9. La disambiguazione.
Trattamento finale. [MT]

9.0 INTRODUZIONE. Una delle prerogative distintive del *Corpus Taurinense*, come dicevamo nella *Premessa* (§ 1.1), è che i suoi testi sono stati «annotati e completamente disambiguati per parti del discorso»: il che vale a dire che, come passo successivo al POS-tagging (per la cui impostazione cfr. il ¶ 4, per la sua messa in pratica il ¶ 7, e per il suo raffinamento tutta la sezione IIIJ), è stato predisposto un consistente processo di disambiguazione al fine di garantire l'assegnazione univoca delle varie classi grammaticali (POS) ai diversi token costituenti l'intero testo. Nel seguente capitolo[1] si illustreranno, in modo preliminare, i problemi teorici e tecnici con cui è stato necessario confrontarsi per la disambiguazione del CT e, più nel dettaglio, le procedure e le soluzioni computazionali successivamente adottate.

"Disambiguazione", bisogna premettere, può veicolare diversi significati anche solo all'interno del paradigma della *corpus linguistics*. Se in generale tale termine esprime il concetto di eliminazione o riduzione del grado di ambiguità posseduto da un determinato elemento presente all'interno di un sistema complesso, noi ne faremo un uso assai più limitato: in primo luogo, è ovviamente (essendo giocoforza i testi del CT *testi scritti*) esclusa dalla nostra accezione la gestione di qualsiasi informazione di tipo sonoro (omofonia). Eliminata questa area di fenomeni, resta che "disambiguazione" può ancora fare riferimento ad almeno 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 (ambiguità lessicale), 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 (ambiguità grammaticale). Sebbene l'elaborazione computazionale della semantica dei vari token costituenti il testo sia un settore di ricerca molto complesso ed in piena evoluzione, che richiede l'uso di strumenti appropriati quali ontologie e reti semantiche, non è stato possibile per il CT predisporre procedure automatizzate per ciò fare, ma ci siamo accontentati di affrontare in modo limitato e manuale il problema in sede di revisione dell'annotazione, secondo le linee descritte nel § 16.2. Restava quindi solo di occuparci del processo di disambiguazione lessicale al livello grammaticale; ed a tale proposito va notato che se la disambiguazione testuale di natura semantica è obbligatoriamente vincolata all'analisi del contesto specifico, il processo di disambiguazione grammaticale può essere elaborato secondo modelli computazionali sia di tipo *context sensitive* (sensibili al contesto), sia *context free* (svincolati dal contesto); e 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.

Ma, va subito avvertito, non di tutte le ambiguità grammaticali bisognava per forza occuparsi. Se trasformiamo la categoria generica di "ambiguità grammaticale" in quella tecnica, specifica della *corpus linguistics*, di transcategorizzazione (cfr. il § 6.5 per il suo inquadramento), ne possiamo individuare tre tipi: transcategorizzazioni esterne, ossia tra POS diverse, interne, ossia tra MSF diverse, ed intra-POS, ossia tra diversi tipi di una stessa POS (cfr. § 6.5.1). Di questi, l'impalcatura delle raccomandazioni EAGLES>ISLE prescrive la necessaria eliminazione delle prime (ed in subordine delle terze), ma non necessariamente delle seconde. Abbiamo pertanto concentrato le nostre forze sulle transca-

[1] Che sostanzialmente riproduce, di poco adattandolo, Tomatis 2007.

tegorizzazioni esterne ed intra-POS, in genere non intervenendo su quelle interne; in termini formali, ossia, sono sempre e solo risolte tutte le transcategorizzazioni che nel nostro bastone di annotazione hanno la forma

```
(...);(...)
```

Il risultato è stato un corpus disambiguato in modo sostanzialmente completo. E dico *sostanzialmente* solo perché è rimasto un residuo ineliminabile[2] di 2 casi per 8 token totali,

[56a]
```
l'      5     (lem=lo,60,0,4,6;7,0,0);(lem=la,60,0,5,6;7,0,0)
÷l1'    3     (lem=lo,60,0,4,6;7,0,0);(lem=la,60,0,5,6;7,0,0)
```
dal *effedue*.

[56b] `Ladorneccio è una palese tolta de` **`l' altrui`** `contra | $0053$ la volontà de ÷l segnore .` Bono, *Libro vizi*, xxviiij.9, p. 52.

come si può constatare con la query

[57]
```
[lemma=".*\|.*"]
```
query CQP[3].

9.0.1 SISTEMI DI DISAMBIGUAZIONE: UNA PANORAMICA GENERALE. 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 per 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, 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 *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, è 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

[2] Ineliminabile perché in tutti ed otto i casi l'articolo modifica una forma che è lei stessa inerentemente "ancipite" a livello di MSF: *altrui*, `lem=altrui,32,0,4;5,6;7,0,0`.

[3] Poi query 1227a nel § 16.3.

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 era 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.

9.0.2 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, è la 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 si è risolto nell'intervento 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 perlopiù *context sensitive*, ossia strettamente legate al contesto in cui operano) una struttura estremamente flessibile, leggera, versatile e facilmente adattabile ad ulteriori aggiunte o modifiche.

9.1 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[4] (*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: ognuno 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.

9.1.1 CARATTERISTICHE SALIENTI DEL LINGUAGGIO DI SCRIPTING ADOTTATO. Come già accennato in precedenza, GAWK 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. GAWK, 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 *content 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.

Il modulo `dis_prep`:

```
# Source formatting module
#
{
gsub (/^ /, "")
print $0 "¥"
}
```

Il modulo `dis_end`:

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

Tav. 105: I moduli `dis_prep` e `dis_end`.

Questa soluzione, affatto 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

[4] Nel prosieguo non saranno commentate le varie "labels" del tagset del CT, per il quale basta rimandare al capitolo 4 del presente volume.

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.

9.1.2 OTTIMIZZAZIONE DEL SISTEMA. *Last but not least*, per 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 del 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 stabilire 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 cui scelta, 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.

9.2 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 comples-

sità 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.

9.2.1 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. 106: Le linee di commento.

9.2.2 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. GAWK 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. 107: 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 GAWK 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 GAWK così da adattarlo agli scopi dell'utente. Di norma GAWK 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 stam-

pata mediante il comando `print`. Come già detto, la configurazione appena descritta non risulta adeguata agli scopi del nostro progetto, pertanto si rende necessaria una sostanziale modifica di tale comportamento. Poiché GAWK 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, GAWK 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, questo 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.

9.2.3 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 § 9.1.1, l'impostazione delle regole di eliminazione virtuale dei codici testuali non pertinenti (cfr. § 9.1.2) ed infine il motore di disambiguazione vero e proprio (cfr. § 9.1.3), costituito da regole linguistiche (cfr. § 9.3) e funzioni definite dall'utente (cfr. § 9.4).

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.

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 finora, la variabile `NF` richiede invece un commento più articolato. Come già accennato in precedenza, il linguaggio di programma-

zione 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.

```
{
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. 108: Corpo del programma.

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 quando `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.

9.3 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 *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.

9.3.1 ESEMPIO DI REGOLA TRATTA DAL 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 Tav. 109a-f se ne fornisce un esempio (in corpo ridotto per tirannia di spazio), che analizzeremo partitamente.

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 GAWK mediante l'uso del simbolo speciale "~".

B introduce ulteriori elementi di controllo finalizzati alla corretta gestione del marcatore di fine riga (cfr. § 9.1.1).

C è la riga per l'incremento della variabile `nf` che scansiona il testo (cfr. § 9.1.1).

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. § 9.4).

A
```
# Regola per la disambiguazione interna
# ed esterna della forma 'ave'
  else
  if (campo ~ /^ave_/ && campo ~ /\);\(/)
      {
```
B
```
      if (campo ~ /¥$/)
          end = "¥"
      else
          end = ""
```
C
```
      nf++
```
D
```
      if ($fw ~ /^÷lle_/)
          {
          assegna(campo, "221", end)
          }
      else
      if ($fw ~ /^mari[ae]_/)
          {
          assegna(campo, "68", end)
          }
```
E
```
      else
      if ($fw ~ /^÷gli_/)
          {
          sub (/;3/, "", campo)
          sub (/6;/, "", campo)

  assegna(campo,"211",end)
          }
```
F
```
else
          {
          sub (/2;/, "", campo)
          sub (/;7/, "", campo)

  assegna(campo,"211",end)
          }
      }
```

Tav. 109a-f: Una regola di disambiguazione del modulo 1.

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, che 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.

9.3.2 ESEMPIO DISTRIBUITO SU TRE MODULI: IL TRATTAMENTO DELLE FORME STRANIERE. Nel sistema di disambiguazione adottato per il *Corpus Taurinense*, non è rara la presenza di regole la cui capacità di intervento su una HDF specifica è distribuita su vari moduli distinti.

Un esempio di tale distribuzione lo si incontra prendendo in esame la serie di regole che gestiscono la disambiguazione della POS 75, relativa ai termini stranieri. Questa categoria lessicale vieni infatti trattata, con regole differenti, da ben tre moduli, il primo, il terzo ed il quarto. Il modulo 2 ospita alcune regole *ad hoc*; il modulo 3 rappresenta un caso di "modulo misto" in cui sono presenti sia regole che si rifanno al contesto, sia regole più generali; il modulo 4, infine, risolve i casi rimanenti sfruttando una sola regola, parzialmente generale. Per una migliore comprensione del problema, è necessario puntualizzare che la presenza di regole *ad hoc* distribuite su più moduli distinti non è un fatto casuale, come non è altrettan-

to casuale la distribuzione delle regole generali. L'inserimento delle tre regole nel modulo 2, che come si può osservare agiscono su forme di tipo nominale, è giustificato dal fatto che tale modulo gestisce prevalentemente la disambiguazione della categoria *nome*, disambiguazione che verrà utilizzata in vario modo dalle regole contestuali dei moduli seguenti. Pertanto, data la complessità della casistica contestuale in cui vengono a trovarsi le forme di origine straniera, si è reso necessario suddividere l'intiera problematica in frazioni più semplici, più comodamente risolvibili. La scelta effettuata, pur non costituendo forse la soluzione migliore in assoluto, si è rivelata nel suo complesso adeguata al fine di agevolare la formulazione delle regole generali che, attivandosi in momenti di analisi successivi, possono sfruttare le disambiguazioni già precedentemente operate dalle regole *ad hoc*.

Per maggiore chiarezza e completezza dell'esposizione, riportiamo ora alcuni esempi di regole precedute dall'indicazione del modulo in cui agiscono:

modulo 2

```
# Regola per la disambiguazione esterna e interna di 'pater'
    else
    if (campo ~ /^pater_/ && campo ~ /\);\(/)
        {
        if (campo ~ /¥$/)
            end = "¥"
        else
            end = ""
        nf++
        if ($fw ~ /lem=nostro,33,/)
            {
            sub (/6;/, "", campo)
            assegna(campo, "20", end)
            }
        else
            {
            assegna(campo, "75", end)
            }
        }
```

modulo 3

```
# Regola per la disambiguazione esterna di forma straniera
    else
    if (campo ~ /,75,/ && campo ~ /\);\(/)
        {
        ffw = fw
        ffw++
        bbw = bw
        bbw--
        if (campo ~ /¥$/)
            end = "¥"
        else
            end = ""
        nf++
        if (($bw ~ /,75,/ && $fw ~ /,75,/ && ($fw !~ /\);\(/ || $bw !~
/\);\(/)) || ($bw ~ /,75,/ && $bbw ~ /,75,/ && ($bw !~ /\);\(/ || $bbw
!~ /\);\(/)) || ($bw ~ /,75,/ && $fw ~ /,71,/ && $ffw ~ /,75,/ && ($ffw
!~ /\);\(/ || $bw !~ /\);\(/)) || ($fw ~ /,75,/ && $bw ~ /,71,/ && $bbw
~ /,75,/ && ($fw !~ /\);\(/ || $bbw !~ /\);\(/)) || ($fw ~ /,75,/ &&
```

```
$ffw ~ /,75,/ && ($fw !~ /\);\(/ || $ffw !~ /\);\(/)) || ($bw ~ /,75,/
&& $bw !~ /\);\(/ && $fw ~ /lem=italicsclosed/) || ($fw ~ /,75,/ &&
$fw !~ /\);\(/ && $bw ~ /lem=italicsopen/))
            {
            assegna(campo, "75", end)
            }
        else
        print campo
        }
```

modulo 4

```
# Regola per la disambiguazione esterna di forma straniera
    else
    if (campo ~ /,75,/ && campo ~ /\);\(/)
        {
        ffw = fw
        ffw++
        bbw = bw
        bbw--
        if (campo ~ /¥$/)
            end = "¥"
        else
            end = ""
        nf++
        if (($bw ~ /,75,/ && $bw !~ /\);\(/) || ($fw ~ /,75,/ && $fw
!~ /\);\(/ && $fw !~ /^dies_/))
            {
            assegna(campo, "75", end)
            }
        else
            {
            assegna4(campo, "75", end)
            }
        }
```

Tav. 110abc: Una disambiguazione distribuita su tre moduli.

9.4 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 GAWK 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. § 9.3.1) 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 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. 111a-j: le righe 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. 111a-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.

9.5 APPENDICE: DAL MOTORE DI TRANSCATEGORIZZAZIONE AL "PEX". Si è detto come in campi quali la disambiguazione contestuale, la maggior parte delle innovazioni appartengono al ramo delle tecnologie di funzionamento (es. sistemi a regole vs. sistemi stocastici) piuttosto che a questioni metodologiche vere e proprie. Nel caso del disambiguatore sviluppato per il trattamento del *Corpus Taurinense* di italiano antico, le ridotte dimensioni del corpus in questione, unitamente alla mancanza di un *training corpus* precedentemente annotato data la natura assolutamente pionieristica ed innovativa del progetto, ha condotto gli sviluppatori alla scelta obbligata di orientarsi verso la costruzione di un sistema basato su regole. Se il disambiguatore in quanto tale, descritto nei paragrafi precedenti, costituisce già in sé uno strumento di importanza rilevante ai fini della ricerca linguistica e filologica sull'italiano antico, l'iter relativo allo sviluppo delle regole che vi sono state elaborate contribuisce anche a fornire un valido spunto di innovazione a livello metodologico.

9.5.0 L'ITER DI DISAMBIGUAZIONE ED IL "PEX". Relativamente a quest'ultimo punto, vorremmo qui innanzi tutto discutere la principale differenza tra l'iter di sviluppo comunemente adottato e quello impiegato nel disambiguatore oggetto di discussione.

In genere la realizzazione di un sistema di regole da implementare in un software di disambiguazione contestuale è caratterizzato dalle fasi di sviluppo:

(1) studio teorico dei vari tipi di regole;
(2) implementazione pratica delle stesse in un determinato formalismo, di solito strettamente dipendente dal programma di disambiguazione;
(3) attivazione del processo di elaborazione;
(4) verifica sul testo disambiguato del corretto operato delle regole;
(5) correzione, nella cosiddetta fase di post-editing, di quelle regole che per vari motivi disattendono i risultati previsti;
(6) rielaborazione del testo ambiguo originale.

Il sistema per lo sviluppo delle regole in italiano antico, invece, prevede i passaggi:

(1) studio teorico dei vari tipi di regole;
(2) implementazione virtuale delle singole regole mediante l'utilizzo di un apposito sistema di emulazione;
(3) analisi dei dati riportanti il funzionamento del sistema;
(4) eventuale correzione delle regole;
(5) elaborazione del testo da disambiguare.

Come si può immediatamente notare, quest'ultimo sistema, utilizzato all'interno del progetto, risulta nettamente più veloce in quanto richiede un minor numero di passaggi. In particolare, evita la necessità di dover iterare più volte il processo di elaborazione, azione che, in genere, richiede un tempo relativamente elevato, direttamente proporzionale alla complessità del gruppo di regole di disambiguazione da processare. Il punto di forza del sistema di pre-verifica qui descritto, denominato PEX (*PatternEXtractor*), risiede nell'elevata flessibilità del programma di emulazione. Come la maggior parte degli script svilup-

pati all'interno del progetto, anch'esso utilizza il linguaggio GAWK per il suo funzionamento; tuttavia, a differenza degli altri, non rappresenta un'entità chiusa scarsamente adattabile, bensì, pur mantenendo stabile la sua struttura di base, permette facili modifiche in modo da adeguarsi comodamente alle differenti esigenze dell'utente.

9.5.1 DUE ESEMPI DI "PEX". Verranno ora mostrati due esempi di PEX, estremamente differenti in termini di difficoltà delle regole contenute, che permetteranno al lettore di comprendere meglio la modularità e versatilità del sistema.

Il primo esempio, più compatto, presenta la verifica di una regola per la forma *questi*:

```
# PEX v.0.2
# PATTERN EXTRACTOR
# Bidirectional Transcategorization Extraction Module.
BEGIN {
RS = ""
nf = 1
}
{
while (nf <= NF)
     {
     if ($nf ~ /^questi_/)
          {
          campo1 = $nf
          cp = nf
          cp--
          fw = nf
          fw++
     if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\£/ || $fw ~ /\#/)
               fw++
     if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\£/ || $fw ~ /\#/)
               fw++
     if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\£/ || $fw ~ /\#/)
               fw++
          if ($fw ~ /,[123][12][12345678],/ && $fw ~ /,6,/ && $fw !~ /\);\(/)
               {
               nf++
               campo2 = $nf
               nf++
               campo3 = $nf
               nf++
               campo4 = $nf
               nf++
               campod = $cp
               cp--
               campoc = $cp
               cp--
               campob = $cp
               cp--
               campoa = $cp
               print campoa, campob, campoc, campod, campo1,
campo2,campo3, campo4, " "
               }
          else
```

```
            nf++
            }
      else
      nf++
      }
}
```

Tav. 112a: Listato di PEX per *questi*, script in GAWK.

Il secondo esempio, invece, assai più complesso ed articolato, verifica un gruppo di regole per la forma *d'*:

```
# PEX v.0.2
# Modulo D'=di_da:
BEGIN {
RS = ""
nf = 20
}
{
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 = $nf
      fw = nf
      fw++
# omette le stringhe contenenti:
# '@'
# '%'
# '$'
# '£'
# '#'
      if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\£/ || $fw ~ /\#/)
            fw++
      if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\£/ || $fw ~ /\#/)
            fw++
      if ($fw ~ /\@/ || $fw ~ /\%/ || $fw ~ /\$/ || $fw ~ /\£/ || $fw ~ /\#/)
            fw++
      ffw = fw
      ffw++
      bw = nf
      if (nf >=2)
            bw--
      if (($bw ~ /\@/ || $bw ~ /\%/ || $bw ~ /\$/ || $bw ~ /\£/ ||
$bw ~ /\#/) && bw > 2)
            bw--
      if (($bw ~ /\@/ || $bw ~ /\%/ || $bw ~ /\$/ || $bw ~ /\£/ ||
$bw ~ /\#/) && bw > 2)
            bw--
      if (($bw ~ /\@/ || $bw ~ /\%/ || $bw ~ /\$/ || $bw ~ /\£/ ||
$bw ~ /\#/) && bw > 2)
            bw--
```

```
     bbw = bw
     bbw--
# salta i campi che non contengono l'elemento da cercare (d')
     if (campo !~ /^d'_/)
          {
          nf++
          }
# Serie di query CQP trasformate in regole AWK
     else
     if ($fw ~ /,[123]21,/)
          {
          if (($bw ~ /^è_/ && ($fw ~ /^avere_/ || $fw ~ /^aiutare_/
|| $fw ~ /^aprendere_/ || $fw ~ /^operare_/)) || $fw ~ /^amonire_/
|| $fw ~ /^amaestrare_/)
               {
               nf++
               }
          else
          if (($bw ~ /^tempo_/ && $fw ~ /^attendere_/) || ($bw ~
/^parrà_/ && $fw ~ /^avere_/) || ($bw ~ /^via_/ && ($fw ~
/^acquistar_/ || $fw ~ /^intendere_/)) || $bw ~ /^apparecchiamento_/
|| $bw ~ /^guardassi_/ || $bw ~ /^passaggio_/)
               {
               nf++
               }
          else
               {
               nf++
               }
          }
     else
     if (($bw ~ /^che_/ && $fw ~ /^amore_/) || ($fw ~ /^una_/ &&
$ffw ~ /^parte_/) || ($fw ~ /^ogn[ie]_/ && ($ffw ~ /^canto_/ || $ffw
~ /^lato_/ || $ffw ~ /^parte_/)) || ($bw ~ /^ha_/ && $fw ~ /^alta_/)
|| ($bw ~ /^mosso_/ && $fw ~ /^ottima_/) || ($bw ~ /^muta_/ && $fw ~
/^un_/) || (($bw ~ /^solo_/ || $bw ~ /^morto_/) && $fw ~ /^una_/) ||
($bw ~ /^die_/ && $fw ~ /^Andrea_/) || ($bw ~ /^e_/ && $fw ~
/^altra_/) || ($bw ~ /^o_/ && $fw ~ /^alcuna_/) || ($bw ~ /^e_/ &&
$fw ~ /^un_/ && $ffw ~ /^tempo_/) || (($bw ~ /^sapremo_/ || $bw ~
/^o_/ || $bw ~ /^inparare_/) && $fw ~ /^altrui_/) || ($bw ~ /^ella_/
&& $fw ~ /^Amore_/) || ($bw ~ /^accompagnata_/ && $fw ~ /^alcuna_/)
|| ($bw ~ /^consiglio_/ && $fw ~ /^ivi_/) || ($bw ~ /^sol_/ && $fw ~
/^una_/ && $ffw ~ /^donna_/) || ($bw ~ /^tratto_/ && $fw ~
/^occhi_/) || ($fw ~ /^ogni_/ && $ffw ~ /^luogo_/) || $fw ~
/^entro_/)
          {
          nf++
          }
     else
     if (($bw ~ /^che_/ && $fw ~ /^alta_/) || ($bw ~ /^presso_/ &&
$fw ~ /^alcuno_/) || ($bw ~ /^voi_/ && $fw ~ /^angeli\&\[ca\&\]_/)
|| ($bw ~ /^nascono_/ && ($fw ~ /^Invidia_/ || $fw ~ /^Ira_/ || $fw
~ /^Avarizia_/)) || (($bw ~ /^lei_/ || $bw ~ /^e_/ || $bw ~ /^,_/)
```

```
&& $fw ~ /^ogni_/) || ($bw ~ /^nate_/ && $fw ~ /^una_/) || ($bw ~
/^guarde_/ && $fw ~ /^avarizia_/) || ($bw ~ /^e_/ && $fw ~
/^altrui_/) || ($bw ~ /^batte_/ && $fw ~ /^intorno_/) || ($bw ~
/^o_/ && ($fw ~ /^ovo_/ || $fw ~ /^altr[ao]_/)) || ($bw ~
/^informate_/ && $fw ~ /^angelica_/) || (($bw ~ /^mondo_/ || $bw ~
/^veggono_/ || $bw ~ /^e_/ || $bw ~ /^trovai_/ || $bw ~ /^là_/ ||
$bw ~ /^femine_/ || $bw ~ /^ride_/ || $bw ~ /^castella_/ || $bw ~
/^volse_/ || $bw ~ /^veggono_/) && $fw ~ /^intorno_/) || ($bw ~
/^giunto_/ && $fw ~ /^alcun_/) || ($bw ~ /^guarda_/ && $fw ~
/^una_/) || ($bw ~ /^÷ti_/ && $fw ~ /^ogne_/) || ($bw ~ /^÷ti_/ &&
$fw ~ /^esso_/) || ($bw ~ /^che_/ && $fw ~ /^orgogliose_/) || ($bw ~
/^,_/ && $fw ~ /^invidia_/) || ($bw ~ /^proviene_/ && $fw ~
/^eloquenzia_/) || ($bw ~ /^muovono_/ && $fw ~ /^onesto_/) || ($bw ~
/^e_/ && $fw ~ /^utile_/) || ($bw ~ /^nato_/ && $fw ~
/^onestissime_/) || ($bw ~ /^furato_/ && $fw ~ /^una_/) || ($bw ~
/^calice_/ && $fw ~ /^uno_/) || ($bw ~ /^sacrata_/ && $fw ~ /^uno_/)
|| ($bw ~ /^tolse_/ && $fw ~ /^uno_/) || ($bw ~ /^cioè_/ && $fw ~
/^una_/) || ($bw ~ /^ma_/ && $fw ~ /^un_/ && $ffw ~ /^altro_/) ||
($bw ~ /^difensione_/ && $fw ~ /^un_/ && $ffw ~ /^altro_/) || ($bw ~
/^nasce_/ && $fw ~ /^alcuna_/) || ($bw ~ /^ritraeva_/ && $fw ~
/^altre_/) || ($bw ~ /^che_/ && $fw ~ /^esse_/) || ($bw ~
/^estratto_/ && $fw ~ /^alto_/) || ($bw ~ /^cacciat[io]_/ && $fw ~
/^altra_/) || ($bw ~ /^movea_/ && $fw ~ /^amoroso_/) || ($bw ~
/^procede_/ && $fw ~ /^un'_/ && $ffw ~ /^anima_/) || ($bw ~
/^levato_/ && $fw ~ /^una_/) || ($bw ~ /^che_/ && $fw ~ /^allora_/)
|| ($bw ~ /^notaio_/ && $fw ~ /^Oltra\&\[r\&\]no_/) || (($bw ~
/^notaio_/ || $bw ~ /^viturale_/ || $bw ~ /^alberchiere_/) && $fw ~
/_lem=orvieto,/) || ($bw ~ /^÷e_/ && $fw ~ /^uno_/ && $ffw ~
/^paese_/) || ($bw ~ /^terra_/ && $fw ~ /^un_/) || ($bw ~ /^tanto_/
&& $fw ~ /^una_/) || ($bw ~ /^viandante_/ && $fw ~ /^albergho_/) ||
($bw ~ /^che_/ && $fw ~ /^allora_/ && $ffw ~ /^innanzi_/) || ($bw ~
/^tratto_/ && $fw ~ /^ongne_/) || ($bw ~ /^tornò_/ && $fw ~
/^oltremare_/) || ($bw ~ /^nato_/ && $fw ~ /^Inghilterra_/) || ($bw
~ /^passare_/ && $fw ~ /^un_/ && $ffw ~ /^picciolo_/) || ($bw ~
/^crociati_/ && $fw ~ /^Italia_/) || ($bw ~ /^nasscerà_/ && $fw ~
/^una_/) || ($bw ~ /^nato_/ && $fw ~ /^Asscholi_/) || ($bw ~
/^venia_/ && $fw ~ /^Alangna_/) || ($bw ~ /^cardinale_/ && $fw ~
/^Acquassparte_/) || ($bw ~ /^ritornò_/ && $fw ~ /^Inghilterra_/) ||
($bw ~ /^togliesse_/ && $fw ~ /^ogni_/) || ($bw ~ /^÷e_/ && $fw ~
/^una_/ && $ffw ~ /^in_/) || ($bw ~ /^gravati_/ && $fw ~
/^angosciosa_/) || ($bw ~ /^,_/ && $fw ~ /^una_/ && $ffw ~
/^scuritate_/) || ($bw ~ /^e_/ && $fw ~ /^allegrezza_/ && $ffw ~
/^è_/) || ($bw ~ /,21,/ && $bw !~ /^Dio_/ && $bw !~ /^Dido_/ && $fw
~ /,21,/) || ($fw ~ /^altra_/ && $ffw ~ /^maniera_/) || ($fw ~
/^ogn[ei]_/ && $ffw ~ /^tempo_/) || ($bw ~ /^fuori_/ && $fw !~
/^un_/) || ($bw ~ /^For_/ && $fw ~ /^ogne_/) || ($bw ~
/^mercatante_/ && $fw ~ /^Egitto_/) || ($bw ~ /^sapesse_/ && $fw ~
/^arte_/) || (($bw ~ /^sapremo_/ || $bw ~ /^inparare_/) && $fw ~
/^altrui_/) || ($bw ~ /^cittadini_/ && $fw ~ /^uno_/) || ($bw ~
/^segnore_/ && $fw ~ /^esto_/) || (($bw ~ /lem=podestà,/ || $bw ~
/lem=re,/ || $bw ~ /lem=conte,/ || $bw ~ /lem=duca,/ || $bw ~
/lem=cavaliere,/ || $bw ~ /lem=vescovo,/ || $bw ~ /lem=cardinale,/
|| $bw ~ /lem=principe,/ || $bw ~ /^nonane_/ || $bw ~ /lem=notaio,/
```

```
|| $bw ~ /^cheric/ || $bw ~ /^mercatant/ || $bw ~ /^alberchier/) &&
$fw ~ /,21,/) || ($bw ~ /lem=essere/ && ($fw ~ /^Egitto_/ || $fw ~
/^Italia_/)) || $fw ~ /^oltremonte_/ || $fw ~ /^onni_/ || $fw ~
/^oltremirabile_/)
          {
          nf++
          }
     else
     if (campo ~ /^d'_/)
          {
          gsub (/_.*/, "", campo)
          back = nf
          back--
          forw = nf
          forw++
          lineout = campo
          while ($back !~ /_lem=stop,/)
               {
               vback = $back
               if (vback ~ /\@/ || vback ~ /\%/ || vback ~ /\$/ ||
vback ~ /\£/ || vback ~ /\#/)
                    back--
               else
                    {
                    gsub (/_.*/, "", vback)
                    lineout = vback " " lineout
                    back--
                    }
               }
          while ($forw !~ /_lem=stop,/)
               {
               vforw = $forw
               if (vforw ~ /\@/ || vforw ~ /\%/ || vforw ~ /\$/ ||
vforw ~ /\£/ || vforw ~ /\#/)
                    forw++
               else
                    {
                    gsub (/_.*/, "", vforw)
                    lineout = lineout " " vforw
                    forw++
                    }
               }
          vforw = $forw
          gsub (/_.*/, "", vforw)
          lineout = lineout " " vforw
          print lineout
          nf++
          }
     }
}
```

Tav. 112b: Listato di PEX per *d'*, script in GAWK.

Da una prima visione degli script, si nota immediatamente l'estrema semplicità del motore del programma che, di fatto, si riduce unicamente a un comando `while`, presente

all'inizio, utile per la scansione dei token contenuti nelle righe di testo e presenta, nella parte finale, il gruppo di istruzioni utili per acquisire e limitare il contesto della frase che verrà prodotto in uscita. La parte centrale dello script, invece, è riservata a ospitare una o più regole oggetto di verifica, modificabili a piacere.

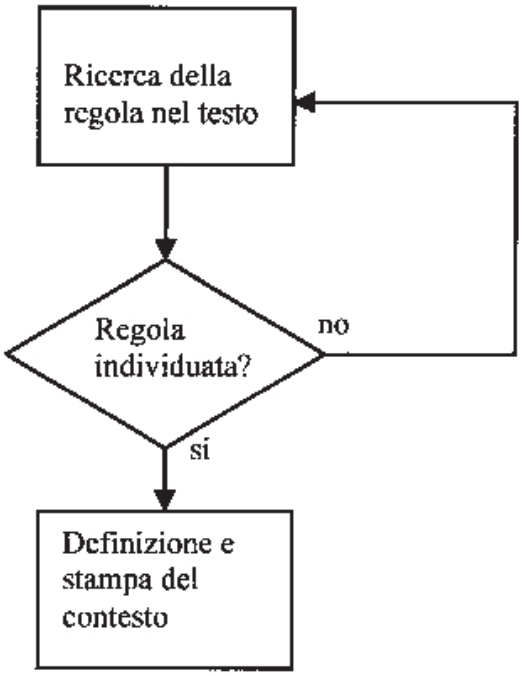
9.5.2 TRA REGOLE CQP E PATTERN "PEX". È importante ricordare che mediante il corretto utilizzo delle espressioni regolari, è possibile convertire qualsivoglia query prodotta in linguaggio CQP in un *pattern* di ricerca facilmente interpretabile e gestibile dal sistema di verifica PEX.

Ad esempio l'espressione in linguaggio CQP 58a può essere convertita in GAWK mediante espressioni regolari in 58b, pronta per essere usata in PEX:

[58a]
```
[pos=".*n.c.*" & lemma!="(fiorino)|(soldo)|(denaro)|(libbra)|(marco)"]
[pos=".*adj.*"]? [word="d."]
[lemma="(avorio)|(argento)|(oro)|(acciaio)|(albero)|(anguilla)|(o
lio)|(acqua)|(aria)"]
```
query CQP,

[58b]
```
if (campo ~ /^d'_/ && ($fw ~ /lem=avorio,/ || $fw ~ /lem=argento,/
|| $fw ~ /lem=oro,/ || $fw ~ /lem=acciaio,/ || $fw ~ /lem=albero,/
|| $fw ~ /lem=anguilla,/ || $fw ~ /lem=olio,/ || $fw ~
/lem=acqua,/ || $fw ~ /lem=aria,/) && $bw ~ /,20,/ && $bw !~
/lem=fiorino,/ && $bw !~ /lem=soldo,/ && $bw !~ /lem=denaro,/ &&
$bw !~ /lem=libbra,/ && $bw !~ /lem=marco,/)
```
PEX pattern.

9.5.3 Struttura globale del "PEX". L'estrema semplicità della struttura del programma si evidenzia anche dallo schema di flusso, riportato di seguito:



Tav. 113: Diagramma di flusso del PEX.