Cristiano Chesi (chesi@media.unisi.it)

CHILDES, ESPRESSIONI REGOLARI E PC-KIMMO

Obiettivi del laboratorio

(1) Esplorare un corpus semi-strutturato (Childes)
(2) Apprezzare l'importanza degli indici di frequenza delle parole
(3) Comprendere le difficoltà e le inefficienze correlate alla descrizione degli elementi lessicali attraverso espressioni regolari
(4) Riflettere su eventuali astuzie (computazionalmente molto efficienti) per cogliere fenomeni linguistici produttivi

Strumenti di analisi ed Espressioni Regolari per CHILDES

(5) Cos'è CHILDES?

Childes (Child Language Data Exchange System) è un archivio di trascrizioni spontanee di bambini (solitamente dai 14 mesi ai quattro anni di età) che interagiscono con adulti in varie situazioni.
Generalmente ogni trascrizione si riferisce ad una conversazione di durata variabile dai 20 ai 60 minuti.
Tali trascrizioni sono codificate secondo il formato standardizzato CHAT che prevede le seguenti convenzioni:

Obligatory Headers
@Begin marks the beginning of a file
@End marks the end of the file
@ID: code for a larger database
@Participants: lists actors in a file

Constant Headers
@Age of XXX: marks a speaker's age
@Birth of XXX: shows date of birth of speaker
@Coder: people doing transcription and coding
@Coding: version of CHAT coding
@Education of XXX: indicates educational level of speaker
@Filename: shows name of file
@Font: sets the default font for the file
@Group of XXX: indicates the subject's group in group studies
@Language: the principal language of the transcript
@Language of XXX: language(s) spoken by a given participant
@SES of XXX: ndicates socioeconomic status of speaker
@Sex of XXX: indicates gender of speaker
@Stim: indicates stimulus for elicited production
@Transcriber: gives the transcriber's name or initials
@Warning: marks defects in file

Changeable Headers
@Activities: component activities in the situation
@Bg and @Bg: begin gem
@Bck: backgrounding information
@Comment: comments
@Date: date of the interaction
@Eg and @Eg: end gem
@g: simple gems
@Location: geographical location of the interaction
@New Episode: point at which a new episode begins and old one ends
@Room Layout: configuration of furniture in room
@Situation: general atmosphere of the interaction
@Tape Location: footage markers from tape
@Time Duration: beginning and end times
@Time Start: beginning time

Words
@ special form markers
xxx unintelligible speech, not treated as a word
xx unintelligible speech, treated as a word
yyy unintelligible speech transcribed on %pho line, not treated as a word
yy unintelligible speech transcribed on %pho line,treated as a word
www untranscribed material
0 actions without speech
& phonological fragment
[?] best guess
text(text)text noncompletion of a word
0word omitted word
0*word ungrammatical omission
00word (grammatical) ellipsis

Basic Utterance Terminators
. period
? question
! exclamation

Tone Unit Marking
-? rising final contour
-! final exclamation contour
-. falling final contour
-'. rise-fall final contour
-,. fall-rise final contour
-, level nonfinal contour
-_ falling nonfinal contour
- low level contour
-' rising nonfinal contour
, syntactic juncture
,, tag question
pause between words
-: previous word lengthened

Prosody Within Words
/ stress
// accented nucleus

/// contrastive stress
: lengthened syllable
:: pause between syllables
^ blocking

Special Utterance Terminators
+... trailing off
+..? trailing off of a question
+!? question with exclamation
+/. interruption
+/? interruption of a question
+//. self-interruption
+//? self-interruption of a question
+"/. quotation follows on next line
+". quotation precedes
+" quoted utterance follows
+^ quick uptake
+< "lazy" overlap marking
+, self-completion
++ other-completion
[c] clause delimiter

Scoped Symbols
·%mov:"*"_0_1073· time alignment marker
[=! text] paralinguistics, prosodics
[!] stressing
[!!] contrastive stressing
["] quotation marks
[= text] explanation
[: text] replacement
[0 text] omission
[:=x text] translation
[=? text] alternative transcription
[%xxx: text] dependent tier on main line
[% text] comment on main line
[$text] code on main tier
[?] best guess
[>] overlap follows
[<] overlap precedes
<text> [<>] overlap follows and precedes
[>number][<number] overlap enumeration
[/] retracing without correction
word (*N) word repetition
[//] retracing with correction
[///] retracing with reformulation
[/-] false start without retracing
[/?] unclear retrace type
[*] error marking
[+ text] postcode
[+ bck] exclued utterance
[+ trn] inclued utterance

Dependent Tiers
%act: actions
%add: addressee
%alt: alternative transcription
%cod: general purpose coding
%coN: additional general coding categories, co1, co2
%coh: cohesion tier
%com: comments by investigator
%def: codes from SALT
%eng: English translation
%err: error coding
%exp: explanation
%fac: facial actions
%flo: flowing version
%gls: target language gloss for unclear utterance
%gpx: gestural and proxemic activity
%int: intonation
%lan: language
%mod: model or target phonology
%mor: morphemic semantics
%mov: movie tier
%par: paralinguistics
%pho: phonetic transcription
%sit: situation
%snd: sonic CHAT sound tier
%spa: speech act coding
%syn: syntactic structure notation
%ssy simple syntactic categories
%tim: time stamp coding

Dependent Tier Special Codes
$ indicates codes
$=N occurs for N following utterances
$sc=N-M codes refer to words N through M on the main tier
<bef> occurrence before an utterance
<aft> occurrence after an utterance

Error Coding
$= source of an error in the %err line
= placed between error and target
; separates errors on %err line

Morphosyntactic Coding
- suffix marker
prefix marker
+ compound or rote form marker
~ clitic marker
~# placed after separable prefix
~- placed before second part of discontinuous morpheme
& fusion marker
= English translation for the stem
-0 omitted affix
-0* incorrectly omitted affix
| follows part-of-speech on %mor line
& nonconcatenated morpheme in %mor line
# prefix delimiter on %mor line
+ (Plus) compound delimiter on %mor line
- (Dash) suffix delimiter on %mor line
: feature fusion on %mor line
~ (Tilde) clitic delimiter on %mor line
0 precedes omitted element
0* precedes incorrectly omitted element

(6) ecco un **breve esempio** di trascrizione:

```
@UTF8
@Begin
@Participants:    CHI Cam Target_Child, DON Mother
@ID:  it|romance|CHI|3;4.9|female|||Target_Child||
@ID:  it|romance|DON|||||Mother||
@Age of CHI:      3;4.9
@Sex of CHI:      female
@Birth of CHI:    3-MAY-1988
@Date:      12-SEP-1991
@Filename:  cx40
@Situation: registrazione seduta
*DON: ma non si sente # prima si registra, e dopo # … come una cosa che
      prima si scrive, e dopo si legge .
%sit: come sempre, quando Camilla vede il registratore, le viene in mente
      di ascoltare le cassette, e vuole impossessarsene per premere i
tasti
*DON: ora bisogna registrarlo # adesso giochiamo un attimino, scusa !
*CHI: io io c'ho messo io ho preso quello che volevo .
*DON: quale volevi ?
*CHI: io volevo questo .
*DON: e e cosa quale, quello .
*CHI: … un nastrino .
*DON: si ma cosa, che canzoni ci sono, sopra .
*CHI: non lo so .
*DON: come non lo sai ?
*CHI: la scuola della &scuo e &dimin .
%act: canticchia, inventando un po' le parole
*DON: ah@i !
*CHI: puffi.
[...]
@End
```

(7) **come si usa CHILDES?**

Il sistema è corredato di un sistema di analisi chiamato **CLAN** (**Computerized Language Analysis**) che mette a disposizione dell'utente una serie di strumenti che permettono di effettuare varie operazioni:

CHAINS	Tracks sequences of interactional codes across speakers.
CHECK	Verifies the accuracy of CHAT conventions in files.
CHIP	Examines parent-child repetition and expansion.
CHSTRING	Changes words and characters in CHAT files.
COLUMNS	Reformats the transcripts into columnar form.
COMBO	Searches for complex string patterns.
COOCUR	Examines patterns of co-occurence between words.
DATES	Uses the date and birthdate of the child to compute age.
DIST	Examines patterns of separation between speech act codes.
DSS	Computes the Developmental Sentence Score.
FLO	Reformats the file in simplified form.
FREQ	Computes the frequencies of the words in a file or files.
FREQMERG	Combines the outputs of various runs of FREQ.
FREQPOS	Tracks the frequencies in various utterance positions.
GEM	Finds areas of text that were marked with gem markers.
GEMFREQ	Computes frequencies for words inside gem markers.
GEMLIST	Lists the pattern of gem markers in a file or files.
KEYMAP	Lists the frequencies of codes that follow a target code.
KWAL	Searches for word patterns and prints the line.
MAKEDATA	Converts data formats for CHAT files across platforms.
MAKEMOD	Adds a %mod line for the target SAMPA phonology
MAXWD	Finds the longest words in a file.
MLT	Computes the mean length of turn.

MLU Computes the mean length of utterance.
MODREP Matches the child's phonology to the parental model.
MOR Inserts a new tier with part-of-speech codes.
PHONFREQ Computes the frequency of phonemes in various positions.
POST Probabilistic disambiguator for the %mor line
POSTLIST Displays the patterns learned by POSTTRAIN
POSTTRAIN Trains the probabilistic network used by POST
RELY Measures reliability across two transcriptions.
SALTIN Converts SALT files to CHAT format.
STATFREQ Formats the output of FREQ for statistical analysis.
TEXTIN Converts straight text to CHAT format.
TIMEDUR Uses the numbers in sonic bullets to compute overlaps.
VOCD Computes the VOCD lexical diversity measure.
WDLEN Computes the length of utterances in words.

(8) L'uso di espressioni regolari:

From "manual-entry: GREP" in Gnu Emacs 20.7.4.

REGULAR EXPRESSIONS
A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

Grep understands two different versions of regular expression syntax: "basic" and "extended." In GNU grep, there is no difference in available functionality using either syntax. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

A list of characters enclosed by [and] matches any single character in that list; if the first character of the list is the caret ^ then it matches any character not in the list. For example, the regular expression [0123456789] matches any single digit. A range of ASCII characters may be specified by giving the first and last characters, separated by a hyphen. Finally, certain named classes of characters are predefined. Their names are self explanatory, and they are:
[:alnum:], [:alpha:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], and [:xdigit:].
For example, [[:alnum:]] means [0-9A-Za-z], except the latter form is dependent upon the ASCII character encoding, whereas the former is portable. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.)
Most metacharacters lose their special meaning inside lists.
To include a literal] place it first in the list. Similarly, to include a literal ^ place it anywhere but first. Finally, to include a literal - place it last.

The period . matches any single character. The symbol \w is a synonym for [[:alnum:]] and \W is a synonym for [^[:alnum:]].

The caret ^ and the dollar sign $ are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols \< and \> respectively match the empty string at the beginning and end of a word. The symbol \b matches the empty string at the edge of a word, and \B matches the empty string provided it's not at the edge of a word.

A regular expression may be followed by one of several repetition operators:

? The preceding item is optional and matched at most once.
* The preceding item will be matched zero or more times.
+ The preceding item will be matched one or more times.
{n} The preceding item is matched exactly n times.
{n,} The preceding item is matched n or more times.
{n,m} The preceding item is matched at least n times, but not more than m times.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator |; the resulting regular expression matches any string matching either subexpression.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference \n, where n is a single digit, matches the substring previously matched by the nth parenthesized subexpression of the regular expression.

In basic regular expressions the metacharacters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \).

Traditional egrep did not support the { metacharacter, and some egrep implementations support \{ instead, so portable scripts should avoid { in egrep patterns and should use [{] to match a literal {.

GNU egrep attempts to support traditional usage by assuming that { is not special if it would be the start of an invalid interval specification. For example, the shell command egrep '{1' searches for the two-character string {1 instead of reporting a syntax error in the regular expression. POSIX.2 allows this behavior as an extension, but portable scripts should avoid it.

OPTIONS

-A NUM, --after-context=NUM
Print NUM lines of trailing context after matching lines.

-B NUM, --before-context=NUM
Print NUM lines of leading context before matching lines.

-C [NUM], -NUM, --context[=NUM]
Print NUM lines (default 2) of output context.

-b, --byte-offset
Print the byte offset within the input file before each line of output.

-c, --count
Suppress normal output; instead print a count of matching lines for each input file. With the -v, --invert-match option (see below), count non-matching lines.

-E, --extended-regexp
Interpret PATTERN as an extended regular expression

-e PATTERN, --regexp=PATTERN
Use PATTERN as the pattern; useful to protect patterns beginning with -.

-F, --fixed-strings
Interpret PATTERN as a list of fixed strings, separated by newlines, any of which is to be matched.

-G, --basic-regexp
Interpret PATTERN as a basic regular expression. This is the default.

-i, --ignore-case
Ignore case distinctions in both the PATTERN and the input files.

-n, --line-number
Prefix each line of output with the line number within its input file.

-v, --invert-match
Invert the sense of matching, to select non-matching lines.

-x, --line-regexp
Select only those matches that exactly match the whole line.

DIAGNOSTICS
Normally, exit status is 0 if matches were found, and 1 if no matches were found. (The -v option inverts the sense of the exit status.) Exit status is 2 if there were syntax errors in the pattern, inaccessible input files, or other system errors.

(9) Una “semplice” espressione regolare per trovare gli imperfetti in italiano:

comando base:
grep -i -n -B3 -A3 –E
"(([[:space:]]|[[:punct:]])[[:alpha:]]*[aei](vo|vi|va|vamo|vate|vano)([[:space:]]|[[:punct:]])|([[:space:]]|[[:punct:]])[eE]r(o|i|a|avamo|avate|ano)([[:space:]]|[[:punct:]]))";

primo filtro:
grep -i -v -E
"(([[:space:]]|[[:punct:]])[dbl][lrea](a)?v[oaei](te)?([[:space:]]|[[:punct:]])|([[:space:]]|[[:punct:]])([Ss]c|[Aa]r)riv[ioea]([[:space:]]|[[:punct:]])|([[:space:]]|[[:punct:]])[cC]attiv|([[:space:]]|[[:punct:]])(copri)?divan|([[:space:]]|[[:punct:]])(s)?c(h)?(i)?av[iaeo]([[:space:]]|[[:punct:]])|[ae]tevi([[:space:]]|[[:punct:]]))"

secondo filtro:
grep -i -B3 -A3 -E
"*CHI:([[:space:]]|[[:punct:]]|[[:alpha:]])*(([[:space:]]|[[:punct:]])[[:alpha:]]*[aei](vo|vi|va|vamo|vate|vano)([[:space:]]|[[:punct:]])|([[:space:]]|[[:punct:]])[eE]r(o|i|a|avamo|avate|ano)([[:space:]]|[[:punct:]]))"

Qualche osservazione da verificare

1. Verificare quali sono gli elementi a più alta frequenza nei vari file

2. Trovare un'espressione regolare che catturi tutti e soli gli articoli

3. Trovare un'espressione regolare che colga tutte e sole le preposizioni

4. Individuare tra le produzioni del bambino delle forme ortografiche nominali o aggettivali scorrette (es. "chetto" per "questo") e cercare di creare un'unica espressione regolare che colga sia la forma corretta che quella scorretta

5. Cercare di cogliere con un'unica espressione regolare una forma corretta ("questo"), le occorrenze scorrette (ragionevolmente riconducibili alla forma corretta, "chetto") e le flessioni grammaticali possibili ("questi", "questa", "queste")

6. Cercare di cogliere con un'espressione regolare fenomeni di accordo articolo-nome

7. cercare le iper-regolarizzazioni del bambino (es. *Mario ha corruto qua!*)

8. cercare le sostituzioni/omissioni di ausiliari (es. *ho caduto*)

Analisi morfologica con PC-Kimmo

WHAT IS PC-KIMMO?
PC-KIMMO is a new implementation for microcomputers of a program dubbed KIMMO after its inventor Kimmo Koskenniemi (see Koskenniemi 1983). It is of interest to computational linguists, descriptive linguists, and those developing natural language processing systems. The program is designed to generate (produce) and/or recognize (parse) words using a two-level model of word structure in which a word is represented as a correspondence between its lexical level form and its surface level form.
Work on PC-KIMMO began in 1985, following the specifications of the LISP implementation of Koskenniemi's model described in Karttunen 1983. The coding has been done in Microsoft C by David Smith and Stephen McConnel under the direction of Gary Simons and under the auspices of the Summer Institute of Linguistics. The aim was to develop a version of the two-level processor that would run on an IBM PC compatible computer and that would include an environment for testing and debugging a linguistic description. The PC-KIMMO program is actually a shell program that serves as an interactive user interface to the primitive PC-KIMMO functions. These functions are available as a C-language source code library that can be included in a program written by the user.

A PC-KIMMO description of a language consists of two files provided by the user:
a **rules** file, which specifies the alphabet and the phonological (or spelling) rules, and
a **lexicon** file, which lists lexical items (words and morphemes) and their glosses, and encodes morphotactic constraints.

The theoretical model of phonology embodied in PC-KIMMO is called **two-level phonology**. In the two-level approach, phonological alternations are treated as direct correspondences between the underlying (or lexical) representation of words and their realization on the surface level. For example, to account for the rules of English spelling, the surface form spies must be related to its lexical form `spy+s as follows (where ` indicates stress, + indicates a morpheme boundary, and 0 indicates a null element):

Lexical Representation: ` s p y + 0 s
Surface Representation: 0 s p i 0 e s

Rules must be written to account for the special correspondences `:0, y:i, +:0, and 0:e. For example, the two-level rule for the y:i correspondence looks like this (somewhat simplified):
y:i => @:C___+:0

Notice that the environment of the rule is also specified as a string of two-level correspondences. Because two-level rules have access to both underlying and surface environments, interactions among rules can be handled without using sequential rule ordering. All of the rules in a two-level description are applied simultaneously, thus avoiding the creation of intermediate levels of derivation (an artifact of sequentially applied rules).
The two functional components of PC-KIMMO are the generator and the recognizer. The generator accepts as input a lexical form, applies the phonological rules, and returns the corresponding surface form. It does not use the lexicon. The recognizer accepts as input a surface form, applies the phonological rules, consults the lexicon, and returns the corresponding lexical form with its gloss. Figure 1 shows the main components of the PC-KIMMO system.

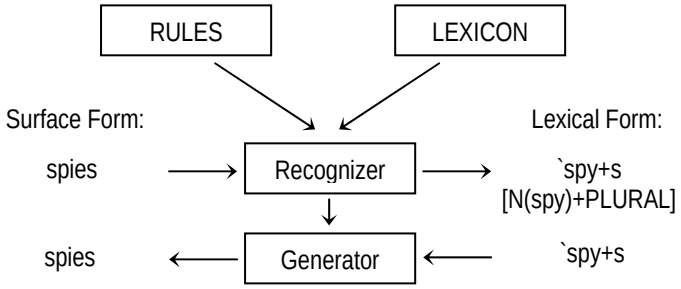

Figure 1: Main components of PC-KIMMO

The rules and lexicon are implemented computationally using finite state machines. For example, the two-level rule shown above for the y:i correspondence must be translated into the following finite state table for PC-KIMMO to use:

```
    | @  y  +  @
    | C  i  0  @
  ------------------
1:| 2  0  1  1
2:| 2  3  2  1
3:| 0  0  1  0
```

(Note: as of May 1991, there is a beta test vesion of a rule compiler available, called KGEN. See below for more information.)
Around the components of PC-KIMMO shown in figure 1 is an interactive shell program that serves as a user interface. When the PC-KIMMO shell is run, a command-line prompt appears on the screen. The user types in commands which PC-KIMMO executes. The shell is designed to provide an environment for developing, testing, and debugging two-level descriptions. Among the features available in the user shell are:
on-line help;
commands for loading the rules and lexicon files;
ability to generate and recognize forms entered interactively from the keyboard;
a mechanism for reading input forms from a test list on a disk file and comparing the output of the processor to the correct results supplied in the test list;
provision for logging user sessions to disk files;
a facility to trace execution of the processor in order to debug the rules and lexicon;
other debugging facilities including the ability to turn off selected rules, show the internal representation of rules, and show the contents of selected parts of the lexicon; and
a batch processing mode that allows the shell to read and execute commands from a disk file.

Because the PC-KIMMO user shell is intended to facilitate development of a description, its data-processing capabilities are limited. However, PC-KIMMO can also be put to practical use by those engaged in natural language processing. The primitive PC-KIMMO functions (including load rules, load lexicon, generate, recognize) are available as a source code library that can be included in another program. This means that the users can develop and debug a two-level description using the PC-KIMMO shell and then link PC-KIMMO's functions into their own programs.

WHO IS PC-KIMMO FOR?
Up until now, implementations of Koskeniemmi's two-level model have been available only on large computers housed at academic or industrial research centers. As an implementation of the two-level model, PC-KIMMO is important because it makes the two-level processor available to individuals using personal computers. Computational linguists can use PC-KIMMO to investigate for themselves the properties of the two-level processor. Theoretical linguists can explore the implications of two-level phonology, while descriptive linguists can use PC-KIMMO as a field tool for developing and testing their phonological and morphological descriptions. Teachers of courses on computational linguistics can use PC-KIMMO to demonstrate the two-level approach to morphological parsing. Finally, because the source code for the PC-KIMMO's generator and recognizer functions is made available, those developing natural language processing language processing applications (such as a syntactic parser) can use PC-KIMMO as a morphological front end to their own programs.
(Note: as of May 1991, a program called KTEXT is available. It uses the PC-KIMMO parser and processes text, producing a morphological parse of each word in the text. See below for more information.)

Warm-up con PCKimmo

1. Scaricare il programma e scompattarlo in una cartella
2. Scaricare il file test.rul dal sito del CISCL: http://www.ciscl.unisi.it/lab/ e dare un'occhiata al file test.txt:

```
ALPHABET
;lexical (upper) and surface (lower) characters:
b c d f g h j k l m n p q r s t v w x y z a e i o u ' - .
sh ch  ;digraphs
B C D F G H J K L M N P Q R S T V W X Y Z A E I O U 1 2 +PL
 ;lexical (upper) only characters:
` +
NULL     0
ANY      @
BOUNDARY #
SUBSET CN     b c d f g h j k l m n p q r s t v w x y z sh ch
SUBSET CNsib  s x z sh ch              ;sibilant consonants
...

RULE
"Defaults 1" 1 33
   b c d f g h j k l m n p q r s t v w x y z sh ch a e i o u ' - ` + @
   b c d f g h j k l m n p q r s t v w x y z sh ch a e i o u ' - 0 0 @
1: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

;=======
;Plurali semplici
;=======

; casa+PL > case

RULE
"[ao]+PL > ei" 3 4
    a  o +PL @
    e  i   0   @
 1:  2  2  0   1
 2.  0  0  3   0
 3:  3  3  3   3

END
```

3. aprire "pckimmo.exe" e digitare "load rule test.txt"
4. digitare "generate casa+PL"

Riferimenti

CHILDES si può scaricare gratuitamente al seguente indirizzo web (software CLAN, dati e documentazione):
http://childes.psy.cmu.edu/

PHPChildes (l'interfaccia web di Childes arricchita con Epressioni Regolari) è consultabile all'indirizzo:
http://www.ciscl.unisi.it/childes/
login: guest
pass: unisi

PC-Kimmo (software e documentazione):
http://www.sil.org/pckimmo/