dei corridoi. Non ce la faremo mai a scriverli a mano (quanto ci abbiamo messo per scriverne uno?). Potremmo invece costruire una macchina che lo fa per noi. Solo che bisogna spiegargli come si fa.

Partiamo con una serie di domande serie: che significa "generare automaticamente" un testo, o una poesia? Ci vuole intelligenza? Lo sa fare un computer (o un telefono)? Piccola digressione, se il contesto lo permette, sugli spambot. Perché è più facile, per un computer, risolvere un cruciverba o un sudoku che scrivere un testo? O meglio: perché impressiona di più?

Per spiegare al computer come fare, dobbiamo prima essere sicuri di aver capito.

Una prima versione ispirata a Raymond Queneau e al suo Cent Mille Milliards de Poèms (purtroppo non esiste una versione italiana; ci si può accontentare di approssimazioni). Si costruisce un archivio di versi e se ne estraggono 5. Non funziona bene, perché non si tiene conto delle rime (AABBA) e della lunghezza (8 sillabe, 5 sillabe). Una variante migliore è quella di dividere i versi per rima e per lunghezza e creare archivi separati: un barattolo per ogni rima ("omo","ero", "ito") e due colori diversi per i versi corti e quelli lunghi. Volendo si può proprio costruire il libro così come l'ha pensato Queneau: dieci pagine, ognuna con cinque versi; ogni pagina è tagliata quattro volte in orizzontale, in modo che si possano creare tanti diversi limericks semplicemente scegliendo un verso per pagina.

Domande: quante versioni diverse possiamo generare in questo modo? Come facciamo per essere sicuri di averle generate tutte (ogni verso dovrebbe essere usato 2000 volte)? Se aggiungiamo anche un solo verso, di quanto aumentano le varianti? (Si, sono domande di matematica e non di lingua. E allora?)

Una seconda versione: separiamo la struttura (sintassi) dagli elementi lessicali (semantica).

Facciamo una lista degli elementi che differenziano un limerick da un altro. Poi creiamo dei depositi di elementi: il protagonista, il luogo, la situazione, l'azione, il commento…; li scriviamo su foglietti di carta e li mettiamo nei barattoli etichettati in maniera corrispondente.

Peschiamo a caso un foglietto da ogni barattolo, li mettiamo uno dopo l'altro e quindi scriviamo un Limerick.

Domande: cosa dobbiamo cambiare perché sembri davvero una poesia? Dobbiamo aggiungere il condimento sintattico (articoli) o modificare le parole (concordanze).

Si tratta adesso di provare a costruire un programma che faccia le stesse cose che finora abbiamo fatto a mano. Quali operazioni abbiamo fatto? Elenchiamole. A seconda del linguaggio che abbiamo scelto, ci saranno modi più o meno comodi di memorizzare le liste di parole, di estrarne una a caso, di inserirla in una struttura testuale.

Versione 0

Termini introdotti

`Random, shuffle, nextInt`

`import`

La prima versione è semplice: basta pescare a caso dai vari depositi (liste di versi) e disporli in ordine. Esistono apposite funzioni che mescolano delle liste (`shuffle`) o che prendono un numero a caso (**`nextInt`**); solo che non fanno parte dell'ambiente standard di Kojo. Per usarle, dobbiamo prima importare la libreria (cioè la raccolta) di funzioni relative

```
import scala.util.Random
```

Una volta importate queste funzioni, possiamo utilizzarle, specificando *da dove* le abbiamo prese. Per pescare un elemento a caso da una lista, rimescoliamo la lista e prendiamo il primo elemento (`head`). Detto in altre parole:

```
Random.shuffle(riga).head
```

Per ripetere l'operazione su ogni riga, mettiamo le righe in una lista di liste di parole e usiamo **`for`**, un sistema molto semplice per estrarre gli elementi dalla lista che abbiamo già incontrato:

```
for (elemento <- lista){ fai qualcosa con elemento}
```

Ecco un esempio di codice sorgente:

```
// Limericks digitali 3.2.0
// Copyright Stefano Penge 2017
// Released under GPL v.3.0

import scala.util.Random // serve ad avere a disposizione le funzioni che generano numeri a caso

class Limerick {
// ogni riga è composta da più versi
val prima_riga = List ("Un signore molto piccolo di Como","Un chirurgo daltonico di Alghero","Un elefante a pallini di Milano")
var seconda_riga = List ("una volta salì in cima al Duomo","un giorno si addormentò sotto un pero","voleva giocare a tressette sul divano")
var terza_riga = List ("ma quando fu sulla cima","passarono tre ore","ma perdeva le carte")
var quarta_riga = List ("era alto come prima","e gli venne il mal di cuore","e restava in disparte")
var quinta_riga = List ("Quel signore micro piccolo di Como","Quell'assonnato chirurgo di Alghero","Quell'irascibile elefante di Milano")

// la lista db (che sta per "database", cioè archivio) li contiene tutti
```

```
var db = List( prima_riga, seconda_riga, terza_riga, quarta_riga, quinta_riga)

def fai_un_limerick {
   for (
     riga <- db
   )  println(pesca(riga))
 }
def pesca (riga: List [String]): String = {
// mescola una riga e prende il primo verso
  return Random.shuffle(riga).head
}
}

// main
 val limerick = new Limerick
 limerick.fai_un_limerick
```

Discussione

Otteniamo subito una serie di limerick originali come questo:

Un elefante a pallini di Milano

una volta salì in cima al Duomo

passarono tre ore

e restava in disparte

Quel signore micro piccolo di Como

Benissimo, funziona ma… a dire la verità il risultato non è un granché. Non rispetta le rime e si perde un po'. Dovremmo organizzare i versi in funzione della rima, come avevamo fatto nella versione manuale.

Versione 1

Termini introdotti

`array`

`while`

`nexInt`

`length`

`extends`

Potremmo usare Map, che è una struttura che accoppia una chiave ad un valore, connettendoli con una freccia verso destra. Per esempio, una riga del Limerick si potrebbe rappresentare così:

```
val prima_riga = Map(
    "rima"->"omo",
    "verso"->"Un signore molto piccolo di Como")
```

in modo da poter "estrarre" la rima di un verso, così:

```
 val una_rima = prima_riga("rima)
```

Come facciamo a mettere tutte varianti possibili insieme? Possiamo creare una fila (array) di Map, così:

```
Array(
    Map(
     "rima"->"omo",
     "verso"->"Un signore molto piccolo di Como"),
    Map(
    "rima"->"ero",
    "verso"->"Un chirurgo daltonico di Alghero"),
    ...
```

In più ci accorgiamo ora (ora?) che il primo e l'ultimo verso vanno presi abbinati, altrimenti si perde completamente il senso. Quindi dobbiamo procedere così: per la prima e la terza riga, prendiamo un verso a caso; per la seconda e la quarta, ne prendiamo uno che faccia rima con la prima e la terza. L'ultima riga deve far rima con la prima.

Quindi creiamo una struttura ancora più grande (Map) che contenga le quattro righe con tutte le varianti possibili:

```
val db = Map(
  1->Array(Map("rima"->"omo","verso"->"Un signore molto piccolo di Como"),Map("rima"->"ero",
"verso"->"Un chirurgo daltonico di Alghero"),Map("rima"->"ano","verso"->"Un elefante a pallini di
Milano")),
  2->Array(Map("rima"->"omo","verso"->"una volta salì in cima al Duomo"),Map("rima"->"ero","verso"-
>"un giorno si addorment sotto un pero"),Map("rima"->"ano","verso"->"voleva giocare a tressette sul
divano")),
...
```

Per prendere un verso che faccia rima con un altro, non ci basta più pescarlo a caso: dobbiamo continuare a pescare finché non ne troviamo uno che vada bene. Per questo possiamo utilizzare un costrutto che ripete un blocco di codice fintantoché (**`while`**) è vera la condizione all'inizio. In questo caso, la condizione è che la rima di quel verso sia diversa dalla rima precedente.

```
while ((riga(r)("rima") != rima){
       r = Random.nextInt(l)
}
return riga(r)
```

riga(r) è la riga r-esima dell'archivio, che abbiamo visto avere due chiavi: riga e verso. riga(r) ("rima") è quindi il contenuto della chiave rima del verso r.

`While` ("mentre") è un costrutto presente in quasi tutti i linguaggi per ripetere una serie di istruzione ma solo *finché* una certa condizione resta vera.[5]

Ma potrebbe succedere che, pescando a caso, non venga mai fuori un verso soddisfacente. Questo è un problema serio: se abbiamo appena detto al computer di provare finché non trova una soluzione, e non c'è nessuna garanzia che la soluzione esista, rischiamo di far partire un programma che non si ferma mai. "Mai" è una parola grossa; vuol dire semplicemente che il programma non è in grado di terminare da solo, ma dovrà essere interrotto dall'esterno (in Kojo, premendo il bottone rosso nella finestra del codice). Questa però non è una buona cosa: immaginate una lavatrice che non finisca mai il programma di lavaggio impostato e che bisogna interrompere staccando la corrente, lasciandola in uno stato indefinito. Dovremmo assicurarci che dopo un numero ragionevole di tentativi (quanti? Facciamo 20?) la funzione restituisca comunque un valore, anche se sbagliato. Oppure, meglio, che restituisca un messaggio di errore ("Aiuto, non trovo nulla").

Ecco un esempio un po' migliorato:

5 In alcuni linguaggi esiste anche il costrutto gemello `do… until`. Come per i gemelli, do ha una psicologia leggermente diverso da `while`: esegue un gruppo di istruzioni *fino al momento in cui* si verifica una condizione. E' come vedere il bicchiere mezzo pieno e mezzo vuoto: c'è chi spontaneamente preferisce l'uno e chi l'altro.

```
// Limericks digitali 3.2.1
// Copyright Stefano Penge 2017
// Released under GPL v.3.0

 import scala.util.Random


 class Metro {

 def trova_rima (rima: String, riga: Array[Map[String,String]]): Map[String,String] = {
       val l = riga.length
       var r = 0
       var volte = 0
       val basta = 20
       while ((riga(r)("rima") != rima) && (volte<basta)){
         r = Random.nextInt(l)
         volte = volte +1
       }
       return riga(r)
 }
 def pesca (riga: Array[Map[String,String]]): Map[String,String] = {
       val l = riga.length
       val r = Random.nextInt(l)
       return riga(r)
 }

 } // fine Metro

 class Limerick extends Metro {

 val db = Map(
   1->Array(
     Map(
     "rima"->"omo",
     "verso"->"Un signore molto piccolo di Como"),
     Map(
       "rima"->"ero",
       "verso"->"Un chirurgo daltonico di Alghero"),
     Map(
       "rima"->"ano",
       "verso"->"Un elefante a pallini di Milano")
     ),
   2->Array(
     Map("rima"->"omo","verso"->"una volta salì in cima al Duomo"),Map("rima"->"ero","verso"->"un
 giorno si addorment sotto un pero"),Map("rima"->"ano","verso"->"voleva giocare a tressette sul
 divano")
     ),
   3->Array(
     Map("rima"->"ima","verso"->"ma quando fu sulla cima"),Map("rima"->"ore","verso"->"passarono tre
 ore"),Map("rima"->"arte","verso"->"ma perdeva le carte")
     ),
   4->Array(
     Map("rima"->"ima","verso"->"era alto come prima"),Map("rima"->"ore","verso"->"e gli venne il
 mal di cuore"),Map("rima"->"arte","verso"->"e restava in disparte")
     ),
   5->Array(
     Map("rima"->"omo","verso"->"Quel signore micro piccolo di Como"),Map("rima"->"ero","verso"-
 >"Quell'assonnato chirurgo di Alghero"),Map("rima"->"ano","verso"->"Quell'irascibile elefante di
 Milano")
     )
   )

 def fai_un_limerick {

     val riga1 = pesca(db(1))
     println(riga1("verso"))

   // il secondo deve rimare col primo
     val riga2 = trova_rima(riga1("rima"),db(2))
     println(riga2("verso"))

     val riga3 = pesca(db(3))
     println(riga3("verso"))

   // il quarto deve rimare col terzo
     val riga4 = trova_rima(riga3("rima"),db(4))
```

```
    println(riga4("verso"))

  // l'ultimo deve rimare col primo
    val riga5 = trova_rima(riga1("rima"),db(5))
    println(riga5("verso"))
}

} // fine Limerick


// main

 val limerick = new Limerick
 limerick.fai_un_limerick
```

Abbiamo utilizzato due classi: Metro e Limerick. Questo perché alcune funzioni sono generali (potrebbero valere per altri tipi di componimenti) e altre sono tipiche del solo Limerick. Ma per fare in modo che Limerick possa utilizzare anche le funzioni di Metro, abbiamo esplicitato il fatto che ogni Limerick è anche un Metro usando il termine **`extends`**:

```
class Limerick extends Metro {…
```

Si dice che in questo caso che "Limerick è una sottoclasse di Metro". Nel tutorial trovate come al solito spiegazioni più approfondite; qui è importante pensare alle classi come ad una genealogia o come alla struttura dei generi e delle specie, da Aristotele a Linneo.

Discussione

Adesso possiamo ottenere opere pregevoli come questa:

Un elefante a pallini di Milano

voleva giocare a tressette sul divano

passarono tre ore

e gli venne il mal di cuore

Quell'irascibile elefante di Milano

Per ora va bene, come diceva quel tale cadendo dal decimo piano di un palazzo. Ma il nostro codice è un po' troppo rigido, il che non è mai una buona cosa: quando si progetta un programma, si cerca di renderlo flessibile, generalizzabile (ma non troppo) in modo da poter ottenere varianti ed estensioni senza dover riscrivere tutto da capo. E questa attenzione al futuro si traduce, spesso, nell'invenzione di strutture dati più potenti.

In questo caso possiamo lavorare un po' sulla struttura del testo. Possiamo immaginarlo come un foglio di carta in cui alcune parole siano state "bucate"; e possiamo preparare tanti fogli diversi che invece contengono solo le parole che sono posizionate in modo da riempire quei buchi. Tenendo fermo il primo foglio e sostituendo a turno gli altri possiamo ottenere frasi diverse. Prima di continuare con il codice Kojo, possiamo costruire fisicamente questa macchina per generare testi.

Versione 2

Riportiamo la struttura del Limerick in un testo con dei "buchi":

I Un <personaggio><aggettivo> di <provenienza>

II <tempo><desiderio><luogo>

III <evento> / IV<risultato>

V Quel <commento><personaggio> di <provenienza>

che potrebbe essere rappresentata così:

```
val  struttura = Map (
  1-> Array ("Un", pesca(db,"personaggio"), pesca(db,"aggettivo"), "di", pesca(db,"provenienza")),
  2->Array(pesca(db,"tempo"), pesca(db,"desiderio"), pesca(db,"luogo")),
  3->Array(pesca(db,"evento")),
  4->Array(pesca(db,"risultato")),
  5->Array("Quel",pesca(db,"commento"), pesca(db,"personaggio"), "di", pesca(db,"provenienza"))
  )
```

in cui la funzione `pesca(db,"personaggio")` fa esattamente quello che dice: pesca un personaggio a caso dal db.

Trascriviamo gli elementi in liste di parole, usando ancora un dizionario:

```
Map(
  "personaggio"->Array(" signore ", " chirurgo ", " elefante "),
  "aggettivo"->Array(" molto piccolo ", " attivissimo ", " a pallini "),
   ...
```

Scriviamo il codice sorgente che per ogni riga stampa la struttura, riempita con elementi a caso:

```
      struttura(q).foreach { elemento =>
        print(elemento)
      }
```

Non abbiamo tenuto conto di un sacco di cose (la rima, i riferimenti interni tra i versi), ma… funziona!

Ecco il codice completo:

```
// Limericks digitali 3.2.2
// Copyright Stefano Penge 2017
 // Released under GPL v.3.0

 import scala.util.Random

class Metro {

def pesca (db:Map[String,Array[String]],tipo: String) : String = {
  val l = db(tipo).length
  val r = Random.nextInt(l)
  return  db(tipo)(r)
}
} // fine Metro

class Limerick extends Metro  {

val db = Map(
  "personaggio"->Array(" signore "," chirurgo "," elefante "),
  "aggettivo"->Array(" molto piccolo "," attivissimo "," a pallini "),
  "provenienza"->Array(" Como"," Alghero"," Milano"),
  "tempo"->Array("una volta ","un giono ","tutti i mesi "),
  "desiderio"->Array(" salì "," si addormentò "," giocava a tressette "),
  "luogo"->Array(" in cima al Duomo"," sotto un pero"," sul divano"),
  "evento"->Array("ma quando fu sulla cima","ma passate tre ore","ma mangiando le carte"),
  "risultato"->Array("era alto come prima","gli venne il mal di cuore","rimase in disparte"),
```

```
  "commento"->Array(" micropiccolo "," assonnato "," affamato ")
  )
val  struttura = Map (
  1->Array("Un",pesca(db,"personaggio"), pesca(db,"aggettivo"),"di",pesca(db,"provenienza")),
  2->Array(pesca(db,"tempo"), pesca(db,"desiderio"), pesca(db,"luogo")),
  3->Array(pesca(db,"evento")),
  4->Array(pesca(db,"risultato")),
  5->Array ("Quel",pesca(db,"commento"), pesca(db,"personaggio"),"di",pesca(db,"provenienza"))
  )

def fai_un_limerick {
 val s = Seq(1,2,3,4,5)
 s.foreach{ q =>
      struttura(q).foreach { elemento =>
         print(elemento)
     }
     println("")
    }
 }

}

} // fine Limerick


// main

 val limerick = new Limerick
 limerick.fai_un_limerick
```

Un altro limite evidente è che stiamo usando solo personaggi maschili. Come facciamo a modificare la forma per tenere conto degli elementi pescati (ad esempio, quel/quella)? Già che ci siamo, dovremmo marcare gli elementi con le loro caratteristiche morfologiche (maschile, singolare)?

Discussione

Una volta migliorato il programma, prendiamo qualcuno dei limericks prodotti: sembrano davvero scritto da una persona? Quando si, quando no?

Si può fare il gioco cui accennavo nella premessa (Turing game): si mettono insieme limericks di autori umani e limericks prodotti dal programma, e poi si chiede a qualcuno di esterno (un ragazzo di un'altra classe, un genitore volenteroso) di riconoscere quelli digitali, e di dire come ha fatto. Probabilmente è troppo facile. Come si potrebbe migliorare ancora il programma?

Se volessimo costruire una poesia di tipo diverso, cosa dovremmo cambiare?

E se volessimo scrivere un programma in grado di produrre qualsiasi poesia?

Conclusioni

Riprendiamo le domande che abbiamo posto durante l'attività e proviamo a rispondere di nuovo.

Che differenza c'è, in termini di "potenza espressiva", tra la prima e la seconda versione? La seconda è più rustica, certo, ma è molto generale. Che altro ci si potrebbe fare?

Chi è l'autore dei nostri limericks? Il computer o noi? Che vuol dire "autore"? Ci possono essere più autori di uno stesso testo? E se qualcuno traduce una poesia? E se la modifica, chi è l'autore? Viceversa, se scrive una poesia ma usa un correttore automatico, o un dizionario di sinonimi?

Adesso andiamo a (ri)leggere una poesia qualsiasi, già conosciuta. Per esempio, un canto dell'Inferno. Guardiamola con gli occhi di un programma: la potrebbe costruire? Quanto è difficile scrivere una poesia? Cosa bisogna "sapere"?

O invece di una poesia, leggiamo una lettera, o una pubblicità. E' più facile o più difficile? Ci sono tipi di testi che *non* si potrebbero generare in questo modo?

Che succederebbe se si scoprisse che gli oroscopi, le previsione del tempo, gli articoli di un giornale, le voci di un'enciclopedia, o le pagine di un libro, fossero state scritte da un programma? Quali di queste ci darebbero più fastidio? Perché?

Probabilmente tutta l'attività risulterebbe più gradita agli studenti se invece della struttura limerick si fosse scelta la cronaca di una partita di calcio o quella di un reality show. Si sarebbe persa tutta la parte dedicata alla rima e alla forma chiusa, ma se ne sarebbe guadagnato in partecipazione. Sta al docente interpretare la sensibilità della propria classe e scegliere il modello più funzionale.

Altre idee possono venire dalla lettura di questo divertente manualetto[6] di Umberto Eco, in cui vengono raccolti degli esercizi svolti da studenti del secondo anno di Scienze della Comunicazione (1994-1995), basati a loro volta su schemi resi famosi dal gruppo dell'OuLiPo.[7] In particolare, si potrebbe cercare di costruire una versione digitale delle sostituzioni in S^n: si prendono le parole di un testo molto noto e si sostituiscono con la n-sima parola che le segue nel vocabolario.

6 Povero Pinocchio. Giochi linguistici degli studenti al Corso di Comunicazione, Bologna, Comix, 1995
7 https://it.wikipedia.org/wiki/OuLiPo

3.2.1 Versione il LibreLogo

Si tratta della versione in LibreLogo del terzo esempio di codice, quello del testo con buchi. Anche in questo caso, per semplificare, non si fa nessun controllo sulle rime e sui riferimenti interni. Il codice però è molto semplice, facilmente espandibile e adatto anche alla scuola primaria.

```
 ;Per eseguire il codice è sufficiente incollarlo dentro una pagina di
 ;Libre Office Writer in cui sia stata installata la ToolBar LibreLogo e poi premere sul bottone
 verde.
 ;Viene mostrato un limerick generato a caso in una finestra di dialogo.
; Copyright Stefano Penge 2017
; Released under GPL v.3.0

 TO limerick
 struttura = [verso1,verso2,verso3,verso4,verso5]
 poesia = ""
 FOR i IN struttura [ poesia = poesia + i ]
 PRINT poesia
 END

 TO verso1
 OUTPUT "Un" + personaggio + aggettivo + "di" + provenienza
 END

 TO verso2
 OUTPUT tempo + desiderio + luogo
 END

 TO verso3
 OUTPUT evento
 END

 TO verso4
 OUTPUT risultato
 END

 TO verso5
 OUTPUT "Quel" + commento + personaggio + "di" + provenienza
 END

 TO personaggio
 OUTPUT RANDOM [' signore ' ,' elefante ',' chirurgo ']
 END

 TO aggettivo
 OUTPUT RANDOM [' molto piccolo ',' a pallini ',' efficientissimo ']
 END

 TO provenienza
 OUTPUT RANDOM [' Como',' Alghero',' Milano']
 END

 TO tempo
 OUTPUT RANDOM [' una volta ',' un giorno ',' tutti i mesi ']
 END

 TO luogo
 OUTPUT RANDOM [' in cima al Duomo ',' sotto un pero ',' sul divano ']
 END

 TO desiderio
 OUTPUT RANDOM [' salì ',' si addormentò ',' giocava a tressette ']
 END

 TO evento
 OUTPUT RANDOM [' ma quando fu sulla cima ',' ma passate tre ore ',' ma mangiando le carte ']
 END

 TO risultato
 OUTPUT RANDOM [' era alto come prima ',' gli venne il mal di cuore' ,' rimase in disparte ']
 END

 TO commento
 OUTPUT RANDOM [' micropiccolo ',' assonnato ',' affamato ']
```

```
END

limerick
```

3.2.2: Versione Prolog

Nota: per eseguire il codice occorre prima salvarlo come file di testo (ad esempio come limericks.pl), poi lanciare l'interprete Prolog (es. swipl o gprolog), infine caricare il sorgente scrivendo: consult(limericks). Anche questa versione è basata sul testo a buchi. Per mostrare le caratteristiche del Prolog, qui ci si limita a generare TUTTE le possibili varianti del limerick. Se si propone il goal "limerick" ne scrive una; per vedere le altre, premere TAB. Si potrebbe modificare in modo da pescare casualmente tra gli elementi.

```
// Copyright Stefano Penge 2017
// Released under GPL v.3.0

/* Vocabolario */
personaggio(signore).
personaggio(elefante).
personaggio(chirurgo).
aggettivo("molto piccolo").
provenienza(como).
provenienza(alghero).
provenienza(milano).
tempo('una volta').
tempo('un giorno').
tempo('tutti i mesi').
desiderio('salì').
desiderio('si addormentò').
desiderio('giocava a tressette').
luogo('in cima al duomo').
luogo('sotto un pero').
luogo('sul divano').
evento('ma quando fu sulla cima').
evento('ma passate tre ore').
evento('ma mangiando le carte').
risultato('era alto come prima').
risultato('gli venne il mal di cuore').
risultato('rimase in disparte').
commento('micropiccolo').
commento('assonnato').
commento('affamato').


/* Grammatica */
verso1(Verso):-
        personaggio(Personaggio),
        aggettivo(Aggettivo),
        provenienza(Provenienza),
        atomic_list_concat(["Un",Personaggio, Aggettivo, "di",Provenienza]," ",Verso).
verso2(Verso):-
        tempo(Tempo),
        desiderio(Desiderio),
        luogo(Luogo),
        atomic_list_concat([Tempo, Desiderio, Luogo]," ",Verso).
verso3(Verso):-
        evento(Evento),
        atomic_list_concat([Evento],Verso).
verso4(Verso):-
        risultato(Risultato),
        atomic_list_concat([Risultato],Verso).
verso5(Verso):-
        personaggio(Personaggio),
        commento(Commento),
        provenienza(Provenienza),
        atomic_list_concat(["quel",Commento, Personaggio, "di",Provenienza]," ",Verso).

limerick:-
        verso1(Verso1),
        verso2(Verso2),
        verso3(Verso3),
        verso4(Verso4),
        verso5(Verso5),
        ListaVersi = [Verso1, Verso2,Verso3,Verso4,Verso5],
        atomic_list_concat(ListaVersi, "\n",Frase),
```

```
write(Frase).
```