MARCO TOMATIS
(Torino)

9. La disambiguazione del Corpus Taurinense. *Problemi teorici e pratici.*

0. INTRODUZIONE. Una delle prerogative distintive del Corpus Taurinense consiste nell'aver subito un consistente processo di disambiguazione come passo successivo al *POS-tagging* (come già accennato in Barbera ¶ 1, *supra*, § 2.2.1). Tale processo si è reso indispensabile al fine di garantire l'assegnazione univoca delle varie classi grammaticali ai diversi token costituenti l'intero testo. Nel seguente articolo si illustreranno, in modo preliminare, i problemi teorici e tecnici con cui è stato necessario confrontarsi per la disambiguazione del Corpus Taurinense e, più nel dettaglio, le procedure e le soluzioni computazionali successivamente adottate.

0.1 SISTEMI DI DISAMBIGUAZIONE: UNA PANORAMICA GENERALE. Per una trattazione il più possibile chiara del tema in questione, si rivela indispensabile fornire una descrizione di massima dei vari significati che il termine "disambiguazione" può veicolare all'interno del paradigma della *corpus linguistics*. È bene precisare, tuttavia, che sebbene tale termine esprima chiaramente il concetto di eliminazione o riduzione del grado di ambiguità posseduto da un determinato elemento presente all'interno di un sistema complesso, quale quello lessicale, la trattazione prenderà in esame unicamente il livello di analisi di natura testuale, tralasciando le questioni relative alla gestione di informazioni di tipo sonoro. Fatta questa premessa, è bene chiarire subito che con "disambiguazione" è possibile fare riferimento a due generi di problemi differenti e ben distinti tra loro. La disambiguazione di un dato elemento testuale, infatti, può riferirsi sia alla definizione univoca delle caratteristiche semantiche che tale elemento possiede, naturalmente in stretta relazione con il contesto in cui si trova inserito, sia alla definizione univoca delle sue caratteristiche in termini di categoria grammaticale di appartenenza (indicata anche con POS, ossia *part of speech*). Sebbene l'elaborazione computazionale della semantica dei vari token costituenti il testo sia un settore di ricerca molto complesso e in piena evoluzione, che richiede l'uso di strumenti appropriati quali ontologie e reti semantiche, è utile precisare che per quanto concerne il processo di disambiguazione lessicale progettato per il Corpus Taurinense, tale elaborazione si limita alla seconda delle accezioni di cui sopra, ossia al livello delle categorie morfosintattiche. A tale proposito bisogna ricordare che a differenza della disambiguazione testuale di natura semantica, obbligatoriamente vincolata all'analisi del contesto specifico, il processo di disambiguazione morfosintattico può essere elaborato secondo modelli computazionali sia di tipo *context sensitive* (sensibili al contesto), sia *context free* (svincolati dal contesto). Al riguardo è utile precisare che questi ultimi, essendo unicamente legati alla natura morfosintattica dei token circostanti, ma non alla loro forma lessicale, risultano intrinsecamente dotati di maggiore potenza e flessibilità rispetto ai primi.

In merito agli strumenti necessari per ottenere la disambiguazione lessicale a livello di categoria grammaticale, l'operatore può scegliere di optare per due diverse soluzioni alternative. La prima di esse, utilizzata nella maggior parte dei casi, prevede l'adozione di sistemi stocastici basati su Modelli Markoviani Nascosti (HMM, *Hidden Markov Models*), strumenti molto potenti e versatili che coniugano in una sola fase il processo di assegnazione delle etichette morfosintattiche e la conseguente disambiguazione. Nonostante le già citate doti di robustezza e flessibilità, è importante tuttavia sottolineare la necessità di tali sistemi di avvalersi di un corpus di dimensioni ridotte, precedentemente annotato, da cui poter trarre le informazioni utili per svolgere l'elaborazione statistica. Tale procedura, denominata *training* (ossia allenamento) del tagger,

Corpora e linguistica in rete, a cura di Manuel Barbera, Elisa Corino e Cristina Onesti, Perugia, Guerra Edizioni, 2007, pp.169-181.

risulta a tutti gli effetti indispensabile, tanto da risultare un elemento fondamentale per la valutazione delle prestazioni del sistema.

Per contro, la seconda delle soluzioni possibili prevede di limitare la fase di etichettatura alla semplice assegnazione delle varie POS a tutti i token che compongono il testo, proseguendo successivamente l'elaborazione con la fase di disambiguazione vera e propria, solitamente costituita da un motore basato su regole linguistiche di tipo *context free* o, meno preferibilmente, *context sensitive*. Naturalmente, a differenza dei sistemi basati su modelli statistici, discussi in precedenza, nel caso in questione sarà necessario l'utilizzo di due processi distinti e nettamente separati anche sotto il profilo dell'elaborazione computazionale.

Per quanto riguarda un bilancio sommario dei pregi e dei difetti di entrambi i sistemi, risulta evidente che il processo stocastico, dato il ridotto numero di fasi di elaborazione, risulti caratterizzato da notevoli vantaggi in termini di velocità di esecuzione e leggerezza computazionale. A tali caratteristiche si somma l'interessante capacità di poter assegnare POS univoche anche laddove la parola specifica risulti totalmente ignota al lessico di riferimento del sistema (es. neologismi). Per contro, la necessità di disporre di un *training corpus* preventivo ne riduce drasticamente la capacità di utilizzo al di fuori dei canoni linguistici già noti e consolidati. I modelli a regole, per contro, pur essendo svincolati dalla necessità di disporre di un corpus già annotato, con il conseguente vantaggio di poter essere applicati anche su corpora di lingue mai precedentemente etichettate, risultano di fatto assai più pesanti in termini di elaborazione computazionale. Inoltre, la necessità di un lungo lavoro di sviluppo di regole linguistiche sulla base di un formalismo ben preciso e definito, rende il sistema maggiormente costoso, nonché molto meno agevole da gestire e mantenere.

Per quanto riguarda il CT, risulta evidente che, nonostante i difetti emersi, volendo trattare una lingua computazionalmente vergine come l'italiano del '200, l'unica soluzione possibile fosse lo sviluppo di un sistema articolato di regole capace di coprire l'intera gamma di possibili varianti e anomalie linguistiche presenti all'interno del corpus.

0.2 PREMESSE METODOLOGICHE. Lo sviluppo di un sistema di disambiguazione contestuale del *Corpus Taurinense* si è presentato fin dai primi momenti come un'opera di non banale complessità. Di diversa natura, infatti, sono i problemi che deve affrontare la persona che si accinge a compiere tale opera: il primo, e più evidente, consiste nella natura del corpus stesso. Trattandosi di una lingua antica, infatti, è necessario l'ausilio di una persona dotata di un buon bagaglio filologico al fine di ottenere una corretta interpretazione del testo e, conseguentemente, una corretta gestione delle diverse problematiche linguistiche che possono presentarsi durante lo svolgimento del lavoro. Il secondo tipo di difficoltà, di natura più eminentemente pratica, consiste nella necessità di scegliere un formalismo od un linguaggio di programmazione che risulti il più adeguato possibile allo scopo che si vuole portare a termine, senza tuttavia introdurre un eccessivo livello di complessità computazionale o difficoltà realizzativa, elementi questi che potrebbero distogliere energie al più importante problema dell'effettiva formulazione della grammatica di disambiguazione.

Se il problema di natura filologica ha visto una soluzione piuttosto agevole grazie al prezioso contributo di Manuel Barbera, la decisione in merito alla tipologia del sistema computazionale da adottare ha richiesto uno sforzo valutativo più intenso. L'elaborazione elettronica delle lingue naturali (NLP - *Natural Language Processing*) dispone di numerosi strumenti informatici, perlopiù linguaggi di programmazione, caratterizzati da peculiarità che permettono di conseguire i risultati desiderati nella maniera più agevole possibile. Pertanto, se un efficiente sistema di analisi morfologica può essere realizzato mediante un automa a stati finiti non deterministici sviluppato in un linguaggio multipiattaforma quale il Java, un sistema di analisi sintattica (*parser*) può essere altrettanto agevolmente prodotto mediante l'uso di un linguaggio dichiarativo basato sulla logica matematica quale il Prolog (termine composto dalla sigla "*Programming in*

Logic"). In seguito a numerose valutazioni tecnico-pragmatiche si è deciso di implementare la grammatica di disambiguazione in una struttura di programma basata sul linguaggio AWK. Tale scelta, forse criticabile per alcuni aspetti di natura più marcatamente informatica relativi a valutazioni di velocità ed efficienza computazionale, ha avuto tuttavia il merito di fornire al sistema di regole (che ricordiamo essere in maggior numero di tipo *context sensitive*, ossia strettamente legate al contesto in cui operano) una struttura estremamente flessibile, leggera, versatile e facilmente adattabile ad ulteriori aggiunte o modifiche.

1. ARCHITETTURA DEL SISTEMA DI DISAMBIGUAZIONE. Data la natura tipicamente procedurale del linguaggio adottato, il sistema di disambiguazione possiede una struttura generale costituita da una serie di moduli indipendenti, operanti secondo una ben precisa gerarchia sequenziale. Attualmente il sistema si compone di sei moduli di disambiguazione e due moduli di formattazione del testo, la cui funzione verrà discussa più avanti. Poiché soltanto il primo dei sei moduli opera su una copia opportunamente formattata del testo etichettato originale, mentre ogni modulo successivo agisce sul testo generato dall'elaborazione del modulo precedente, ecco che l'organizzazione del sistema in una ben precisa gerarchia d'intervento si rivela una soluzione indispensabile. Tale configurazione, infatti, consente di frazionare e distribuire le operazioni di disambiguazione in vari livelli distinti, secondo una disposizione gerarchica che è funzione della rilevanza linguistica e computazionale delle varie POS[1] (*part of speech*) da trattare. Non risulta casuale, quindi, che il modulo iniziale sia composto unicamente dalle regole atte a trattare le forme caratterizzate da ambiguità nome / verbo (es. "fatto"), mentre il successivo comprenda le forme nome / aggettivo non disambiguabili da regole generali.

La struttura interna dei singoli moduli risulta piuttosto semplice: ogni modulo è costituito da una serie di regole a mutua esclusione che agiscono sul testo etichettato come una sorta di *filtro passivo*. L'intero processo di disambiguazione, infatti, si limita ad eliminare le voci di transcategorizzazione non pertinenti semplicemente assegnando, previa selezione, l'elemento morfosintattico più corretto all'interno di ogni token caratterizzato da ambiguità.

Esistono due diversi tipi di ambiguità che il sistema qui descritto è in grado di riconoscere e correggere. Definiti rispettivamente con i termini di *ambiguità interna* ed *ambiguità esterna*, i due generi di ambiguità si differenziano sostanzialmente in base alle loro caratteristiche intrinseche: l'ambiguità interna comprende le ambiguità di MSF (genere, numero, persona, ecc.), mentre quella esterna rappresenta l'intera serie di POS assegnate, in fase di *tagging*, a una data forma. Risulta pertanto evidente la possibilità di coesistenza, all'interno delle forme etichettate, di entrambe le ambiguità.

1.1 CARATTERISTICHE SALIENTI DEL LINGUAGGIO DI *SCRIPTING* ADOTTATO**.** Come già accennato in precedenza, AWK è un linguaggio di natura procedurale. Tuttavia le sue caratteristiche interne di funzionamento fanno sì che esso sia uno dei sistemi più semplici, ma nel contempo più efficienti, per la manipolazione di testi. AWK, infatti, dispone di potenti funzioni predefinite quali ad es. la possibilità di realizzare *pattern matching* mediante l'uso di espressioni regolari o la capacità di segmentare un testo intero dividendolo in righe e in campi contenenti i singoli token appartenenti alla riga stessa.

Tuttavia, nel nostro caso, data la natura estremamente *context-sensitive* delle regole di disambiguazione, si è rivelato indispensabile poter operare sul testo con un elevato grado di elasticità. A tal fine, quindi, si è optato per la soppressione della segmentazione automatica del testo in righe successive, in modo da gestire l'intero documento come se fosse costituito da una singola riga intera.

[1] Nel prosieguo non saranno commentate le varie "labels" del tagset del CT, per quale basta rimandare al contributo precedente in questo volume, Barbera ¶ 8.

Il modulo "dis_prep":	**Il modulo "dis_end":**

`# Source formatting module`
`#`
`{`
`gsub (/^ /, "")`
`print $0 "¥"`
`}`
	`# Format restoring module`
	`#`
	`{`
	`rc = 1`
	`gsub (/\¥ /, "¥")`
	`rec = split ($nf, sp, "¥")`
	`while (rc <= rec)`
	`{`
	`print " " sp[rc]`
	`rc++`
	`}`
	`}`

Tav. 1ab: I moduli "dis_prep" e "dis_end".

Questa soluzione, totalmente priva di svantaggi, ha permesso la creazione di tre puntatori, definiti all'interno del programma dalle variabili "campo", "bw" e "fw". Il primo di essi, "campo", costituisce l'elemento centrale di tutto il sistema di disambiguazione, poiché è preposto alla scansione sequenziale di tutti i token presenti nel testo. Gli altri due puntatori, invece, pur ricoprendo un ruolo importante, possono essere considerati elementi ausiliari in quanto, essendo progettati per esaminare il contenuto del campo immediatamente precedente e immediatamente successivo a quello oggetto di analisi, permettono al linguista di formulare regole contestuali dotate di un notevole grado di precisione. Inoltre l'elevata flessibilità dell'impostazione qui adottata consente, quando necessario, di estendere l'indagine contestuale a una zona di testo anche considerevolmente più ampia rispetto a quella di *default* appena descritta mediante la definizione, all'interno delle regole stesse, di ulteriori puntatori ausiliari. Tuttavia, poiché questa semplice struttura non permette il ripristino della formattazione originale delle righe di testo al termine dell'elaborazione, si è visto necessario affiancare ai 6 moduli costituenti il motore di disambiguazione, due moduli appositamente creati per la gestione dell'aspetto grafico del testo. Il primo di tali moduli, chiamato "dis_prep", cura l'inserimento di un carattere speciale ("¥", scelto arbitrariamente) al termine di ogni linea del testo etichettato originale. Detto carattere funge da marcatore di fine riga, consentendo al secondo modulo di formattazione "dis_end" la fedele ricostruzione del formato grafico originario.

1.2 OTTIMIZZAZIONE DEL SISTEMA. *Last but not least*, allo scopo di restringere l'indagine del disambiguatore unicamente agli elementi testuali considerati linguisticamente rilevanti, si è provveduto al riconoscimento, da parte del sistema, di tutti i codici di markup presenti all'interno delle frasi. Tali codici, del tutto privi di contenuto linguistico, verranno automaticamente saltati dai menzionati puntatori in fase di analisi. Quest'ultimo accorgimento, semplice ma estremamente utile, fa sì che il sistema operi su un testo che può essere considerato a tutti gli effetti 'virtuale' in quanto, ad esclusione dei codici strettamente legati al *tagging* delle varie forme, risulta virtualmente privo di tutte quelle stringhe di caratteri aggiuntive non presenti sul testo cartaceo originale. Può essere ora utile fornire una brevissima analisi delle tecniche di programmazione adottate nello sviluppo del sistema.

Come già accennato in precedenza, l'organizzazione interna dei singoli moduli che formano il disambiguatore è costituita da una serie di regole linguistiche a mutua esclusione. Tuttavia, al fine di ottimizzare al massimo la struttura informatica di tale sistema, si è deciso di sfruttare la caratteristica di AWK che consente la gestione di funzioni definite dall'utente. Una funzione consiste in una parte di codice di programmazione che può essere richiamato, all'interno del programma, da un comando corrispondente al nome della funzione stessa. Al fine di poter sta-

bilire un legame comunicativo tra il corpo del programma e la funzione è necessario che, unitamente al comando di attivazione, vengano forniti una serie di valori denominati "parametri". La scelta di tali parametri, definita in fase di progettazione, è unicamente vincolata al particolare tipo di elaborazione per cui la funzione è stata predisposta.

L'architettura qui descritta, che, è bene sottolineare, non incide in alcuna misura sui livelli di rendimento computazionale del sistema, offre numerosi vantaggi. Innanzitutto fornisce alle regole linguistiche una maggiore chiarezza espositiva: le regole, essendo meno circondate da linee di programma, potranno essere più facilmente gestibili e modificabili dal personale incaricato anche numerosi anni dopo la conclusione del progetto. Altri vantaggi si riflettono a livello di riduzione delle dimensioni complessive del sistema e di maggiore facilità nella manutenzione della struttura del software.

2 DESCRIZIONE ANALITICA DEGLI ELEMENTI STRUTTURALI COSTITUENTI I VARI MODULI. Per una migliore comprensione di quanto presentato nei paragrafi precedenti, viene ora fornita una descrizione dettagliata dei blocchi funzionali che si possono incontrare all'interno dei vari moduli. È utile precisare che a parte le funzioni definite dall'utente, tutto ciò che, a livello generale, verrà descritto nel presente paragrafo dovrà necessariamente apparire in ogni modulo. Per quanto riguarda il caso specifico delle funzioni, invece, poiché la scelta della specifica funzione da implementare dipende unicamente dalla complessità computazionale di ciascun modulo, vi saranno moduli in cui potranno coesistere ben quattro funzioni definite dall'utente e moduli in cui una sola funzione risulterà sufficiente per il corretto funzionamento del sistema.

2.1 LINEE DI COMMENTO. Ogni modulo può iniziare con una o più linee di commento in cui vengono indicati il nome del modulo e il tipo di regole ivi ospitate. Tali linee sono immediatamente riconoscibili in AWK in quanto precedute dal simbolo "#"

```
# Motore di disambiguazione - Versione 2.0
#
# Modulo 4:
#       Disambiguazione di:
#
#       - preposizioni, verbi, congiunzioni, ecc.
#
```

Tav. 2: Le linee di commento.

2.2 INIZIO DEL PROGRAMMA. Terminate le righe di commento iniziali, la parte di programma vero e proprio incomincia con una 'regola' di programma chiamata "BEGIN". È necessario puntualizzare che il termine 'regola' appena usato non denota una regola linguistica di disambiguazione, bensì una ben precisa procedura inerente al linguaggio di programmazione stesso. AWK richiede che, a parte i comandi "BEGIN", "END" e le funzioni definite dall'utente, tutte le "regole" che costituiscono un programma siano incluse tra parentesi graffe.

```
BEGIN {
RS = ""
# gestisce l'input come se fosse formato da una riga unica
ORS = " "
# inserisce uno spazio alla fine di ogni 'print'
nf = 1
}
```

Tav. 3: L'inizio del programma

Il comando “BEGIN” viene usato con lo scopo di far eseguire una serie di passi di programma una sola volta all’inizio dell’elaborazione. Nello specifico, in fase di progettazione si è deciso di utilizzare tale comando al fine di definire preventivamente il valore di alcune variabili che verranno usate successivamente all’interno del corpo del programma. In AWK vi sono fondamentalmente due tipi di variabili: le variabili di sistema e le variabili generiche. Le prime, denotate da sigle contenenti solo lettere maiuscole, hanno il potere di modificare impostazioni predefinite o svolgere funzioni particolari; le seconde, invece, definite in genere da lettere minuscole, rappresentano le variabili classiche presenti in ogni linguaggio di programmazione e vengono utilizzate con lo scopo di immagazzinare valori (di tipo numerico o stringa) che possono essere modificati a piacere a seconda delle esigenze. Nel nostro caso specifico, il comando “BEGIN” ci consente di impostare il valore delle variabili di sistema che si occupano della segmentazione del testo in righe. Le variabili in questione, denotate dalle sigle “RS” (*record separator*) e “ORS” (*output record separator*), possono essere programmate al fine di modificare il comportamento standard di AWK così da adattarlo agli scopi dell’utente. Di norma AWK agisce segmentando il testo d’ingresso in righe basandosi sul carattere di fine riga, non visibile, “\n”. In fase di scrittura, invece, il linguaggio inserisce un carattere di fine riga al termine di ogni parte di testo stampata mediante il comando “print”. In accordo con quanto già affermato nel § 1.1, la configurazione appena descritta non risulta adeguata agli scopi del nostro progetto, pertanto si rende necessaria una sostanziale modifica di tale comportamento. Poiché AWK consente di definire, mediante le variabili citate in precedenza, il carattere che l’utente desidera riservare alle funzioni di separatore di riga del testo d’ingresso e separatore di riga in fase di stampa, assegnando alla variabile “RS” un carattere nullo (“”) e ad “ORS” un carattere di spazio (“ ”), si è consentito al disambiguatore di gestire l’intero testo etichettato come composto da una sola riga e di produrre un testo di uscita costituito anch’esso da una sola riga in cui le diverse parti frutto di stampa risultino separate tra loro da uno spazio.

Oltre alle variabili preposte alla gestione della segmentazione delle righe, AWK possiede altre due variabili, “FS” (*field separator*) e “OFS” (*output field separator*). Dette variabili, aventi caratteristiche operative del tutto simili alle precedenti, risultano però responsabili della gestione dei campi. Nel funzionamento di base, i campi contenuti in ogni riga di testo vengono separati tenendo conto della spaziatura. Pertanto, sebbene sia di agevole modifica, questa comportamento viene lasciato del tutto inalterato all’interno dei vari moduli di disambiguazione.

In ultima istanza, nella riga conclusiva del blocco di programma facente capo alla funzione “BEGIN” è stata definita la variabile “nf”, caricata con il valore intero “1”. L’utilizzo di quest’ultima variabile, che descriveremo nel paragrafo successivo, è di importanza fondamentale per il funzionamento stesso del sistema.

2.3 CORPO DEL PROGRAMMA. Le righe iniziali del corpo del programma sono tra le più importanti:

In esse, infatti, si trova la definizione dei tre puntatori cui si fa riferimento nel par. 1.1, l’impostazione delle regole di eliminazione virtuale dei codici testuali non pertinenti (cfr. § 1.2) ed infine il motore di disambiguazione vero e proprio, costituito da regole linguistiche e funzioni definite dall’utente (cfr. § 3 e sg.).

Il funzionamento dell’intero sistema di disambiguazione da noi proposto ruota intorno a un nucleo centrale costituito dalla riga:

```
while (nf <= NF)
```

Nonostante la sua apparente semplicità, tale riga riveste un’importanza fondamentale in quanto è proprio per mezzo di essa che il disambiguatore può procedere al lavoro di scansione all’interno del testo dei vari token ambigui. È doveroso, a questo punto, fornire una descrizione dettagliata di questa linea di codice e del suo funzionamento.

```
{
while (nf <= NF)
    {
#!*!* Inizio regole di disambiguazione *!*!
#
# Creazione di 3 puntatori:
# 'nf' -> punta al campo corrente
# 'bw' -> punta al campo che precede 'nf' di N posizioni
# 'fw' -> punta al campo che segue 'nf' di N posizioni
#
    campo = $nf
    fw = nf
    fw++
    if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\Ÿ/ || $fw ~ /\#/)
        fw++
    if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\Ÿ/ || $fw ~ /\#/)
        fw++
    if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\Ÿ/ || $fw ~ /\#/)
        fw++
# omette le stringhe contenenti:
# '@'
# '%'
# '$'
# 'Ÿ'
# '#'
    bw = nf
    if (nf >=2)
        bw--
if (($bw ~ /\@/ || $bw ~ /\%/ || $bw ~ /\$/ || $bw ~ /\Ÿ/ || $bw ~ /\#/) && bw >
                2)
        bw--
if (($bw ~ /\@/ || $bw ~ /\%/ || $bw ~ /\$/ || $bw ~ /\Ÿ/ || $bw ~ /\#/) && bw >
                2)
        bw--
if (($bw ~ /\@/ || $bw ~ /\%/ || $bw ~ /\$/ || $bw ~ /\Ÿ/ || $bw ~ /\#/) && bw >
                2)
        bw--
# omette le stringhe contenenti:
# '@'
# '%'
# '$'
# 'Ÿ'
# '#'
#
```

Tav. 3: Corpo del programma

Iniziamo con l'analisi del comando "while". Questo comando indica al sistema di ripetere un certo tipo di istruzione, o gruppo di istruzioni, finché la condizione espressa all'interno della parentesi tonda continui a risultare vera. Il ciclo si chiude ed il programma continua il proprio flusso normale solo nel momento in cui la condizione dovesse restituire un risultato negativo, ossia di non verità. Nel nostro caso, quindi, il gruppo di istruzioni incluse nel ciclo "while" verranno ripetute tante volte finché la variabile "nf" non contenga un valore numerico maggiore di "NF". È evidente, quindi, come la procedura di aggiornamento di "nf" ricopra un ruolo delicato: se non ben realizzata, può presentarsi il rischio di un ingresso in *loop* dell'esecuzione del programma (caratterizzato dalla ripetizione all'infinito dello stesso comando) o, in alternativa, possono risultare alcune perdite di dati nel testo di uscita. Per ovviare a tali rischi, pertanto, il valore contenuto in "nf" viene aggiornato dal programma immediatamente dopo l'analisi di ciascun elemento testuale. Se riguardo a "nf" non vi è molto da aggiungere a quanto già detto fino-

ra, la variabile "NF" richiede invece un commento più articolato. Come già accennato in precedenza, il linguaggio di programmazione da noi adottato utilizza al suo interno una serie di variabili di sistema dedicate allo svolgimento di compiti ben precisi. La variabile "NF" (*Number of Field*) è anch'essa una variabile di sistema che però, a differenza di quelle già incontrate, fornisce il conteggio della quantità di campi presenti all'interno del testo. Poiché nel nostro sistema i campi vengono divisi tenendo conto del carattere di spazio, "NF" fornirà il valore corrispondente alla quantità di token presenti nel testo da analizzare.

Date tali premesse, diventa più agevole comprendere la riga di programma presentata: finché la variabile generica "nf", inizialmente caricata con il valore numerico '1', conterrà un valore minore od uguale al numero totale dei campi contenuto in "NF", il sistema procederà all'esecuzione delle varie regole di disambiguazione presenti all'interno del ciclo "while". La scansione del testo si interromperà, invece, solo nel momento in cui "nf" conterrà un valore maggiore di "NF", segno che anche l'analisi dell'ultimo token ha trovato compimento.

In AWK, come in altri linguaggi quali il C, C++, Java, Perl, ecc. è necessario l'uso delle parentesi graffe per includere quelle parti di programma che risultano gerarchicamente dipendenti da altre. Pertanto il corpo delle regole di disambiguazione, dipendendo direttamente dal precedente comando "while", dovrà essere preceduto da una parentesi graffa aperta.

Proseguendo con la descrizione analitica del programma, ci accingiamo ora ad esaminare nel dettaglio la definizione dei puntatori "bw", "fw" e "campo". I tre puntatori qui elencati si trovano all'interno del gruppo di istruzioni che, gerarchicamente dominate dal "while" di cui sopra, costituiscono il sistema di disambiguazione vero e proprio. Il puntatore "campo", infatti, è una variabile definita dalla riga:

```
campo = $nf
```

Tale linea di programma fa sì che all'interno di "campo" venga caricata la stringa di caratteri appartenente al campo indicato dal valore di "nf". Il simbolo "$" che precede "nf" indica appunto che "campo" conterrà un valore di tipo stringa e non di tipo numerico.

Gli altri due puntatori, invece, partendo sempre dal valore di "nf", consentono di leggere il contenuto del testo presente nei campi immediatamente precedenti ed immediatamente successivi a "campo". Tuttavia in questo contesto si inserisce anche il sistema di controllo automatico dei codici di markup, elementi testuali totalmente privi di rilevanza in seno al processo di disambiguazione. Tale sistema automatico prevede l'incremento del valore contenuto nella variabile "nf" ed "fw" ed il decremento di "bw" ogniqualvolta il sistema incontri un campo in cui siano presenti i simboli: "@", "%", "$", "*f*" e "#". Come già accennato in precedenza, questo accorgimento consente di elaborare regole di disambiguazione che agiscono su materiale puramente testuale, senza dover tenere conto di tutti gli elementi di natura extralinguistica presenti nel testo etichettato. Per maggiore completezza descrittiva è bene precisare che solo "nf", in quanto variabile centrale, subirà un incremento pari ad uno. Le altre due variabili "fw" e "bw", invece, in virtù della loro funzione ausiliaria, potranno subire variazioni differenti, in stretta relazione con il numero di codici di markup che è necessario saltare prima di incontrare un elemento di testo valido. Poiché il testo può presentare i suddetti codici in posizione consecutiva fino a un massimo di quattro, si è predisposto un sistema di controllo per evitare che "bw" possa assumere valori negativi, rischio presente soprattutto nei momenti iniziali dell'elaborazione.

3 REGOLE DI DISAMBIGUAZIONE. Non potendo, per ovvie ragioni di spazio, presentare un'analisi completa di ciascuna delle regole linguistiche implementate nel sistema, ci limiteremo ad un excursus parziale prendendo in esame alcune delle regole più significative presenti nei vari moduli. Prima di addentrarci nell'argomento, però, è opportuno precisare che, poiché all'interno di un modulo le varie regole sono organizzate in un sistema sequenziale a mutua esclusione, queste dovranno essere disposte tenendo conto del loro livello di generalizzazione.

Una regola che agisce prendendo in esame i valori di HDF ed MSF sarà dotata di una capacità di disambiguazione nettamente più ampia e generale rispetto ad una regola che basa la sua capacità di azione unicamente sull'analisi del lemma o della forma di un dato token. Date queste premesse, risulta chiaro che la presenza in uno stesso modulo di due regole differenti che trattano una problematica comune (es. le forme straniere), richiederà uno studio accurato sulla loro dislocazione all'interno del modulo stesso, al fine di evitare che l'entrata in funzione di una determinata regola *ad hoc* (ossia *context sensitive*) venga impedita dalla compresenza di una regola generale di tipo *context free*.

Una norma che consente di ottenere una certa sicurezza organizzativa consiste nel disporre le regole dotate di maggiore generalizzazione in una posizione più avanzata rispetto a quelle legate al contesto specifico, che saranno pertanto le prime ad entrare in azione. Questo aspetto, che incide in primo luogo sull'organizzazione interna, si riflette anche a livello esterno sulla disposizione sequenziale dei moduli: quelli caratterizzati dal possedere regole generali, infatti, entreranno in funzione solo in un momento successivo rispetto ai moduli costituiti da regole sensibili al contesto. Tuttavia, è bene precisare che la scelta del tipo di regole da inserire all'interno dei vari moduli è anche strettamente legato alla capacità di analisi che si intende attribuire ai moduli stessi. Se si prende in esame, in qualità di esempio, il sistema di regole adottato per il trattamento degli articoli determinativi transcategorizzanti con pronomi, è possibile notare che, a differenza di quanto detto poc'anzi, le regole di portata generale sono presenti in un modulo antecedente a quello che contiene le regole che agiscono ad un livello più specifico. Questo tipo di scelta, apparentemente in contrasto con i principi base di ortodossia organizzativa, trova la sua giustificazione nel fatto che i risultati di questa specifica azione di disambiguazione, che richiede un sistema di analisi piuttosto complesso ed articolato, possano essere immediatamente utilizzati da altre regole presenti nei moduli immediatamente successivi. Mediante tale disposizione, infatti, la disambiguazione avviene in due moduli ed in due momenti ben precisi e distinti: il primo gruppo di regole, infatti, agisce nel terzo modulo di programma e si comporta come un filtro a maglia larga, occupandosi quasi unicamente di discriminare gli articoli determinativi dalle corrispettive forme pronominali. Il secondo gruppo, invece, che agisce nel quarto modulo, si occupa più nello specifico di assegnare loro i corretti valori di lemma. Poiché numerose regole richiedono la disambiguazione dell'articolo o del pronome per poter portare a termine il proprio compito, appare evidente come l'importanza di una discriminazione, seppur grossolana, della POS sia nettamente prioritaria rispetto al compito di assegnazione del lemma corretto; da qui la scelta, quasi obbligata, di una organizzazione delle regole in una maniera che può apparire, a prima vista, alquanto irrazionale. In conclusione, ritornando al discorso riguardante l'importante aspetto dell'organizzazione interna del sistema di regole, possiamo comunque ragionevolmente affermare che è sempre consigliabile optare, ogniqualvolta si presenti la possibilità, verso l'accorpamento, nei diversi moduli, delle regole con caratteristiche comuni, in modo da evitare il più possibile la promiscuità tra tipi di regole caratterizzate da capacità di analisi differente.

3.1 ESEMPIO DI REGOLA TRATTA DA "MODULO 1". Formato unicamente da regole di tipo *context sentitive*, il modulo 1 è interamente dedicato al trattamento dei casi di ambiguità verbale interna e/o esterna non risolvibili mediante regole generali.

In tavola 4 se ne fornisce un esempio (in corpo ridotto per economia di spazio), che sarà poi partitamente analizzato.

A	# Regola per la disambiguazione interna
	# ed esterna della forma 'ave'
	else
	if (campo ~ /^ave_/ && campo ~ /\);\(/)
	{
		E	else
			if ($fw ~ /^÷gli_/)
			{
			sub (/;3/, "", campo)
			sub (/6;/, "", campo)
			assegna(campo,"211",end)
			}

B	if (campo ~ /¥$/)
	end = "¥"
	else
	end = ""
		F	else
			{
			sub (/2;/, "", campo)
			sub (/;7/, "", campo)
			assegna(campo,"211",end)
			}
			}

C	nf++		

D	if ($fw ~ /^÷lle_/)
	{
	assegna(campo, "221", end)
	}
	else
	if ($fw ~ /^mari[ae]_/)
	{
	assegna(campo, "68", end)
	}		

Tav. 4a-f: Una regola di disambiguazione del modulo 1

A è l'elemento di controllo che si occupa di verificare la possibilità dell'entrata in funzione della regola mediante l'esecuzione di un confronto (*pattern matching*) tra il valore di stringa contenuto nella variabile "campo" e lo specifico token che la regola intende trattare. La richiesta di un'operazione di confronto tra modelli di stringhe viene inoltrata al linguaggio AWK mediante l'uso del simbolo speciale "~".

B introduce ulteriori elementi di controllo finalizzati alla corretta gestione del marcatore di fine riga (cfr. § 1.1).

C è la linea riservata all'incremento della variabile "nf" che scansiona il testo (cfr. § 1.1).

D è la porzione di regola che rappresenta l'aspetto *context sensitive* del disambiguatore: utilizzando il confronto tra la stringa contenuta nel campo successivo e quella necessaria per poter assegnare un determinato valore di POS, la regola comanda al sistema di eseguire l'operazione di eliminazione dell'ambiguità esterna. Tale ordine viene impartito ricorrendo alla funzione "assegna", alla quale devono essere comunicati i parametri necessari per lo svolgimento del lavoro di disambiguazione vero e proprio (cfr. § 4).

E, poi, è la parte di regola che, oltre alla funzione descritta nel punto precedente, comprende anche la gestione dell'ambiguità interna. Questa viene eliminata ricorrendo al comando "sub" (*substitution*), funzione che consente di modificare un determinato valore alfanumerico all'interno di una variabile stringa. In dettaglio, la disambiguazione interna viene ottenuta sostituendo all'interno di "campo" il valore di MSF non desiderato con un carattere nullo.

F, infine, è il finale della regola, costituito in questo specifico caso unicamente da comandi per la disambiguazione interna, indica al sistema il comportamento a cui attenersi nel caso in cui i precedenti controlli sui campi circostanti dovessero dare esito negativo. Il finale di regola qui descritto è importante poiché consente di evitare la formulazione di regole specifiche necessarie a coprire tutta l'ampia casistica di variazione del contesto, pertanto è presente in quasi tutte le regole appartenenti ai vari moduli.

4 FUNZIONI DEFINITE DALL'UTENTE.

Riguardo a questo argomento il manuale di AWK afferma che «Definitions of functions can appear anywhere between the rules of an 'awk' program», ossia le funzioni definite dall'utente possono trovarsi ovunque tra le regole di programma. Questa caratteristica, che volendo consente al programmatore di inserire le funzioni anche

al fondo dell'intero listato di codice, è data dal fatto che questo linguaggio di programmazione esamina preventivamente l'intero programma prima di procedere all'esecuzione. Pertanto noi tratteremo il presente argomento come una sorta di entità autonoma e separata rispetto al corpo del programma vero e proprio.

In AWK una funzione si dichiara usando il comando "function" seguito dal nome della funzione stessa. Esso è a sua volta seguito da una parentesi tonda contenente i parametri (cfr. § 3.1) e le variabili che operano all'interno della funzione. Le varie funzioni del nostro programma sono caratterizzate dall'avere un numero di parametri costante, ma un numero di variabili differente. Occorre infine precisare che il carattere di spazio che separa i due blocchi di elementi all'interno della parentesi tonda è totalmente privo di qualsiasi utilità computazionale: il suo utilizzo viene consigliato unicamente per favorire la leggibilità del programma.

Le funzioni definite dall'utente costituiscono, nel nostro sistema, il motore vero e proprio del sistema di disambiguazione. È al loro interno, infatti, che avviene il processo di selezione ed assegnazione della categoria grammaticale corretta e l'eliminazione di tutte le altre transcategorizzazioni superflue.

Per una migliore comprensione del processo di disambiguazione, riportiamo in Tav. 5 le linee di programma riferite alla funzione "assegna", seguite dalla relativa descrizione analitica.

A	`function assegna(campo, pos, end, cpn, cp, csp, sp, spl, cl)` `  {`
B	`  cpn = 1`
C	`  cp = split (campo, sp, /\);\(/)`
D	`  csp = split (sp[1], spl, /\(/)`
E	`  pos = pos ","`
F	`while (cpn <= cp)` `  {`
G	`  if (cpn > cp)` `  break`
H	`  if (sp[cpn] ~ pos)` `  {`

I	`  if (sp[cpn] ~ /\)$/ \|\| sp[cpn] ~ /\)¥$/)`
	`  {`
	`  cl = sp[cpn]`
	`  sub (/\)/, "", cl)`
	`  print spl[1] cl`
	`  }`
	`  else`
	`  if (cpn == 1)`
	`  print spl[1] spl[2] end`
	`  else`
	`  print spl[1] sp[cpn] end`
	`  }`

J	`  cpn++`
	`  }`
	`  }`

Tav. 5a-j: La funzione "assegna"

A contiene la dichiarazione della funzione, dei parametri e delle variabili adottate e B la dichiarazione della variabile "cpn" ed assegnazione del valore numerico "1".

C fa uso della funzione predefinita “split” al fine di separare le varie transcategorizzazioni inserendo i diversi valori di POS in una tabella (*array*).

In D, poi, si utilizza di nuovo “split” per separare il token dal gruppo di transcategorizzazioni.

In E si inserisce un segno di virgola al termine della stringa di caratteri numerici convogliata dal parametro “pos”.

In F, per mezzo del comando “while” e l’uso della variabile “cpn”, si istituisce un ciclo iterativo per scansionare le varie POS presenti nell’*array* precedentemente costituito.

G, quindi, verifica il punto di scansione per l’interruzione al momento opportuno del ciclo iterativo; e H seleziona la categoria corretta mediante il confronto tra il contenuto del parametro “pos” e le POS transcategorizzanti oggetto di scansione.

I, in caso di esito positivo del confronto, ricostruisce e stampa su file la nuova linea di testo etichettata. Il simbolo “¥” viene utilizzato al fine di permettere, al termine dell’elaborazione del modulo finale, il ripristino della formattazione del testo del file originale.

J, infine, in caso di esito negativo, incrementa la variabile “cpn” e continua il ciclo iterativo di scansione.

BIBLIOGRAFIA.

ARMSTRONG
1994 *Using Large Corpora*, edited by Susan Armstrongs, Cambridge (Mass.) - London (En.), The MIT Pres, 1994 “A Bradford Book”, “ACL-MIT Press Series in Computational Linguistics” [= “Computational Linguistics” XIX (1993)[1-2]].

ATWELL - SOUTER 1993 → SOUTER - ATWELL 1993

BARBERA
¶ 1 Manuel Barbera, *Per la storia di un gruppo di ricerca. Tra bmanuel.org e corpora.unito.it*, in questo volume, pp. 3-20.
¶ 8 Manuel Barbera, *Un tagset per il Corpus Taurinense. Italiano antico e linguistica dei corpora*, in questo volume, pp. 135-168.

BRENNAN
2000 Michael Brennan, *GAWK: Effective AWK Programming: A User’s Guide for GNU Awk*, 2nd edition, Free Software Foundation Inc., 2000, disponibile online alla pagina: `http://www.gnu.org/software/gawk/manual/gawk.html`.

GUTHRIE
1993 Louise Guthrie, *A Note on Lexical Disambiguation*, in SOUTER - ATWELL 1993, pp. 11-24.

HINDLE - ROOTH
1994 Donald Hindle - Mats Rooth, *Structural Ambiguity and Lexical Relations*, in ARMSTRONG 1994, pp. 103-120.

MASON
2000 Mason, Oliver, *Programming for Corpus Linguistics - How to Do Text Analysis with Java*, Edinburgh, Edinburgh University Press, 2000 “Edinburgh Textbooks in Empirical Linguistics”.

MATTHEWS
1998 Matthews, Clive, *An Introduction to Natural Language Processing through Prolog*, New York, Longman, 1998 “Learning about language”.

MITKOV
2003 *The Oxford Handbook of Computational Linguistics*, edited by Ruslan Mitkov, Oxford, Oxford University Press, 2003.

SOUTER - ATWELL
1993 *Corpus-based Computational Linguistic*s, edited by Clive Souter and Eric Atwell, Amsterdam - Atalanta, Rodopi, 1993 "Language and Computers: Studies in Practical Linguistics" 9.

STEVENSON - WILKS
2003 Mark Stevenson - Yorick Wilks, *Word-Sense Disambiguation*, in MITKOV 2003, pp. 249-265.

VOUTILAINEN
2003 Atro Voutilainen, *Part-of-Speech Tagging*, in MITKOV 2003, pp. 218-232 [soprattutto § 11.6 *Handwritten Disambiguation Grammars*, pp. 227-230].

WEISCHEDEL et alii
1994 Ralph Weischedel - Marie Meteer - Richard Schwartz - Lance Ramshaw - Jeff Palmucci, *Coping with Ambiguity and Unknown Words through Probabilistic Models*, in ARMSTRONG 1994, pp. 319-342.

CORPORA DI RIFERIMENTO.

Corpus Taurinense → CT.

CT `http://www.bmanuel.org/projects/ct-HOME.html`.