Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Specialistica in Informatica

Studio e implementazione di uno strumento computazionale per l'analisi morfologica della lingua italiana

Tesi di Laurea in Linguistica Computazionale

Relatore:
Chiar.mo Prof.
Fabio Tamburini

Presentata da:
Matias Melandri

III Sessione
Anno Accademico 2009/10

Indice

Capitolo 1

Introduzione

Il linguaggio è la caratteristica del processo cognitivo che distingue l'uomo dalle altre specie. In campo informatico la modellizzazione del linguaggio naturale è un problema che viene affrontato da lungo tempo: si cerca di raggiungere la capacità di comprendere, tradurre e generare il linguaggio con diverse finalità (comprensione, interpretazione, traduzione, controllo, ...). Il compito è difficile, a causa dell'insita complessità del linguaggio naturale ed i diversi aspetti di questo problema sono stati affrontati in un gran numero di settori.

Questo lavoro si inserisce nel campo della morfologia, ovvero quell'ambito d'indagine che ha per oggetto lo studio della struttura grammaticale delle parole: ne stabilisce la classificazione e l'appartenenza a determinate categorie, e le forme della flessione, come la coniugazione per i verbi, la declinazione per i sostantivi e la costruzione di forme derivate mediante affissazione.

Nel campo della morfologia computazionale, i primi programmi di elaborazione del linguaggio naturale, sviluppati fino agli anni '80, usavano lemmari tabulari di migliaia di termini per effettuare la classificazione delle parole. I primi prototipi, tuttavia, erano difficilmente utilizzabili data la scarsità di vocabolari elettronici.

L'approccio utilizzato attualmente si deve ad una svolta avvenuta nei primi anni '80, ad opera di Kimmo Koskenniemi [11]. La *two-level morphology*, da lui introdotta, si basa sullo sfruttamento di fenomeni morfologici comuni a molte parole. L'identificazione di queste regolarità, presenti nella struttura e nelle caratteristiche linguistiche delle parole, rende possibile l'utilizzazione del meta-linguaggio delle espressioni regolari e, di conseguenza, lo sfruttamento del formalismo degli automi a stati finiti, in particolare come trasduttori a stati finiti (FST). Ciò ha portato, quindi, allo sviluppo di sistemi basati su un insieme di dizionari di segmenti lessicali (basi, prefissi, suffissi, desinenze) e su un insieme di regole per la corretta formazione delle parole nelle varie lingue.

Il progetto descritto in questa tesi segue l'approccio introdotto da Koskenniemi, allo scopo di sviluppare un analizzatore morfologico open-source per la lingua italiana, *AnIta*.

Per l'effettiva implementazione di questo analizzatore, sono stati studiati ed analizzati in dettaglio vari software disponibili per lo sviluppo di strumenti per la morfologia computazionale. Lo strumento che è risultato ottimale per lo sviluppo del progetto descritto in questa tesi, è il toolkit open-source

HFST. Quest'applicazione è distribuita dall'Università di Helsinki; è un software recente, seguito ed aggiornato, e presenta quelle caratteristiche di efficienza, flessibilità e libertà del codice sorgente, che lo hanno reso adatto per lo sviluppo di AnIta.

Mediante gli strumenti messi a disposizione da HFST, sono state implementate le regole morfotattiche della lingua italiana, ovvero quelle regole composizionali che permettono di ottenere forme flesse tramite la combinazione dei singoli segmenti lessicali.

In accordo con il formalismo della *two-level morphology*, a fianco del modulo che gestisce le regole morfotattiche, è stato implementato un secondo modulo che contiene la gestione delle regole grafotattiche, ovvero quelle regole di riaggiustamento che risultano necessarie per modificare la forma dei segmenti lessicali in base al contesto in cui si trovano e gestire, quindi, quelle piccole alterazioni che compaiono all'interno di alcune forme.

L'inserimento dei lemmi che costituiscono il dizionario dell'analizzatore, non è avvenuto in maniera manuale. È stato considerato un importante e pre-esistente dizionario di circa 120.000 lemmi, predisposto per l'utilizzazione con un software proprietario. Questo è stato, quindi, analizzato, nuovamente etichettato e predisposto in modo da poter essere utilizzato tramite il formalismo degli FST, utilizzato da AnIta, l'analizzatore qui implementato.

Una volta terminata l'implementazione dei vari moduli che costituiscono l'analizzatore morfologico AnIta, questi sono stati compilati singolarmente ed, in seguito, composti assieme, per ottenere un trasduttore unico.

È stato necessario effettuare, poi, una fase di test su tutte le componenti di AnIta, volta ad individuare e correggere eventuali errori linguistici, come ad esempio l'errata generazione di forme inesistenti nella lingua italiana od il mancato riconoscimento di forme corrette. Al termine di questa fase, il trasduttore risulta, quindi, in grado di soddisfare le richieste che erano state poste all'inizio di questa tesi: l'esecuzione dei task di generazione e di analisi di forme e lemmi appartenenti alla lingua italiana.

Quando si parla di analisi, si intende la capacità di individuare, a partire da una forma flessa, tutti i lemmi che possono aver generato quella forma e tutte le caratteristiche che la contraddistinguono in maniera univoca. Quando si parla, invece, di generazione, si intende la capacità di generare, a partire da un lemma e dalle caratteristiche scelte, la forma flessa corrispondente.

Il lavoro svolto in questa tesi ha, quindi, prodotto uno strumento liberamente distribuibile e modificabile che permette la corretta generazione ed il riconoscimento di più di due milioni di forme della lingua italiana.

Infine, in questa tesi, viene fornita la descrizione di un procedimento, attraverso il quale risulta possibile espandere l'insieme di informazioni da gestire, per ogni forma riconosciuta e generata.

Viene messa a disposizione, infatti, la possibilità di memorizzare ulteriori informazioni di tipo morfologico o fonologico, attraverso la creazione di nuovi trasduttori dedicati, che vengono poi combinati tra loro, per ottenere una gestione personalizzata delle informazioni linguisticamente interessanti.

Viene, quindi, fornito un esempio relativo alla gestione della struttura

della forma flessa generata, relativamente ai morfemi che la compongono.

Nei capitoli che compongono questa tesi verrà descritto in dettaglio il processo che ha portato alla creazione di AnIta.

Nel capitolo 2 viene data una descrizione dell'ambito in cui si inserisce questo progetto, la morfologia computazionale, e nel capitolo 3, vengono illustrati i dettagli e le scelte implementative effettuate durante lo sviluppo di questa applicazione.

Il capitolo 4 contiene la descrizione della fase di testing eseguita al termine dell'implementazione e nel capitolo 5 sono descritte le modalità con cui è possibile estendere l'analizzatore qui implementato, in modo da gestire eventuali informazioni fonologiche o morfologiche aggiuntive, relative ad ogni parola riconosciuta. In questo capitolo è, inoltre, mostrata l'estensione progettata per AnIta.

Infine, il capitolo 6 contiene le conclusioni tratte da questo lavoro ed alcune considerazioni su un eventuale sviluppo futuro dell'applicazione. Viene, poi, fornita un'appendice che contiene un tutorial sull'utilizzazione e sulla possibile estensione di AnIta.

Capitolo 2

Morfologia computazionale

2.1 Morfologia

Una buona strategia per affrontare un problema di difficile soluzione è suddividerlo in parti che possano essere studiate indipendentemente: comprendere e modellizzare il funzionamento del linguaggio naturale è certamente uno di questi problemi. Il suo studio è tradizionalmente suddiviso in numerosi ambiti che includono la morfologia, la sintassi, la semantica, la pragmatica, ecc...

Questo lavoro si inserisce nell'ambito della morfologia, ovvero quella disciplina che ha per oggetto lo studio della struttura grammaticale delle parole, che ne stabilisce la classificazione e l'appartenenza a determinate categorie, e le forme della flessione, come la coniugazione per i verbi e la declinazione per i sostantivi.

Possiamo identificare in parole e morfemi le unità base della morfologia. Il concetto di parola è difficilmente definibile in maniera univoca, data la grande varietà delle lingue naturali, ma possiamo assumere che per parole si intendano quelle unità del linguaggio umano istintivamente identificate come tali dai parlanti. Il concetto di morfema è, al contrario, intuitivamente meno evidente ma di più facile definizione: consideriamo un morfema come la minima unità grammaticale con significato proprio. I morfemi possono essere liberi o legati: sono morfemi liberi quelli che possono ricorrere da soli in una frase, costituiscono parola a sé e non si devono quindi legare ad altri morfemi per assumere significato (ora, ieri, virtù). Al contrario sono morfemi legati quelli che, per formare una parola e assumere quindi significato, devono "aggiungersi" ad un altro morfema. Parliamo quindi di flessione (canto-avo, mar-e), derivazione (bell-ezza, ri-fare) e composizione (capo-stazione) [15].

In linguistica computazionale la necessità di descrivere e manipolare le strutture morfologiche, sintattiche e semantiche di frasi del linguaggio naturale, in modo da tradurle per essere poi usate da un sistema computerizzato, risulta estremamente rilevante. Restringendo il campo all'ambito morfologico occorre disciplinare l'analisi di sequenze composte da più morfemi.

Come risposta a questa esigenza nasce la morfologia computazionale, che si pone come obiettivo quello di riconoscere stringhe ben formate di caratteri e di metterle in relazione con i morfemi che la compongono. A formare questa relazione sono quindi i concetti di lemma e di forma lessicale, che necessitano di una definizione più accurata. Con lemma intendiamo la forma di citazione

di una parola, ovvero quella che si può trovare sui vocabolari. Per convenzione questa è, per i verbi, la voce all'infinito presente mentre per i sostantivi e gli aggettivi, quando presente, è la forma maschile singolare. Le parole collegate ad ogni lemma vengono chiamate forme lessicali o entrate lessicali e sono tutte le forme che quel lemma può assumere, quindi per i verbi, tutte le coniugazioni e per i sostantivi e gli aggettivi tutte le declinazioni possibili.

La duplicità della relazione che si viene a creare tra forma e lemma, permette alla morfologia computazionale di focalizzarsi su due operazioni fondamentali: la generazione di una o tutte le forme di un lemma e, specularmente, l'analisi di una forma per individuare il relativo lemma e tutte le caratteristiche che l'hanno generata.

Un componente necessario per effettuare questa associazione è chiamato lessico o dizionario e per definirlo è necessario fare una distinzione fra tre diversi livelli di rappresentazione [14]:

- *User definitions*: in questo livello si rappresenta il modo in cui si possono inserire nel sistema le caratteristiche di ogni entrata lessicale, secondo una specifica notazione formale.

- *Stored lexicon*: in questo livello si rappresenta il modo in cui il sistema memorizza le informazioni relative alle entrate lessicali. Questa non è una semplice traduzione in linguaggio macchina ma un'elaborazione che dipende fortemente dall'implementazione scelta.

- *Effective lexicon*: in questo livello è rappresentato il processo di look-up di una parola ed è quindi rappresentata la totalità delle informazioni riguardanti la parola stessa. Questo livello deriva dai due precedenti ma differisce dalla loro rappresentazione in quanto le informazioni sono generate dinamicamente dal processo di look-up.

2.2 I lemmari tabulari

Il primo approccio all'implementazione di un lessico è tramite l'utilizzo di lemmari tabulari. È abbastanza intuitivo infatti pensare di mappare ogni entrata lessicale in una riga di una enorme tabella che rappresenta l'intero lessico.

LEMMA	FORMA	
porta	porta	NOME SING FEMM
porta	porte	NOME PLUR FEMM
portare	porta	VERBO IND PRES 3a SING
		VERBO IMP PRES 2a SING
portare	porti	VERBO IND PRES 2a SING
		VERBO CONG PRES 1a SING
		VERBO CONG PRES 2a SING
tavolo	tavolo	NOME SING MASC
tavolo	tavoli	NOME PLUR MASC

In questo modo sarebbe possibile costruire un dizionario il cui *stored lexicon* verrebbe implementato sfruttando semplicemente le funzionalità di gestione tabelle, disponibili in qualunque linguaggio di programmazione. Seguendo questa idea basterebbero le semplici funzioni *built-in* per le tabelle, per rappresentare i livelli di *user definitions* e di *effective lexicon*. Si può quindi notare che il contenuto linguistico dei tre livelli sarebbe identico, semplificando i processi di trasformazione delle informazioni tra i livelli.

Altrettanto chiari sono però anche i lati negativi di un approccio di questo tipo. Innanzi tutto le informazioni da memorizzare per ogni parola sono così numerose che sarebbe decisamente tedioso per l'utente scriverle in maniera esplicita e dispendioso memorizzarle nello *stored lexicon*. Inoltre fenomeni morfologici comuni a molti termini non vengono gestiti in modo efficiente, ma vengono semplicemente introdotte ripetizioni. Questa situazione sarebbe perciò accettabile solo per sistemi molto piccoli, dove il contenuto del lessico non cambia molto spesso, mentre prendendo in considerazione un'intera lingua questo approccio è impraticabile, se non per piccole porzioni del linguaggio.

2.3 Metodologie basate su FST: Two-level Morphology

Per ovviare ai problemi di una soluzione tabulare si sfruttano fenomeni morfologici comuni a molti termini e parziali regolarità del linguaggio naturale. È infatti facile osservare che le parole nel linguaggio naturale mostrano molte re-

golarità nella loro struttura e nelle loro caratteristiche linguistiche. Possiamo distinguere queste regolarità in regolarità flessionali e regolarità derivazionali.

Le regolarità flessionali presentano le seguenti caratteristiche [14]:

- sistematicità: l'aggiunta di una desinenza ad un morfema ha lo stesso effetto grammaticale per tutti i morfemi di quella categoria (can-i, luc-i, tavol-i ...);

- produttività: nuove parole del linguaggio acquisiscono il comportamento della categoria a cui appartengono;

- preservazione della categoria: le categorie grammaticali non vengono alterate dal processo di flessione.

Le regolarità derivazionali invece si comportano in maniera opposta:

- non sistematicità: l'aggiunta dello stesso affisso a morfemi diversi può avere effetti semantici radicalmente diversi;

- produttività parziale: non è sempre vero che le parole del linguaggio assumono le stesse regolarità derivazionali della categoria a cui appartengono;

- alterazione della categoria: le parole create mediante l'aggiunta di affissi possono appartenere ad una categoria lessicale diversa dalla parola di partenza (dolce - aggettivo, dolce-mente - avverbio).

Viste queste regolarità, si pone ora il problema di avere delle regole composizionali dette morfotattiche, che permettano di creare le parole a partire dai morfemi. In questa operazione è necessario porre notevole attenzione, in quanto ogni affisso o desinenza può essere aggiunto solamente a morfemi di una certa categoria lessicale. La categoria grammaticale di appartenenza della parola ottenuta, inoltre, potrebbe essere la stessa del morfema di partenza oppure potrebbe dipendere dall'affisso aggiunto; ed infine è necessario tenere conto del fatto che entrambi i morfemi che formano la parola portano informazioni grammaticali sulla parola stessa (ad es. cant-avo: il fatto che si tratti del verbo cantare deriva dal primo morfema, mentre il fatto che si tratti della prima persona singolare del modo indicativo, tempo imperfetto deriva dalla desinenza, il secondo morfema).

Capovolgendo il problema, vi è la necessità di avere delle regole che siano in grado di dividere la parola, per poter risalire alle informazioni contenute nei diversi morfemi che la compongono. Questo compito non può essere strutturato come un semplice riconoscimento di un morfema all'interno di una parola. È infatti facile notare che la ricerca di morfemi noti all'interno di una parola potrebbe facilmente portare ad errori concettuali (ad es. all'interno della parola sedia non posso riconoscere -a come una desinenza verbale).

A fianco delle regole morfotattiche sono necessarie delle regole di riaggiustamento, dette grafotattiche, per modificare la forma dei morfemi in base al contesto in cui si trovano e gestire quindi quelle piccole alterazioni che compaiono all'interno di alcune parole (ad es. panc-a al plurale diventa panc-e e

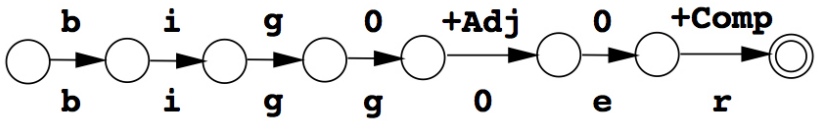

Figura 2.1: Esempio di FST (Finite State Transducer)

ha bisogno dell'inserimento di una h).

Con la consapevolezza che la relazione tra forma e lemma è una relazione di tipo regolare, possiamo definirla con il meta-linguaggio delle espressioni regolari e possiamo compilarla quindi con il formalismo degli automi a stati finiti.

Un trasduttore è una macchina astratta che effettua una traduzione tra sequenze di simboli. Un FST (finite state transducer) si comporta come un semplice automa a stati finiti con, in più, la possibilità di fornire un output durante la scansione dei simboli in input. Questa caratteristica aggiuntiva è ottenuta mantenendo per ogni transizione tra gli stati (arco) due simboli invece di uno solo: uno rappresenta il simbolo di input che permette di effettuare la transizione, l'altro rappresenta il simbolo da fornire in output quando viene effettuata la transizione sull'arco. Una delle due serie di simboli rappresenterà la forma e l'altra serie di simboli, il lemma più le caratteristiche di quell'entrata (vedi figura 2.1). Per indicare queste due serie di simboli si usa la notazione a:b, dove a è il simbolo di input e b è quello di output.

Un vantaggio di questa implementazione è che il trasduttore così creato è inerentemente bidirezionale, non c'è quindi un lato di input privilegiato. Lo

stesso trasduttore può essere utilizzato quindi sia per il task della generazione, che per quello dell'analisi, rovesciando l'interpretazione tra il simbolo di *input* ed il simbolo in *output*.

Insiti nel linguaggio naturale sono presenti casi di ambiguità e implementando lo *stored lexicon* con FST questi risultano implicitamente gestiti: ad una forma lessicale possono infatti essere associati più lemmi (ad es. porta è associato sia al lemma nominale porta, che a quello verbale portare) e ad un lemma possono essere associate più entrate (ad es. la terza persona singolare del verbo vendere è contemporaneamente vendei e vendetti).

Per la gestione delle regole grafotattiche, un approccio decisivo è stato introdotto da Koskenniemi [11], che propose uno schema per la loro gestione basato su un insieme di trasduttori. L'idea alla base del suo approccio consiste nel vedere le due serie di simboli del trasduttore come due livelli - le forme lessicali (lemma più caratteristiche) e le forme flesse (entrate lessicali). Koskenniemi sviluppò delle regole ad alto livello per mettere in relazione i due livelli, che poi trasformò mediante una compilazione in trasduttori a stati finiti. Ogni regola gestisce una regola grafotattica, quindi potremmo avere la regola che inserisce un'h nel caso in cui dobbiamo formare il plurale della parola panc-a:

```
p a n c - e
p a n c h e
```

il primo livello rappresenta il risultato del trasduttore lessicale che ha ac-

costato i morfemi corretti, il secondo livello è il risultato dell'applicazione della regola appena vista (-:h il simbolo - si trasforma nel simbolo h), che entra in azione in presenza del contesto appropriato, cioè in presenza di una *c* o *g*, seguita da una *e* o una *i*.

Queste regole vengono quindi compilate in una serie di trasduttori. Prima dell'affermazione dell'approccio di Koskenniemi, questi trasduttori venivano posti in cascata, in modo che le regole venissero applicate una dopo l'altra. Questa modalità però presenta il difetto di non poter esprimere regole nella forma "e:l subito dopo il contesto in cui l:i". L'introduzione della two-level morphology di Koskenniemi risolve questo problema, ponendo i trasduttori relativi alle regole grafotattiche in parallelo, in modo che vengano applicati contemporaneamente. In questo modo non solo si possono applicare regole come quella illustrata prima, ma si ha la sicurezza che un'entrata lessicale viene riconosciuta se e solo se rispetta tutte le regole contemporaneamente.

2.4 Panoramica sui trasduttori

La prima implementazione della two-level morphology risale al 1983 [11], ma ad oggi sono molti i tentativi di miglioramento che sono stati prodotti. In questa sezione verranno analizzati i pacchetti software presi in considerazione per questo lavoro e verranno specificati i motivi che hanno determinato la scelta del più adatto.

L'implementazione del modello di Koskenniemi più importante è quella

di Karttunen, che ha portato allo sviluppo del software XFST della Xerox e dal quale sono state tratte moltissime copie e varianti. XFST è il primo software che è stato preso in considerazione per lo sviluppo di questa tesi, perché presenta caratteristiche che lo rendono uno dei migliori software di morfologia computazionale presenti in questo momento. Nel paragrafo 2.4.1 ne verrà illustrata la storia ed il funzionamento.

Un altro software che è stato analizzato e valutato è PC-KIMMO, applicazione coperta da copyright ma liberamente distribuibile. PC-KIMMO viene spesso utilizzato in ambito universitario proprio per la sua disponibilità, e la sua relativa semplicità d'uso e verrà illustrato nel paragrafo 2.4.2.

Infine si è analizzata una recente implementazione open source del modello di Koskenniemi, HFST, sviluppata seguendo lo stile di XFST, dal dipartimento di linguistica dell'università di Helsinki nel 2009 (par. 3.1).

2.4.1 XFST

Come già accennato, XFST rappresenta il metro di paragone per tutte le applicazioni nel campo dell'analisi morfologica ed è stata sviluppata all'interno della Xerox Corporation, società che tuttora è proprietaria e detiene i diritti del software. Le due componenti su cui si basa sono un compilatore di lessico, lexc, ed un compilatore di regole, twolc.

Lexc prende in input un file testuale in cui le entrate del lessico possono esprimere una relazione tra due forme, caratteristica principale che lo differenzia dagli altri approcci (ad es. PC-KIMMO, paragrafo 2.4.2). Questo

approccio al compilatore del lessico è basato sull'implementazione di un'applicazione simile, scritta in Common Lisp da Lauri Karttunen nel 1990 e formalizzata dallo stesso Karttunen con Todd Yampol [7].

La struttura di un file di input di lexc è caratterizzata dalla dichiarazione di caratteri multi-simbolo e di un lessico principale (Root) e sottolessici. Questi ultimi rappresentano intuitivamente gli stati che devono essere attraversati per riconoscere o generare una forma. Di seguito viene mostrato un semplice esempio per forme nominali:

```
LEXICON Root

    Nomi;

LEXICON Nomi

    bambin SuffNomi;
    gatt SuNomi;

LEXICON SuffNomi

    a #;
    e #;
    o #;
    i #;
```

In questo esempio si può vedere l'idea generale di costruzione del lessico: la sezione LEXICON Nomi indica che, trovandosi nello stato "Nomi", è possibile passare nello stato "SuffNomi", solo dopo aver riconosciuto la stringa

"bambin" o la stringa "gatt". Con questo automa sono quindi riconosciute tutte le forme dei lemmi gatto e bambino. L'output di lexc è, quindi, un automa a stati finiti che può essere composto con un trasduttore di regole a due livelli, prodotto dal programma twolc.

La prima versione di twolc, che implementava l'idea della two-level morphology di Koskenniemi [11], venne scritta in Interlisp da Lauri Karttunen, Kimmo Koskenniemi e Ronald M. Kaplan [5]. L'attuale versione, scritta in linguaggio C, è invece basata sull'implementazione in Common Lisp di Karttunen, ed è stata formalizzata da Karttunen stesso con Todd Yampol, Kenneth R. Beesley (Microlytics) e Ronald M. Kaplan [6].

Twolc prende in input un file che contiene le regole grafotattiche (scritte in base al modello two-level) relative al linguaggio considerato. Il trasduttore in output viene composto a quello relativo al lessico per ottenere un automa completo che riconosce e genera le parole del linguaggio.

I comandi di XFST sono divisi in tre gruppi generali: Input/Output, Operations e Display. Esistono tre comandi che possono essere considerati principali: compile-source crea l'automa contenente il lessico; read-rules permette di leggere i trasduttori delle regole grafotattiche, create da twolc; compose-result, infine, compone l'automa del lessico con l'automa delle regole per ottenere l'automa finale.

La generazione di una forma, tramite XFST, avviene in due passaggi: il lemma con le entrate lessicali relative alla forma da generare (panca+F+P) viene passato all'automa del lessico, che restituisce la forma intermedia (panc-

e); questa poi viene passata all'automa delle regole, che applica contemporaneamente quelle regole il cui contesto è rispettato e restituisce la forma finale (panche).

2.4.2 PC-KIMMO

PC-KIMMO è un parser morfologico basato sulla two-level morphology e sulle specifiche dell'implementazione del suo modello, Kimmo, descritto da Karttunen [4]. La prima versione rilasciata risale ai primi anni '90 [1] e riveste una notevole importanza in quanto rappresenta la prima implementazione del modello di Koskenniemi ad essere liberamente distribuita. Fino alla sua nascita infatti esistevano numerose applicazioni che implementavano questo modello, ma tutte erano disponibili solamente all'interno di grandi centri accademici di ricerca. La versione qui presa in considerazione è la 2.0 [2], risalente al 1995, in cui sono state introdotte alcune modifiche, tra cui la più rilevante è l'aggiunta del file grammar di cui si parlerà in seguito.

La struttura di un linguaggio sviluppato con PC-KIMMO si basa su tre file principali: rules (.rul), lexicon (.lex) e grammar (.grm).

A dispetto del nome, lo scopo principale del file .rul non è quello di specificare delle regole, quanto piuttosto quello di fornire una lista di elementi associati a delle parole chiave. Più specificatamente, nella prima parte del file rules vengono definiti un alfabeto con eventuali sottoinsiemi e una serie di caratteri speciali: il carattere null, quello any e il carattere limitatore di fine parola. Vengono poi definite delle semplici regole che specificano, im-

maginando ogni carattere dell'alfabeto come nodo facente parte di un grafo, quali nodi sono fra loro connessi. Questa seconda parte è sostanzialmente un "residuo" della prima versione di PC-KIMMO. Qui infatti erano descritte le regole che, nella versione analizzata (ver. 2), sono contenute nel file grammar.

Esistono più file di tipo lexicon, che contengono il lessico della lingua presa in considerazione, cioè i morfemi che PC-KIMMO può riconoscere. In quello principale vengono inseriti tutti i morfemi principali (stem), mentre nei file secondari sono contenute le desinenze flessive e gli affissi. In particolare la struttura base del file lexicon principale è:

```
ALTERNATION (nome alternation) (lista sublexicon)
FEATURES (lista features)
FIELDCODE (lexical item code) U
FIELDCODE (sublexicon code) L
FIELDCODE (alternation code) A
FIELDCODE (features code) F
FIELDCODE (gloss item code) G
INCLUDE (path file .lex secondari)
END
```

Alternation serve a definire con un unico nome un insieme di tipi lessicali, ad esempio se si vuole definire un elemento Root che può essere un sostantivo od un verbo, è sufficiente scrivere ALTERNATION Root N V. Features definisce una lista di parole chiave che saranno utilizzate nei file lexicon secondari

e quindi nella grammatica. I "Fieldcode X code" (dove X sta per lexical item, sublexicon, ecc...) sono usati per definire quali codici saranno usati per marcare ogni tipo di campo nella lexical entry. Include, infine, serve ad includere i file lexicon che contengono solamente una lista di lexical entry nella forma:

```
\(lexical item code) (lexical item)
\(sublexicon code) (sublexicon item)
\(alternation code) (alternation item)
\(features code) (features list)
\(gloss code) (gloss string)
```

Tutti i code sono già stati definiti nei campi fieldcode del file lexicon principale, il lexical item è il morfema vero e proprio, mentre il sublexicon indica il tipo di morfema (ad es. sostantivo, verbo, suffisso...). Le features indicano ulteriori proprietà del morfema che serviranno nella grammatica, mentre gloss fornisce ulteriori indicazioni a seconda del tipo di morfema (ad es. se il morfema è il suffisso di un verbo, viene indicato a quale tempo e persona questo suffisso corrisponde). Per chiarire le idee viene mostrato un esempio di morfema:

```
/lf autist
/lx N
/alt Suffix
/fea nm ng1
```

```
/gl autista
```

Questo esempio illustra come "autist" (\lf autist) è la radice di un sostantivo (\lx N) maschile ed appartiene al gruppo dei sostantivi che finiscono per -a (\fea nm ng1) e rappresenta la parola autista (\gl autista). Alternation (\alt Suffix) infine va inteso come risposta alla domanda "quale gruppo di morfemi si può combinare a destra di questo morfema?": in questo caso la risposta è un suffisso.

Infine, nel file grammar, inserito nella seconda versione del programma, sono contenute tutte le regole morfotattiche espresse con un paradigma context-free. Le regole che compongono la prima parte di questo file servono per trasformare le lexical entry dei file .lex in record chiamati "feature structure". Le regole che compongono la seconda e ultima parte del file sono regole morfotattiche, servono perciò ad istruire PC-KIMMO su come combinare i morfemi tra di loro. La struttura di queste regole viene illustrata con i seguenti esempi:

```
Word = PREFISSO Stem
Stem = V SUFFIX
Stem = N
```

2.4.3 La scelta operata

XFST è sicuramente un ottimo prodotto e soddisfa tutti i requisiti tecnici per lo sviluppo di un tool per l'analisi morfologica della lingua italiana. Quest'ap-

plicazione però, come già accennato nella breve introduzione precedente, è un software proprietario della Xerox Corporation e dunque i suoi contenuti non sono accessibili liberamente. È dunque impossibile l'utilizzo di XFST per lo sviluppo di un tool open source come quello che si vuole creare in questo lavoro.

PC-KIMMO invece presenta alcuni difetti di natura tecnica. Innanzitutto è impossibile la gestione delle lettere accentate e, data la natura della lingua italiana, una soluzione sviluppata con questo programma risulterebbe complicata e inelegante, sia dal punto di vista implementativo, che soprattutto da quello dell'utente finale.

Inoltre PC-KIMMO non da la possibilità di comporre più trasduttori e questa limitazione ostacolerebbe un'eventuale aggiunta di features al tool principale (vedi cap. 5).

Infine un altro punto a sfavore di PC-KIMMO è rappresentato dal fatto che questo è un software abbastanza datato. L'ultima versione aggiornata infatti risale al 1995 e si è quindi pensato di puntare su un software più giovane e in evoluzione.

Alla luce di queste considerazioni, si è dunque pensato di adottare HFST, che verrà ampiamente presentato nel capitolo successivo. Questo è infatti un software open source sviluppato recentemente e ancora in fase di evoluzione, e che inoltre fornisce le funzionalità che PC-KIMMO non permette di avere.

Capitolo 3

Anita

In questo capitolo verrà descritta la fase di realizzazione di AnIta. In una prima sezione si analizzerà il toolkit HFST, utilizato per lo sviluppo dell'analizzatore/generatore. Nella seconda sezione invece saranno illustrati i dettagli implementativi del progetto.

3.1 HFST

Il software HFST è stato sviluppato nel 2009 dal team di Krister Lindén nel Dipartimento di General Linguistics dell'Università di Helsinki [12]. È un'applicazione open source rilasciata con licenza GNU lesser general public license. La decisione di rilasciare un software open source deriva dal desiderio di coinvolgere il maggior numero possibile di persone nel progetto, e dalla volontà di rendere possibile lo sviluppo di analizzatori per il maggior numero

di lingue possibile.

HFST è stato implementato per fornire gli strumenti di base per sviluppare in maniera efficiente, compilare ed eseguire analizzatori morfologici. Con questo scopo è stata creata un API (Application Programming Interface) unificata, in grado di interfacciare varie librerie di trasduttori a stati finiti e che permette di incorporare ulteriori librerie al momento del bisogno. Al di sopra di questa interfaccia, sono poi stati creati dei tool di base, quali HFST-TWOLC, HFST-LEXC ed altri, che permettono l'effettiva scrittura di analizzatori e generatori morfologici.

Nei prossimi paragrafi verranno illustrati uno per uno gli strumenti più importanti che fanno parte di HFST.

3.1.1 HFST-LEXC

HFST-LEXC è il compilatore del lessico ed è implementato prendendo come modello, teorico e pratico, lexc di XFST. Prende quindi in input un file contenente il lessico e fornisce in output il trasduttore che genera le parole del lessico stesso.

Il file di input di HFST-LEXC è suddiviso in due sezioni: la definizione dei simboli multicarattere ed i lessici.

Nella prima sezione sono, quindi, dichiarati tutti i simboli costituiti da più di un carattere, che il compilatore deve considerare come un unico simbolo. Per meglio identificare questi simboli, essi sono solitamente espressi in maiuscolo e preceduti da un carattere speciale (ad es. +, ^, ~). Altri carat-

teri speciali sono il %, carattere di *escape* che permette di utilizzare simboli protetti (utilizzato nel file .twolc), e il #, utilizzato come *word boundary*. La dichiarazione di simboli multicarattere è una lista il cui separatore è lo spazio, che a sua volta è quindi un carattere protetto. In questa sezione è possibile dichiarare simboli formati da un numero arbitrario di caratteri, ma sono sconsigliate (e anche inutili) dichiarazioni di simboli di un solo carattere. L'algoritmo che gestisce possibili casi di ambiguità tra questi simboli multicarattere riconosce la stringa più lunga da sinistra a destra, quindi, esemplificando, in presenza dei simboli `ab bc cd de`, la stringa `abcde` diventa `ab cd e`.

```
Multichar_Symbols

    +NOME +SING +PLUR +FEMM +MASC
```

La seconda sezione del file consiste nella dichiarazione dei lessici ovvero di tutti i morfemi della lingua. Ogni *lexicon entry* consta di tre elementi: una coppia lemma:forma, una classe di continuazione ed il simbolo ; che ha la funzione di terminatore. La coppia lemma:forma rappresenta le due etichette degli archi dell'automa trasduttore risultante e il carattere separatore è, come mostrato, il simbolo :. Per classe di continuazione si intende il lessico in cui procede il riconoscimento della stringa. Esso rappresenta lo stato dell'automa in cui andare una volta riconosciuta la coppia lemma:forma. Nel caso il riconoscimento della stringa sia terminato la classe di continuazione viene sostituita dal carattere word boundary #.

```
LEXICON Root

    bambino+NOME:bambin SuffNomi;

    gatto+NOME:gatt SuffNomi;

    film+NOME+MASC+SING:film # ;

LEXICON SuffNomi

    +FEMM+SING:a #;

    +FEMM+PLUR:e #;

    +MASC+SING:o #;

    +MASC+PLUR:i #;
```

La compilazione di un file del lessico tramite HFST-LEXC, produce un albero di lessici, in cui i figli di ogni radice descrivono ogni possibile continuazione per le forme appartenenti a quel lessico. Considerando l'esempio precedente, la compilazione produrrà un trasduttore che, in corrispondenza di `gatto+NOME+MASC+PLUR`, restituirà la forma `gatti`.

3.1.2 HFST-TWOLC

HFST-TWOLC è il compilatore delle regole ed è implementato seguendo il modello di twolc di XFST. Prende, quindi, in input un file contenente le regole da applicare e restituisce in output il trasduttore completo delle regole grafotattiche.

Il file di input di HFST-TWOLC è suddiviso in una prima parte di definizioni e in una seconda contenente la dichiarazione delle regole grafotattiche.

La prima definizione contenuta nel file è quella dell'alfabeto, che elenca tutti i simboli usati nella grammatica. Occorre, quindi, inserire tutte le lettere dell'alfabeto della lingua analizzata, comprese le vocali accentate, se presenti, seguite da tutti i simboli multicarattere definiti nel file di HFST-LEXC. Da notare che Hfst richiede la codifica dei caratteri UTF-8, ciò permette di esprimere anche lettere accentate e simboli particolari. In questa sezione è necessario utilizzare il carattere protetto di *escape* %, per poter dichiarare simboli contenenti caratteri protetti. Già in questa sezione è possibile specificare il comportamento di alcuni simboli, tramite la consueta modalità a:b, che, però, viene presa in considerazione solo nel caso in cui il simbolo non venga modificato all'interno di una qualche regola.

```
Alphabet

    a b c ... x y z à è é ì ò ù
    %+NOME:0 %+SING:0 %+PLUR:0 %+FEMM:0 %+MASC:0 ;
```

Nell'esempio sopra notiamo come i simboli utilizzati per descrivere le caratteristiche delle forme, vengano eliminati perché non compaiano nella generazione di una forma relativa ad un lemma.

Dopo la definizione dell'alfabeto vi è l'eventuale dichiarazione di insiemi di simboli, nelle sezioni denominate Sets e Definitions. Questi insiemi possono essere definiti utilizzando anche il formalismo delle espressioni regolari e risultano utili per rendere più semplice l'espressione del contesto di alcune regole.

```
Sets

    Vocali = a e i o u ;

Definitions

    NonVocaliSeq = [ \:Vocali ]* ;
```

La seconda parte del file è caratterizzata dalla definizione delle regole grafotattiche. Queste regole, espresse secondo il modello a due livelli, consistono di un centro, un operatore ed un contesto. Il centro contiene la modifica di uno o più simboli, espresse nella consueta forma a:b. L'operatore solitamente utilizzato è <=> che indica la biunivocità della regola, sono però ammessi anche gli operatori unidirezionali => e <=. Vi è inoltre un operatore, /<=, che proibisce l'esecuzione del centro in presenza di un certo contesto. Il contesto è una coppia di espressioni regolari separate dal carattere _, che indica la posizione in cui deve agire il centro della regola.

```
Rules

    0:h <=> g _ [ e | i ] ;
```

Questa semplice regola inserisce una h tra una g ed una tra le due vocali indicate.

Infine HFST-TWOLC permette la scrittura di regole che contengono variabili, che possono essere dichiarate in una sezione del file apposita, oppure inserendo una clausola `where`, subito dopo il contesto della regola in cui appaiono.

3.1.3 Altri tools per la gestione dei trasduttori

Hfst-Compose-Intersect

Il lessico compilato con Hfst-LexC e le regole grafotattiche compilate con Hfst-TwolC, vengono combinate insieme tramite Hfst-Compose-Intersect. Questo tool prende, quindi, in input il trasduttore del lessico, il trasduttore delle regole e restituisce in output un trasduttore che permette di risolvere il task della generazione della forma a partire dal lemma.

Hfst-Compose-Intersect è l'implementazione dell'algoritmo di composizione - intersezione di Karttunen [8], per cui il risultato dell'operazione è equivalente alla composizione del trasduttore del lessico, con l'intersezione dei trasduttori delle regole. Karttunen ha osservato che l'esecuzione del solo algoritmo di intersezione può produrre un risultato di dimensioni spropositate, in tempi decisamente troppo elevati; mentre la contemporanea composizione con il lessico, permette di restringere l'intersezione dei trasduttori delle regole, riducendo sensibilmente i tempi e le dimensioni del risultato.

Hfst-Lookup

Hfst-Lookup prende in input un trasduttore e permette di interrogare il trasduttore stesso per ottenere le informazioni da esso generate. L'interrogazione è fatta tramite linea di comando ma è possibile passare in input un file testuale contenente le interrogazioni. Utilizzando il tool da riga di comando per terminarlo occorre immettere la stringa *exit*.

```
$ hfst-lookup nome_analizzatore.hfst
> bambine
bambine bambino+NOME+FEMM+PLUR
```

Tools per operazioni algebriche

HFST consente di effettuare operazioni algebriche tra trasduttori. Le operazioni più interessanti sono:

- HFST-INVERT: inverte le etichette degli archi del trasduttore, per cui invertendo un generatore, si ottiene un analizzatore e viceversa.

- HFST-COMPOSE: effettua una composizione tra due trasduttori, per cui l'input del risultante rappresenta l'input del primo trasduttore, il cui output ha la funzione di input per il secondo. Il risultato finale sarà quindi l'output del secondo trasduttore.

- HFST-CONCATENATE: concatena due trasduttori, per cui il trasduttore risultante riconoscerà le stesse stringhe dei due trasduttori di partenza concatenate tra loro e restituirà l'output concatenato di entrambi (ad es. se concateniamo due trasduttori uguali, dovremo interrogare il risultante con una stringa duplicata - `canecane` - ed otterremo `cane+NOME+MASC+SING cane+NOME+MASC+SING`). Risulta molto interessante quando si ha a che fare con trasduttori che analizzano aspetti diversi della stessa lingua.

- HFST-CONJUNCT: opera la congiunzione logica di due trasduttori. Nel risultante trasduttore verranno riconosciute solo le stringhe riconosciute da entrambi i trasduttori coinvolti nell'operazione.

- HFST-DISJUNCT: effettua la disgiunzione logic tra due trasduttori. Nel risultante trasduttore verranno riconosciute le stringhe riconosciute da almeno uno dei due trasduttori coinvolti nell'operazione.

Tools per il testing

HFST mette, inoltre, a disposizione alcuni tools per le fasi di testing dei trasduttori:

- HFST-FST2STRINGS: restituisce in output la lista di tutte le stringhe riconosciute dal trasduttore in input.

- HFST-COMPARE: mette a confronto i due trasduttori presi in input, per verificarne l'uguaglianza.

- HFST-SUMMARIZE: fornisce in output le caratteristiche principali del trasduttore in input. Esempi di queste caratteristiche sono: numero di stati, numero di stati finali, numero di archi, e così via.

3.2 Descrizione del progetto

In questo capitolo verranno descritte in dettaglio le caratteristiche di questo progetto. AnIta (ANalizzatore ITAliano) è un analizzatore-generatore mor-

fologico. Offre infatti all'utente, la possibilità di interrogare il sistema per ottenere una forma lessicale a partire dal lemma, tramite la sua componente generativa ed il lemma a partire dalla forma, tramite quella analizzativa.

Entrando nel dettaglio, AnIta è costituito principalmente da due moduli: italiano.lexc e italiano.twolc, che verranno illustrati e spiegati in questo capitolo. L'analisi procederà prendendo in considerazione questi due moduli, in parallelo con l'analisi dei casi della lingua italiana, che sono stati studiati e poi implementati per la loro creazione.

Una volta che questi file sono stati compilati tramite i tool appropriati (HFST-LEXC e HFST-TWOLC), verranno composti tramite il tool HFST-COMPOSE-INTERSECT, che permette di ottenere il trasduttore che rappresenta la componente generativa di AnIta. Per ottenere poi l'analizzatore, si dovrà utilizzare il tool HFST-INVERT, che fornirà in output la componente analizzativa di AnIta.

Per poter gestire ed usufruire comodamente di queste due componenti, è stata scritta un'applicazione in linguaggio bash, che svolge la funzione di intermediario tra l'utente ed AnIta, prendendo in input le richieste dell'utente, richiamando l'analizzatore o il generatore a seconda dei casi e restituendo il loro output all'utente.

Questa applicazione mostra all'utente un menu testuale in cui è possibile scegliere se utilizzare il generatore o l'analizzatore:

```
Benvenuto su AnIta
Digita A per l'analizzatore
```

```
Digita G per il generatore
Digita EXIT per uscire
```

Per utilizzare il generatore, l'utente dovrà inserire la richiesta nella forma:

```
l_mangiare+V_FIN+IND+PAST+1+SING
```

Si otterrà quindi come risposta:

```
mangiai
```

Speculare risulta il caso di utilizzo dell'analizzatore, in cui l'utente dovrà inserire la seconda stringa mostrata (`mangiai`) ed otterrà come risultato la prima stringa (`l_mangiare+V_FIN+IND+PAST+1+SING`).

3.2.1 Il lessico

Come è stato detto precedentemente, per sviluppare un trasduttore morfologico tramite i tool di HFST, è necessario scrivere il lessico della lingua presa in analisi, in un file testuale, da compilare in seguito con HFST-LEXC. In questo paragrafo verrà illustrato in dettaglio il lessico contenuto nel file italiano.lexc.

Multichar Symbols

La prima parte del file è caratterizzata dalla dichiarazione dei simboli multicarattere:

```
Multichar_Symbols

    +NN +NN_P
    +ADJ +ADJ_DIM +ADJ_IND +ADJ_IES +ADJ_NUM +ADJ_POS
    +PRON +PRON_P +PRON_DIM +PRON_IND +PRON_IES
    +PRON_REL +PRON_POS +PRON_PER
    +ADV +ART +INT +PREP +PREP_A +CONJ +CONJ_S
    +MASC +FEMM +SING +PLUR
    +DIM +AUM +VEZ +PEG +SUP +INF
    +V_ARE +V_ERE +V_IRE +V_FIN +V_NOFIN +V_PP
    +IND +SUBJ +COND +IMP +PART +GER
    +PRES +IMPERF +PAST +FUT
    +1 +2 +3
    +GLI +VEL_A +VEL_Y
```

Come si può notare, sono state dichiarate tutte le possibili *feature* dei morfemi della lingua italiana. È stato utilizzato il carattere + come delimitatore iniziale, per evitare di creare ambiguità con eventuali parole del linguaggio ed è stato utilizzato, per convenzione, il carattere _ per aggiungere informazione alla *feature*.

I primi simboli dichiarati rappresentano le categorie lessicali delle parole della lingua italiana. Possiamo identificare le parti variabili del discorso: i nomi, comuni (+NN) e propri (+NN_P), i verbi (+V), tutti i tipi di aggettivi (+ADJ) e di pronomi (+PRON) e gli articoli (+ART); e le parti del discorso

dette invariabili: gli avverbi (+ADV), le preposizioni (+PREP), le congiunzioni (+CONJ) e le interiezioni (+INT).

Per i verbi, in particolare, oltre ad identificare le tre coniugazioni (+V_ARE, +V_ERE, +V_IRE), vi è un'ulteriore divisione in verbo finito (+V_FF), che comprende tutti i modi le cui desinenze definiscono sempre una persona, in verbo non finito (+V_NOFIN) - modi infinito e gerundio - ed in verbo al participio (+V_PP). Altri simboli legati ai verbi sono quelli relativi ai modi (+IND, +SUBJ, ...), i tempi (+PRES, +IMPERF, ...) e le persone (+1, +2, +3) di tutte le forme verbali.

Ulteriori informazioni relative a più categorie lessicali sono il genere (+MASC, +FEMM), il numero (+SING, +PLUR) ed i valutativi (+DIM, +VEZ, ...).

Infine, sono presenti tre simboli multicarattere, che non vengono utilizzati per dare informazione all'utente, ma per indicare alcuni aggiustamenti morfologici da applicare in seguito e di cui si parlerà nel paragrafo 3.2.2.

Descrizione del lemmario

La seconda parte di italiano.lexc, è caratterizzata dalla definizione di tutti i lessici in cui è stata strutturata la lingua italiana.

Il lemmario di questo progetto consta di più di 120.000 lemmi e copre gran parte della lingua italiana. I lemmi presenti nel file del lessico sono stati ricavati da un dizionario disponibile al momento della stesura di questa tesi, all'interno del Dipartimento di Studi Linguistici e Orientali dell'Università di Bologna. La prima parte del lavoro che è stato svolto per lo svolgimento

di questo progetto, è stata, quindi, quella di trasformare questo lemmario, per portarlo da un semplice elenco di lemmi con le relative caratteristiche, ad una serie di *lexicon entry*, adatte alla struttura ed al formato del progetto in sviluppo. Non verranno qui illustrati i tediosi dettagli della conversione, ma ci si soffermerà sulla struttura ottenuta, nei paragrafi seguenti.

Il lemmario così ottenuto è contenuto nel lessico `root` che consideriamo come principale. Questo, quindi, non è altro che un elenco di coppie lemma:forma, in cui sono presenti i caratteri speciali : per dividere il lemma e le caratteristiche relative dalla forma, il carattere # come *boundary* ed il simbolo ; come terminatore di riga. Si è assunta, inoltre, la convenzione di prefissare ai lemmi i caratteri `l_`, in modo da meglio distinguerli dalle forme lessicali, durante le interrogazioni al nostro sistema.

Una prima distinzione che può essere fatta all'interno del lessico è tra morfemi liberi e morfemi legati. Circa un sesto del lemmario è, infatti, composto da morfemi liberi, cioè morfemi che possono ricorrere da soli in una frase. A questa categoria appartengono, ad esempio, i nomi propri e le parti invariabili del discorso.

Nel lemmario queste parole vengono, quindi, gestite completamente all'interno del lessico `root`, senza la necessità di utilizzo di classi di continuità.

```
l_cribbio+INT:cribbio #;
l_laura+NN_P:laura #;
l_perché+CONJ_S:perché #;
l_perché+ADV:perché #;
```

```
l_pop_art+NN+FEMM:pop_art #;
l_sempre+ADV:sempre #;
```

Si può notare, in questi casi, l'assenza della classe di continuazione ed il fatto che non verranno applicate regole di riaggiustamento, in quanto si tratta di morfemi non componibili tra loro.

Di questa categoria fanno, inoltre, parte, pur non essendo morfemi liberi, alcune forme verbali o addirittura interi verbi, che risultano altamente irregolari. La gestione di questi casi potrebbe avvenire nel modo consueto, cioè mediante la creazione della forma tramite la classe di continuazione appropriata, ma ci troviamo di fronte a situazioni che risultano, nella maggior parte dei casi, uniche o quasi; la creazione, quindi, di regole di formazione apposite appesantirebbe inutilmente il lemmario. Per questo motivo, si è scelto di gestire tutte le desinenze di questi verbi in maniera manuale, inserendo completamente le loro forme verbali nel lessico `root`.

```
l_andare+V_FIN+IND+PRES+1+SING:vado #;
l_andare+V_FIN+IND+PAST+1+SING:andai #;
l_andare+V_PP+PART+PRES+SING:andante #;
l_andare+V_NOFIN+INF+PRES:andare #;
l_andare+V_FIN+IND+PAST+3+PLUR:andarono #;
l_andare+V_FIN+SUBJ+IMPERF+3+SING:andasse #;
l_andare+V_FIN+SUBJ+IMPERF+1+PLUR:andassimo #;
l_andare+V_FIN+IND+PAST+2+PLUR:andaste #;
```

```
l_andare+V_FIN+SUBJ+IMPERF+2+PLUR:andaste #;
l_andare+V_FIN+IND+PAST+2+SING:andasti #;
l_andare+V_NOFIN+GER+PRES:andando #;
l_andare+V_FIN+COND+PRES+3+PLUR:andrebbero #;
...
```

Per la gestione dei morfemi legati, ovvero quei morfemi che, per poter comparire all'interno di una frase, hanno bisogno di legarsi ad altri morfemi, si sfrutta la seguente struttura:

```
lemma+features:morfema+features continuation-class ;
```

Alcuni esempi di morfemi legati sono libr-, -i, salt-, -iamo. Nella gestione di questi casi, le *feature* che vengono specificate, sono quelle derivabili a partire dal solo lemma e che non vengono, quindi, modificate dal processo di flessione. Possiamo avere, perciò, casi come quello di alcuni morfemi nominali, in cui le *feature* sono tutte fornite a questo livello, tranne il numero, o esempi come quello di alcuni morfemi aggettivali, per cui non è specificata nessuna *feature*, in quanto alcuni casi di derivazione (-issimamente) ne possono modificare anche la categoria lessicale.

```
l_aratro+NN+MASC:aratr+NN+MASC SufNomMasc0;
l_dormire:dorm+V_IRE SufVerbIre;
l_veloce:veloc SufAggE;
```

Come possiamo notare dall'esempio sopra riportato, anche a fianco del morfema sono presenti le informazioni lessicali, diversamente da quanto visto

nella gestione dei morfemi liberi. Queste sono specificate in modo da poter essere utilizzate nei processi di riaggiustamento effettuati dal trasduttore delle regole. Non tutte le informazioni che vengono specificate, sono poi necessariamente coinvolte nelle regole, ma si è comunque scelto di fornirne il maggior numero possibile, per darne la disponibilità, nel caso si vogliano, in seguito, aggiungere nuove regole.

Alle *feature* indicate, segue la classe di continuazione, la cui utilità è quella di stabilire secondo quali regole debbano comporsi i morfemi base con le desinenze flessive, per creare le forme lessicali. Possiamo individuare tre categorie principali in cui sono state divise le classi di continuazione: quelle riguardanti i verbi, quelle riguardanti i sostantivi e quelle relative agli aggettivi. Queste classi non sono strettamente legate alle categorie lessicali da cui prendono il nome, ma permettono di generare l'intera lingua. Si potrà, infatti, notare che alcune parti del discorso, come i pronomi seguono un comportamento dettato da queste classi. Di seguito verranno analizzate in dettaglio le caratteristiche di queste classi di continuazione.

Classi di continuazione verbali

I verbi regolari italiani seguono tre diverse coniugazioni, sono infatti suddivisi in verbi il cui lemma termina in -ARE, in -ERE ed in -IRE. Tutti i verbi appartenenti ad una stessa coniugazione si comportano seguendo il medesimo schema, per ogni modo, tempo e persona.

Per gestire, quindi, questi verbi regolari sono stati implementati tre lessici,

uno per ogni coniugazione, in cui sono elencate tutte le desinenze per tutti i modi e i tempi verbali e per tutte le persone. In ogni entrata lessicale di questi lessici, vengono fornite le informazioni riguardanti la forma verbale che si viene a creare, sottoforma di simboli multicarattere, accostati sia a destra che a sinistra del ":".

Ogni verbo regolare avrà, quindi, un'entrata nel lessico `root`:

```
l_saltare:salt+V_ARE SufVerbAre;
l_vendere:vend+V_ERE SufVerbEre;
l_dormire:dorm+V_IRE SufVerbIre;
```

La flessione del verbo, poi, avverrà tramite i lessici `SufVerbAre`, `SufVerbEre`, `SufVerbIre`, di cui viene mostrato uno stralcio a titolo di esempio:

```
LEXICON SufVerbAre

    ...
    +V_FIN+IND+IMPERF+1+SING:+IND+IMPERF+1+SING*avo #;
    +V_FIN+IND+IMPERF+2+SING:+IND+IMPERF+2+SING*avi #;
    +V_FIN+IND+IMPERF+3+SING:+IND+IMPERF+3+SING*ava #;
    +V_FIN+IND+IMPERF+1+PLUR:+IND+IMPERF+1+PLUR*avamo #;
    +V_FIN+IND+IMPERF+2+PLUR:+IND+IMPERF+2+PLUR*avate #;
    +V_FIN+IND+IMPERF+3+PLUR:+IND+IMPERF+3+PLUR*avano #;
    ...

LEXICON SufVerbEre
```

```
    ...
    +V_FIN+SUBJ+IMPERF+1+SING:+SUBJ+IMPERF+1+SING*essi #;
    +V_FIN+SUBJ+IMPERF+2+SING:+SUBJ+IMPERF+2+SING*essi #;
    +V_FIN+SUBJ+IMPERF+3+SING:+SUBJ+IMPERF+3+SING*esse #;
    +V_FIN+SUBJ+IMPERF+1+PLUR:+SUBJ+IMPERF+1+PLUR*essimo #;
    +V_FIN+SUBJ+IMPERF+2+PLUR:+SUBJ+IMPERF+2+PLUR*este #;
    +V_FIN+SUBJ+IMPERF+3+PLUR:+SUBJ+IMPERF+3+PLUR*essero #;
    ...

LEXICON SufVerbIre

    ...
    +V_FIN+COND+PRES+1+SING:+COND+PRES+1+SING*irei #;
    +V_FIN+COND+PRES+2+SING:+COND+PRES+2+SING*iresti #;
    +V_FIN+COND+PRES+3+SING:+COND+PRES+3+SING*irebbe #;
    +V_FIN+COND+PRES+1+PLUR:+COND+PRES+1+PLUR*iremmo #;
    +V_FIN+COND+PRES+2+PLUR:+COND+PRES+2+PLUR*ireste #;
    +V_FIN+COND+PRES+3+PLUR:+COND+PRES+3+PLUR*irebbero #;
    ...
```

Da notare l'uso del carattere * al posto del simbolo flessivo - usato solitamente in letteratura. Nel lessico sono infatti presenti parole che contengono questo simbolo (ad es. nord-occidentale) per cui si è preferito utilizzare, in questa fase implementativa, un carattere differente.

Un approccio diverso è necessario per gestire, invece, i verbi irregolari. Come già spiegato nel capitolo riguardante la morfologia computazionale (par. 2.1), anche le irregolarità di una lingua spesso seguono un comportamento regolare. La lingua italiana non fa eccezione e, per lo sviluppo di questo progetto, è stata utilizzata la schematizzazione che viene proposta nell'articolo di Pirrelli e Battista [13].

Come viene illustrato in questo articolo, nella lingua italiana sono presenti circa duecento verbi irregolari (la maggior parte appartengono alla seconda coniugazione) e di questi, solamente una decina (quattro della prima coniugazione e sei tra la seconda e la terza coniugazione) non rispettano lo schema proposto. Secondo gli autori, le irregolarità dei verbi della lingua italiana non affliggono le desinenze, ma bensì le radici delle forme verbali. Ogni verbo irregolare, quindi, mostra l'alternarsi di diverse radici (fino a sei diverse) e lo schema presentato in questo articolo illustra la regolarità con cui queste radici variano all'interno del paradigma del verbo.

Da notare che questo schema rende ininfluente la coniugazione di un verbo, in quanto la modifica riguarda la radice e non la desinenza.

	ind. pres.	ind. imp.	ind. pass.	ind. fut.
1s	r2	r1	r4	r5
2s	r3	r1	r1	r5
3s	r3	r1	r4	r5
1p	r1	r1	r1	r5
2p	r1	r1	r1	r5
3p	r2	r1	r4	r5

	cong. pres.	cong. imp.	cond. pres.	imp. pres.
1s	r2	r1	r5	-
2s	r2	r1	r5	r3
3s	r2	r1	r5	r2
1p	r1	r1	r5	r1
2p	r1	r1	r5	r1
3p	r2	r1	r5	r2

inf. pres.	ger. pres.	part. pres.	part. pass.
r1	r1	r1	r6

Questo approccio permette di affrontare la gestione dei verbi irregolari, organizzando ogni differente radice (r1, ..., r6) in un lessico dedicato. Avremo, dunque, un lessico `r1` che conterrà le desinenze verbali indicate con questa sigla nella tabella qui mostrata. Chiaramente, anche se le voci saranno le stesse, è necessario implementare tre diversi lessici `r1`, in quanto le desinenze variano in accordo alla coniugazione del verbo.

```
LEXICON r1Ere


    +V_FIN+IND+PRES+1+PLUR:+IND+PRES+1+PLUR*iamo #;
    +V_FIN+IND+PRES+2+PLUR:+IND+PRES+2+PLUR*ete #;
    +V_FIN+IND+IMPERF+1+SING:+IND+IMPERF+1+SING*evo #;
    +V_FIN+IND+IMPERF+2+SING:+IND+IMPERF+2+SING*evi #;
    +V_FIN+IND+IMPERF+3+SING:+IND+IMPERF+3+SING*eva #;
    ...
    +V_FIN+IMP+PRES+2+PLUR:+IMP+PRES+2+PLUR*ete #;
    +V_NOFIN+INF+PRES:+INF+PRES*ere #;
    +V_PP+PART+PRES+SING:+PART+PRES+SING*ente #;
    +V_PP+PART+PRES+PLUR:+PART+PRES+PLUR*enti #;
    +V_NOFIN+GER+PRES:+GER+PRES*endo #;
```

A fianco del lessico `r1Ere`, avremo, dunque, anche `r1Are` ed `r1Ire`.

Per comporre, infine, tutte le forme verbali di un verbo irregolare, è necessario inserire nel lessico `root` un'entrata per ogni radice diversa, da abbinare alla giusta classe di continuazione:

```
l_sedere:sed+V_ERE r1Ere;
l_sedere:sied+V_ERE r2Ere;
l_sedere:sied+V_ERE r3Ere;
l_sedere:sed+V_ERE r4Ere;
l_sedere:sied+V_ERE r5Ere;
```

```
l_sedere:sed+V_ERE r6Ere;
```

Questo approccio di gestione delle diverse radici, permette, in maniera molto semplice, di gestire anche quelle forme verbali che ammettono più di una radice.

Prendendo come esempio il verbo sedere, possiamo notare come siano ammesse entrambe le forme seggo e siedo. Seguendo lo schema utilizzato in questo progetto, è sufficiente aggiungere un entrata nel lessico `root` per gestire la radice diversa, utilizzando la stessa classe di continuazione:

```
l_sedere:sied+V_ERE r2Ere;
l_sedere:segg+VEL_Y+V_ERE r2Ere;
```

Questo esempio offre, inoltre, la possibilità di mostrare l'utilizzo di un ulteriore lessico, `rQ`. Vi sono infatti alcuni verbi della coniugazione in Ere che, nella prima e terza persona singolare e nella terza persona plurale del passato remoto, ammettono due forme verbali diverse (-ei ed -etti). A fianco, quindi, della desinenza "ufficiale" -ei, contenuta nel lessico `r4`, si aggiunge la desinenza -etti, contenuta nel lessico `rQ`. In questo modo, risulta molto semplice gestire questa particolarità, sia per i verbi regolari, che per quelli irregolari, in quanto necessita della semplice aggiunta di una riga:

```
l_sedere:sed+V_ERE rQ;
LEXICON rQ
```

```
+V_FIN+IND+PAST+1+SING:+IND+PAST+1+SING*etti #;
+V_FIN+IND+PAST+3+SING:+IND+PAST+3+SING*ette #;
+V_FIN+IND+PAST+3+PLUR:+IND+PAST+3+PLUR*ettero #;
```

In maniera simile ad `rQ`, vi sono altri lessici, creati per gestire ulteriori irregolarità dello schema presentato precedentemente. Alcuni verbi in Ere e rari verbi in Ire, infatti, subiscono un processo di troncamento di alcune desinenze, come ad esempio il verbo porre, che subisce vari troncamenti, tra cui quello nella desinenza dell'infinito (porre invece di ponere) e nelle desinenze del futuro indicativo (porrò invece di ponerò). Questa ulteriore irregolarità viene gestita tramite i lessici `r4` (per il passato remoto), `r5` (per il futuro ed il condizionale presente), `r6` (per il participio passato) ed `r7` (per l'infinito), che presentano, quindi, le desinenze delle relative voci troncate.

```
l_porre:pos+V_ERE r4;
l_porre:por+V_ERE r5;
l_porre:post+V_ERE r6;
l_porre:por+V_ERE r7;

LEXICON r6

    +V_PP+PART+PAST+MASC+SING:+PART+PAST+MASC+SING*o #;
    +V_PP+PART+PAST+FEMM+SING:+PART+PAST+FEMM+SING*a #;
```

```
        +V_PP+PART+PAST+MASC+PLUR:+PART+PAST+MASC+PLUR*i #;
        +V_PP+PART+PAST+FEMM+PLUR:+PART+PAST+FEMM+PLUR*e #;

    LEXICON r7

        +V_NOFIN+INF+PRES:+INF+PRES*re #;
```

Vi sono poi alcuni lessici meno rilevanti che non verranno illustrati maggiormente in dettaglio, in quanto gestiscono parziali irregolarità, già descritte in precedenza. È il caso, ad esempio, di verbi che presentano un comportamento regolare, all'infuori di un troncamento nella desinenza del participio passato. Questi sono, quindi, gestiti mediante un comportamento regolare ed un participio passato troncato.

Classi di continuazione nominali

Con classi di continuazione nominali intendiamo quei lessici che gestiscono la flessione dei sostantivi che non modificano il loro genere. Questi sostantivi contengono, quindi, l'informazione relativa al genere a livello del lemma, ovvero nella entry del lessico `root`.

Una prima suddivisione che qui è stata effettuata tra i sostantivi regolari appartenenti a questa categoria è quella relativa alla classe di flessione: sostantivi in *a* (ad es. monarca), in *e* (ad es. cancelliere) ed in *o* (ad es. feudo). La seconda distinzione che viene fatta è quella tra sostantivi inerentemente maschili e inerentemente femminili (ad es. termosifone e sedia).

Per gestire i sostantivi regolari appartenenti a questa categoria vi sono, quindi, i lessici: `SufNomMascA`, `SufNomMascE`, `SufNomMascO`, `SufNomFemmA`, `SufNomFemmE`, `SufNomFemmO`.

```
LEXICON SufNomFemmA

    +SING:+SING*a #;
    +PLUR:+PLUR*e #;
    : SufModFemm;
```

In questo esempio possiamo notare l'utilizzo di : `SufModFemm`. Con questa scrittura, che verrà approfondita in seguito, si richiama la classe di continuazione contenente i vari suffissi di alterazione (vezzeggiativi, peggiorativi, e così via).

La classe di continuazione sopra mostrata verrà richiamata da entrate nel lessico `root`, di cui mostriamo un esempio:

```
l_fotocamera+NN+FEMM:fotocamer+NN+FEMM SufNomFemmA;
```

Le irregolarità dei sostantivi gestite in questa categoria sono quelle relative a sostantivi difettivi e sovrabbondanti. Esistono due tipologie di sostantivi chiamati difettivi: quelli che vengono utilizzati solo nella forma singolare o nella forma plurale. Per mantenere la convenzione già usata per i verbi, alla prima classe è stata assegnata la sigla `r1`, mentre alla seconda classe, la sigla `r2`.

```
LEXICON SufNomFemmAr1
```

```
        +SING:+SING*a #;
        : SufModFemm;

    LEXICON SufNomFemmAr2

        +PLUR:+PLUR*e #;
        : SufModFemm;
```

Per quanto riguarda i sostantivi sovrabbondanti, si gestisce il caso in cui un sostantivo accetti una forma singolare e due forme plurali, tramite la creazione del lessico `SufNomIrr`. Un esempio è quello del termine braccio, declinabile solamente al maschile, nel caso singolare ed invece nel maschile bracci e nel femminile braccia, nel caso plurale.

```
    LEXICON SufNomIrr

        +MASC+SING:+MASC+SING*o #;
        +MASC+PLUR:+MASC+PLUR*i #;
        +FEMM+PLUR:+FEMM+PLUR*a #;
        : SufModMasc;
```

Infine, vengono gestiti anche quei sostantivi che non ammettono i suffissi di alterazione, tramite lessici analoghi ai precedenti, il cui nome termina, però, con `NoMod`.

Come si accennava nell'introduzione, queste classi di continuazione non vengono utilizzate solamente per i sostantivi, ma anche per altre categorie

lessicali che ne mutuano il comportamento. Di seguito viene mostrato l'esempio del lemma rispetto, che può essere considerato sia un sostantivo che un avverbio. Vengono elencate quindi le entry di questo lemma nel lessico `root` e la classe di continuità utilizzata.

```
l_rispetto+ADV+MASC:rispett+ADV+MASC SufNomMascONoMod;
l_rispetto+NN+MASC:rispett+NN+MASC SufNomMascONoMod;
LEXICON SufNomMascONoMod

    +SING:+SING*o #;
    +PLUR:+PLUR*i #;
```

Classi di continuazione aggettivali

Con classi di continuazione aggettivali intendiamo quei lessici che gestiscono la flessione degli aggettivi e dei sostantivi che modificano sia il loro genere, che il loro numero. La suddivisione che viene qui effettuata, dunque, riguarda solamente la classe di flessione: aggettivi o sostantivi che terminano in *a* (ad es. ciclista), in *e* (ad es. infelice) ed in *o* (ad es. caldo).

Per la loro gestione sono stati quindi creati i lessici `SufNomA`, `SufNomE`, `SufNomO`, `SufAggA`, `SufAggE`, `SufAggO`. I sostantivi gestiti con queste classi, presentano un'entrata lessicale nel lessico `root` che indica la loro categoria lessicale, mentre, come visto prima, per alcuni aggettivi, la categoria non viene specificata, in quanto, ad esempio, l'aggiunta del morfema -issimamente

provoca un cambiamento da aggettivo ad avverbio. Nei lessici riferiti agli aggettivi abbiamo inoltre la gestione del suffisso -issimo.

```
l_elettricista+NN:elettricist+NN SufNomA;
l_revisionista:revisionist SufAggA;


LEXICON SufNomA

    +MASC+SING:+MASC+SING*a #;
    +FEMM+SING:+FEMM+SING*a #;
    +MASC+PLUR:+MASC+PLUR*i #;
    +FEMM+PLUR:+FEMM+PLUR*e #;
    +MASC:+MASC SufModMasc;
    +FEMM:+FEMM SufModFemm;

LEXICON SufAggA

    +ADJ+MASC+SING:+ADJ+MASC+SING*a #;
    +ADJ+FEMM+SING:+ADJ+FEMM+SING*a #;
    +ADJ+MASC+PLUR:+ADJ+MASC+PLUR*i #;
    +ADJ+FEMM+PLUR:+ADJ+FEMM+PLUR*e #;
    +ADJ+MASC:+ADJ+MASC SufModMasc;
    +ADJ+FEMM:+ADJ+FEMM SufModFemm;
    +ADJ:+ADJ SufIssimo;
    +ADV+SUP:+ADV+SUP*issimamente #;
```

Anche per questa classe di continuazione, vi sono irregolarità relative ad una modifica della radice, a cui vengono, quindi, assegnate, mantenendo la convenzione, le sigle `r1` ed `r2`. Questo cambio di radice si verifica, ad esempio nei lemmi tuo, suo e mio.

```
l_tuo+ADJ_POS:tuo+ADJ_POS SufAggOr1;
l_tuo+PRON_POS:tuo+PRON_POS SufAggOr1;
l_tuo+ADJ_POS:tu+ADJ_POS SufAggOr2;
l_tuo+PRON_POS:tu+PRON_POS SufAggOr2;


LEXICON SufAggOr1

    +MASC+PLUR:+MASC+PLUR*i #;

LEXICON SufAggOr2

    +MASC+SING:+MASC+SING*o #;
    +FEMM+SING:+FEMM+SING*a #;
    +FEMM+PLUR:+FEMM+PLUR*e #;
```

Un'ultima irregolarità gestita è quella relativa alla classe di aggettivi e sostantivi terminanti in *e*, ma che non seguono le regole già mostrate per questa categoria. Questi aggettivi e sostantivi sono gestiti tramite il lessico `SufAggIrr`.

```
l_accattone:accatton SufAggIrr;
```

```
LEXICON SufAggIrr

    +ADJ+MASC+SING:+ADJ+MASC+SING*e #;

    +ADJ+FEMM+SING:+ADJ+FEMM+SING*a #;

    +ADJ+MASC+PLUR:+ADJ+MASC+PLUR*i #;

    +ADJ+FEMM+PLUR:+ADJ+FEMM+PLUR*e #;

    +ADJ+MASC:+ADJ+MASC SufModMasc;

    +ADJ+FEMM:+ADJ+FEMM SufModFemm;

    +ADJ:+ADJ SufIssimo;

    +ADV+SUP:+ADV+SUP*issimamente #;
```

Altre classi di continuazione

Qui verranno elencate quelle classi di continuazione che non risultano necessarie per una corretta gestione della lingua italiana, ma che sono state create per rendere più agevole l'implementazione, la lettura e la comprensione di questo codice.

Come già citato in precedenza, vi sono, dunque, due lessici che gestiscono le alterazioni, ovvero quel caso particolare di suffissazione, che modifica il concetto espresso dalla parola, senza però alterarne la categoria lessicale di appartenenza. In questa categoria sono compresi quindi, suffissi diminutivi, accrescitivi, vezzeggiativi e peggiorativi.

```
LEXICON SufModMasc

    +SING+DIM:+SING+DIM*ino #;
```

```
        +PLUR+DIM:+PLUR+DIM*ini #;
        +SING+AUM:+SING+AUM*one #;
        +PLUR+AUM:+PLUR+AUM*oni #;
        +SING+VEZ:+SING+VEZ*uccio #;
        +PLUR+VEZ:+PLUR+VEZ*ucci #;
        +SING+VEZ:+SING+VEZ*etto #;
        +PLUR+VEZ:+PLUR+VEZ*etti #;
        +SING+PEG:+SING+PEG*accio #;
        +PLUR+PEG:+PLUR+PEG*acci #;

    LEXICON SufModFemm

        +SING+DIM:+SING+DIM*ina #;
        +PLUR+DIM:+PLUR+DIM*ine #;
        +SING+AUM:+SING+AUM*ona #;
        +PLUR+AUM:+PLUR+AUM*one #;
        +SING+VEZ:+SING+VEZ*uccia #;
        +PLUR+VEZ:+PLUR+VEZ*ucce #;
        +SING+VEZ:+SING+VEZ*etta #;
        +PLUR+VEZ:+PLUR+VEZ*ette #;
        +SING+PEG:+SING+PEG*accia #;
        +PLUR+PEG:+PLUR+PEG*acce #;
```

Un ultimo lessico specificato è quello che riguarda il suffisso superlativo -issimo e derivati relativo alle forme aggettivali.

```
LEXICON SufIssimo

    +MASC+SING+SUP:+MASC+SING+SUP*issimo #;

    +FEMM+SING+SUP:+FEMM+SING+SUP*issima #;

    +MASC+PLUR+SUP:+MASC+PLUR+SUP*issimi #;

    +FEMM+PLUR+SUP:+FEMM+PLUR+SUP*issime #;
```

3.2.2 Il file delle regole

Dopo aver definito il file per HFST-LEXC, è necessario creare un file che gestisca le regole di riaggiustamento grafotattico, da compilare tramite HFST-TWOLC. Nella scrittura di questo file sono state seguite le indicazioni fornite nel capitolo 3.1.2, relativo alla spiegazione di HFST-TWOLC. Verranno quindi illustrate in dettaglio le diverse parti che compongono il file italiano.twolc.

Alfabeto

La prima parte del file è caratterizzata dalla definizione dei caratteri che compongono l'alfabeto in uso. A fianco dei classici caratteri alfabetici espressi nella codifica UTF-8, vi è l'elenco di tutti i simboli multicarattere dichiarati nel file italiano.lexc.

Già in questa sezione, vi è un primo semplice fenomeno di riaggiustamento: qui infatti, ogni simbolo multicarattere non coinvolto esplicitamente in un riaggiustamento definito da una regola, viene eliminato dalla forma che si sta creando. Una volta, quindi, che l'utente utilizzerà il generatore per

ottenere una forma, questi simboli non compariranno nel risultato richiesto dall'utente, ma rimarranno presenti come *features* a fianco del lemma.

Questo risultato è ottenuto tramite la scrittura `%+V_ARE:0`, che specifica, ad esempio, l'eliminazione del simbolo `%+V_ARE`.

```
Alphabet
a à b c d e è é f g h i ì j k l m n o ò p q r s t u ù v w x y z
%*:0 %-
%+NN:0 %+NN_P:0
%+ADJ:0 %+ADJ_DIM:0 %+ADJ_IND:0
%+ADJ_IES:0 %+ADJ_NUM:0 %+ADJ_POS:0
%+PRON:0 %+PRON_P:0 %+PRON_DIM:0 %+PRON_IND:0
%+PRON_IES:0 %+PRON_REL:0 %+PRON_POS:0 %+PRON_PER:0
%+ADV:0 %+INT:0 %+PREP:0 %+PREP_A:0
%+CONJ:0 %+CONJ_S:0 %+ART:0
%+V_ARE:0 %+V_ERE:0 %+V_IRE:0 %+V_FIN:0 %+V_NOFIN:0 %+V_PP:0
%+INF:0 %+IND:0 %+SUBJ:0 %+COND:0 %+IMP:0 %+PART:0 %+GER:0
%+PRES:0 %+IMPERF:0 %+PAST:0 %+FUT:0
%+1:0 %+2:0 %+3:0
%+MASC:0 %+FEMM:0
%+SING:0 %+PLUR:0
%+DIM:0 %+AUM:0 %+VEZ:0 %+PEG:0 %+SUP:0
%+GLI:0 %+VEL_A:0 %+VEL_Y:0 #:0;
```

Alcuni di questi simboli sono stati inseriti in alcune entrate del file italiano.lexc, nonostante non siano state sfruttate in quel contesto. Vengono infatti utilizzati all'interno di questo file, per poter applicare alcune regole di riaggiustamento.

Sets e Definitions

In queste sezioni sono stati creati dei raggruppamenti che sono risultati utili, per una più agevole scrittura e comprensione delle regole di riaggiustamento. Qui di seguito ne viene mostrato uno stralcio a titolo di esempio:

```
Head = %+NN: | %+NN_P: | %+V_ARE: | %+V_ERE: | %+V_IRE:
| %+V_FIN: | %+V_NOFIN: | %+V_PP: | %+ADJ: | %+ADJ_DIM:
| %+ADJ_IND: | %+ADJ_IES: | %+ADJ_NUM: | %+ADJ_POS:
| %+ADV: | %+PRON: | %+PRON_P: | %+PRON_DIM: | %+PRON_IND:
| %+PRON_IES: | %+PRON_REL: | %+PRON_POS: | %+PRON_PER:
| %+INT: | %+PREP: | %+PREP_A: | %+CONJ: | %+CONJ_S: | %+ART: ;
Nonverbi = %+NN: | %+NN_P: | %+ADJ: | %+ADJ_DIM: | %+ADJ_IND:
| %+ADJ_IES: | %+ADJ_NUM: | %+ADJ_POS: | %+ADV: | %+PRON:
| %+PRON_P: | %+PRON_DIM: | %+PRON_IND: | %+PRON_IES:
| %+PRON_REL: | %+PRON_POS: | %+PRON_PER: | %+INT: | %+PREP:
| %+PREP_A: | %+CONJ: | %+CONJ_S: | %+ART: ;
Verbi = %+V_ARE: | %+V_ERE: | %+V_IRE: | %+V_FIN:
| %+V_NOFIN: | %+V_PP: ;
```

`Head`, che risulta quello meno intuitivo, rappresenta l'insieme di tutte le possibili categorie lessicali.

Le regole

In questa sezione del file sono esplicitate le regole che servono per effettuare quei riaggiustamenti necessari, quando vengono accostati i morfemi, durante la creazione delle forme lessicali.

L'applicazione delle regole di riaggiustamento dipende dalla presenza o meno dei simboli multicarattere `+GLI`, `+VEL_A` e `+VEL_Y`. Innanzitutto analizziamo quelle regole che non coinvolgono questi simboli. Queste vengono, dunque, applicate a tutte le forme in cui questi simboli non compaiono e che rispettano il contesto della regola.

```
"Eliminazione i"
i:0 <=> _ (Head) (Modo) (Tempo) (Pers) (Genere)
        (Numero) (Modif) %*: i [a | à] ;
```

Come è possibile notare dall'esempio, questa regola si occupa dell'eliminazione dell'ultima *i* del primo morfema, nel caso questa sia seguita da un dittongo *ia* (ad es. marci-are diventa nella forma indicativa presente prima persona plurale, marc-iamo e non marci-iamo).

Analizzando la struttura della regola, `i:0` esprime l'operazione da effettuare, mentre il contesto è indicato dopo il simbolo `<=>`. All'interno di quest'ultimo possiamo notare la presenza del carattere `_`, che indica in quale

posizione deve essere effettuata l'operazione e l'utilizzo delle parentesi, che indica che la presenza di queste classi di simboli, precedentemente definite, è facoltativa. Lo sfruttamento delle parentesi, permette, in questo modo, di scrivere una sola regola che risulta valida per tutte le categorie lessicali. Infatti, quando questa viene analizzata per un verbo, saranno assenti le classi `Genere` e `Modif`, mentre per un sostantivo, saranno sicuramente assenti le classi `Modo` e `Tempo`.

Analogamente alla regola vista nell'esempio, sono presenti anche regole per l'inserimento di caratteri.

```
"Inserimento i per c|g"
0:i <=> [c | g] _ (Head) (Modo) (Tempo) (Pers) (Genere)
        (Numero) (Modif) %*: [a | à | o | ò | u | ù] ;
```

In questo caso, si gestisce l'inserimento di una i tra una c o g al termine del primo morfema, ed una delle vocali a, o ed u. L'unica differenza, rispetto alla regola mostrata prima, è la presenza di `0:i` al posto di `i:0`, ad indicare l'inserimento invece dell'eliminazione.

Vengono ora analizzate le regole che dipendono dai simboli `+GLI`, `+VEL_A` e `+VEL_Y`. Mostriamo di seguito un esempio relativo al simbolo `+GLI`, che viene utilizzato per identificare quei morfemi che terminano con la vocale i:

```
"Eliminazione i con glide"
i:0 <=> _ %+GLI: (Head) (Modo) (Tempo) (Pers) (Genere)
        (Numero) (Modif) %*: [i | ì] ;
```

La presenza del simbolo +GLI permette, quindi, di gestire quelle forme che non ammettono il raddoppio della *i* nel processo di flessione (ad es. fischi-o diventa al plurale fisch-i e non fischi-i).

Il simbolo +VEL serve, invece, per identificare quei morfemi che terminano in consonante velare *c* o *g*. Per evitare ambiguità sono stati utilizzati due simboli, +VEL_A e +VEL_Y, di cui mostriamo di seguito due esempi di utilizzo:

```
"Inserimento h per c|g con velar = y"
%+VEL_Y:h <=> [c | g] _ (Head) (Modo) (Tempo) (Pers) (Genere)
              (Numero) (Modif) %*: [e | è | é | i | ì] ;
"Inserimento h per c|g con velar = a"
%+VEL_A:h <=> [c | g] _ (Nonverbi) (Genere) (Numero)
              (Modif) %*: [e | è | é] ;
```

Nella prima regola vediamo come il simbolo +VEL_Y indichi la necessità dell'inserimento dell'*h* nel caso in cui il primo morfema sia seguito da una delle vocali *e* o *i* (ad es. cuoc-o diventa al plurale cuoch-i e non cuoc-i).

Nella seconda regola invece si gestisce l'inserimento dell'*h* nel caso in cui il primo morfema sia seguito dalla vocale *e*. Questo esempio rappresenta un caso particolare, in quanto l'applicazione della regola è limitata alle forme non verbali, a differenza di tutte le regole viste in precedenza (ad es. sovietic-o diventa al plurale femminile sovietich-e e non sovietic-e).

Capitolo 4

Fase di testing

Per verificare la correttezza dell'implementazione dei trasduttori creati con i tool HFST, è stata effettuata una fase di testing, sfruttando Magic, un software consolidato, come metro di paragone.

4.1 Magic

Magic è una piattaforma di morfologia computazionale per l'analisi e la generazione di parole italiane. È stato sviluppato in C nel 1999 da Marco Battista e Vito Pirrelli [3]. Il nucleo della piattaforma è costituito da tre moduli: il compilatore del lessico, l'analizzatore morfologico ed il generatore morfologico.

Il compilatore del lessico rappresenta un'interfaccia tra il lessico definito dall'utente e i componenti di analisi e di generazione morfologica. La

prima funzione del compilatore è quella di rendere linguisticamente fondata la scrittura del lessico. La seconda funzione è quella di eseguire un controllo sull'integrità dei dati specificati dall'utente. La terza è, infine, quella di memorizzare il lessico inserito dall'utente, in un formato indicizzato, che verrà utilizzato dalle ricerche lessicali, effettuate dall'analizzatore.

L'input del compilatore del lessico è un file testuale definito dall'utente, che contiene un elenco di entrate lessicali che si utilizzeranno per l'analisi e la generazione morfologica. L'insieme del lessico definito dall'utente si articola in realtà in diversi moduli, cioè in differenti file-utente, che gestiscono particolari tipologie di entrate lessicali. Il file MG_lemmas, ad esempio, contiene il repertorio dei lemmi, mentre il file MG_endings contiene le desinenze. Di seguito viene mostrato un esempio di entrata lessicale: la codifica della desinenza -i del congiuntivo presente dei verbi della prima coniugazione.

```
$
phon=>i
morphtype=>i_ending
inflex_class=>[v,a]
head=>v_fin
pers=>(1;2;3)
numb=>s
tense=>pres
mood=>subj
$
```

L'analizzatore morfologico prende in input liste di parole e carica in memoria i vari lemmari forniti dal compilatore e gli indici che puntano agli oggetti contenuti in quei repertori. L'analizzatore è composto da due moduli che gestiscono le regole: un componente di morfografemica, per catturare riaggiustamenti di tipo grafotattico ed un componente di morfotassi per verificare, sulla base delle informazioni immagazzinate nel lessico, la possibilità che una radice, una desinenza, un clitico, e così via, possano co-selezionarsi per dare luogo a una parola corretta. Questa operazione di verifica delle restrizioni lessicali di co-selezione è effettuata tramite l'unificazione dei valori di certi attributi. L'output dell'analizzatore consiste in un oggetto linguistico di tipo `word`, che contiene il valore di dieci attributi relativi alla parola, tra cui lemma, forma ortografica, persona, genere, numero e così via:

```
-> ama
[1] MACRO:word[l_amare,ama,v_fin,(3/2),!,s,pres,(ind/imp),!,!]
```

Il generatore prende in input una lista di lemmi. Caricato il lessico in memoria, viene cercata, per ogni lemma in input, la lista delle forme che, nel lessico, corrispondono a quel lemma. Sulla base di questa lista, di solito di un solo elemento per i lessemi regolari, il componente di morfotassi verifica la correttezza delle combinazioni tra le radici e le desinenze immagazzinate nel lessico ed il componente di morfografemica esegue eventuali riaggiustamenti grafotattici come, ad esempio, l'eventuale inserimento di un'*h* o di una *i*. L'output del generatore consiste, per ogni lemma, nella lista di tutte le

parole flesse del paradigma di quel lemma, ognuna specificata per tutti gli attributi rilevanti:

```
-> MACRO:word[l_cane,_,_,_,_,plur,_,_,_,_]
[1] MACRO:word[l_cane,cani,nn,!,m,p,!,!,!,pos]
```

Da notare l'utilizzo del simbolo _, che rappresenta l'operatore di sottospecificazione. Questa è la semplificazione estrema della disgiunzione semplice: indica cioè la possibile presenza di qualsiasi valore. Il simbolo !, invece, sta ad indicare che gli attributi corrispondenti (nell'esempio persona, tempo, modo e caso) non sono rilevanti per la categoria lessicale in questione (in questo caso nome).

4.2 Testing

La fase di testing che viene analizzata in questo capitolo è quella che è stata eseguita una volta terminata l'implementazione del software AnIta. È però necessario specificare che un primo testing di tipo preventivo, è stato effettuato durante tutta la fase di implementazione del lemmario, delle regole morfotattiche e delle regole grafotattiche. Man mano che sono state implementate queste regole, sono stati effettuati dei test mirati e specifici sulle parole coinvolte nelle regole in oggetto.

Una volta terminata l'implementazione dei trasduttori, è stato sfruttato Magic per eseguire un test più generale e approfondito. Tra i tool di questo

software, vi è la possibilità di ottenere in output una lista di tutte le forme, generate dai lemmi memorizzati nel dizionario. Come è già stato detto in precedenza, anche HFST fornisce questa opzione, tramite il comando HFST-FST2STRINGS. Sfruttando questi tool, è stato, quindi, effettuato un test basato sul confronto tra le stringhe generate dal trasduttore di Magic e quelle generate dal trasduttore implementato con HFST.

Un'altra motivazione per cui è stato seguito questo approccio, è il formato di output con cui Magic fornisce l'elenco di tutte le forme generate dai lemmi memorizzati.

```
[1] lemma[l_avere,abbia,v_fin,_,!,s,pres,subj,!,!,abbia]
```

Come si può notare, il formato utilizzato da Magic segue un rigido schema e risulta, quindi, facilmente manipolabile per modificarne la struttura. In questo modo, è stato possibile adattare questo elenco al formato di output utilizzato da HFST, tramite uno script bash ed eseguire, così, un confronto di correttezza.

Per poter eseguire un test di questo tipo, è stato necessario, innanzitutto, escludere tutti i lemmi gestiti da un trasduttore, ma non dall'altro. È stato quindi effettuato un confronto tra tutti i lemmi e sono stati "nascosti" quelli non presenti contemporaneamente in entrambi i trasduttori. Essendo relativamente pochi i lemmi presenti in AnIta, ma non nel trasduttore di Magic, è

stato possibile effettuare senza sforzo, un test sulle forme generate da questi lemmi, in maniera manuale.

Il testing basato sul confronto è stato suddiviso in due fasi: una prima in cui si confrontano le semplici parole generate ed una seconda, più approfondita, in cui vengono confrontate tra loro le forme con i relativi lemmi e caratteristiche. Utilizzare due livelli di testing è risultato utile, in quanto in ogni fase viene gestito un minor numero di errori. La creazione e la gestione dei file per la seconda fase del testing e, di conseguenza, il rilevamento degli errori con questa procedura, risulta, infatti, più complesso. Perciò, una scrematura degli errori, effettuata tramite una prima fase, più semplice, diminuisce il lavoro richiesto per la seconda fase di test.

Il primo confronto, semplice e meno selettivo, è stato eseguito ottenendo la lista delle forme generate da AnIta:

```
hfst-fst2strings generatore.hfst
```

Questo comando restituisce la lista di tutte le forme generate, con questo formato:

```
l_pensare+V_FIN+IND+FUT+2+SING:penserai
```

Tramite il comando awk, sono state memorizzate in un file tutte le forme generate, ignorando, per il momento, i lemmi e le caratteristiche, in modo da avere un elenco di semplici parole.

Per ottenere la stessa lista di forme dal trasduttore di Magic, è stato necessario effettuare, anche qui, un'estrazione, per avere una lista delle parole riconosciute. Ciò che viene restituito da Magic ha, infatti, la seguente forma:

```
-> "penserai"
[1] lemma[l_pensare,penserai,v_fin,2,!,s,fut,ind,!,!]
```

È stata, quindi, estratta la parola delimitata dai simboli "" ed è stata, poi, memorizzata in un file testuale.

Una volta ottenuti i due elenchi, mediante un confronto parola per parola, sono stati rilevati gli errori da correggere.

Gli errori evidenziati tramite questo test riguardano i casi in cui siano state generate, tramite HFST, parole scorrette o i casi in cui non siano state generate classi di parole. Le parole scorrette derivano da una errata concatenazione di radici e suffissi, causata da errori nelle classi di continuazione e nei sotto-lessici.

Il secondo confronto è stato effettuato per evidenziare errori di attribuzione di caratteristiche alle parole. In questo secondo tipo di test, vengono rilevate, infatti, anche quelle parole errate che, per caso, coincidono con altre parole corrette e quelle forme che non vengono generate, ma sono comunque presenti perché generate a partire da altri lemmi.

Per effettuare questo confronto, vengono utilizzate, infatti, non solo le semplici parole generate, ma le forme lessicali in combinazione con il lemma e le caratteristiche, che ne hanno causato la generazione.

Per eseguire questo test è necessaria, quindi, la lista delle forme generate da AnIta, ottenuta tramite lo stesso tool visto precedentemente (HFST-FST2STRINGS), il cui risultato non verrà modificato in alcun modo. Occorre, poi, convertire la lista delle forme generate da Magic, nello stesso formato fornito da HFST. Questa operazione presenta, però, una difficoltà: Magic "collassa" in una linea sola, le forme uguali con caratteristiche diverse:

```
-> "abbia"
[1] lemma[l_avere,abbia,v_fin,_,!,s,pres,subj,!,!]
```

Nel caso mostrato in questo esempio, il simbolo `_` posizionato dopo `v_fin`, sta ad indicare che il campo "persona" può assumere tutti i valori possibili, in questa forma. Infatti, "abbia" rimane uguale per tutte e tre le persone singolari del congiuntivo presente. È necessario, quindi, far combaciare questo formato con quello di HFST, che, per la stessa forma, fornisce in output:

```
l_avere+V_FIN+SUBJ+PRES+1+SING:abbia
l_avere+V_FIN+SUBJ+PRES+2+SING:abbia
l_avere+V_FIN+SUBJ+PRES+3+SING:abbia
```

Per fare ciò, è stata necessaria la preventiva espansione di tutte le forme, fornite da Magic, che presentano simboli `_`, mediante la loro sostituzione con linee che presentino tutti i possibili valori attribuibili a quel campo.

In questo modo è stato possibile eseguire un confronto linea per linea e rilevare, così, le differenze da correggere.

Questa seconda fase di testing è risultata utile per rilevare quegli errori caratterizzati dal riconoscimento di forme corrette generate da lemmi errati. Sono state rilevate, ad esempio, alcune forme valutative attribuite ad una categoria lessicale errata (aggettivo) invece che a quella corretta (sostantivo).

In generale, si può affermare che, anche grazie alla fase di test preventivo eseguita durante lo sviluppo di AnIta, il testing qui descritto non ha rilevato errori concettuali nella formalizzazione delle regole morfotattiche e grafotattiche per la lingua italiana. Si può dire, però, che esso è risultato necessario in quanto, grazie ad esso, sono stati comunque individuati alcuni errori di entità minore, che avrebbero minato la correttezza dell'applicazione sviluppata.

Capitolo 5

Estensione dell'analizzatore

Il trasduttore implementato in questo progetto e, in generale, tutti i trasduttori basati su metodologie FST, mantengono una relazione tra lemma e forma, basata sulle caratteristiche della categoria lessicale.

Spesso capita, però, che ci si trovi di fronte al bisogno di ottenere più informazioni relative ad una forma lessicale, oltre a quelle relative al lemma di riferimento; potrebbe esistere la necessità di gestire ulteriori informazioni di tipo morfologico o fonologico per ogni parola riconosciuta.

Queste problematiche non risultano, però, risolvibili con un trasduttore simile a quello implementato in questo progetto. L'automa ottenuto è in grado di associare ad ogni forma lessicale, solamente un'altra informazione, tramite l'applicazione ad ogni arco di due etichette (che noi indichiamo con la scrittura a:b); nel nostro caso, queste etichette sono rappresentate dalla forma lessicale e dal lemma di partenza e le relative *features*. Per ottenere la gestione

di più informazioni per ogni forma, sarebbe, quindi, necessario memorizzare, per ogni arco dell'automa risultante, più di due etichette. Questo non è possibile con il formalismo FST.

Grazie al formalismo dei trasduttori, però, è possibile un approccio diverso al problema: si possono creare trasduttori diversi per ogni informazione relativa alla forma lessicale da gestire. Le relazioni che si vengono, poi, a creare tra questi automi, permettono di gestire ed ottenere la totalità delle informazioni memorizzate.

In particolare, HFST mette a disposizione una serie di operazioni che possono essere effettuate fra trasduttori, tra cui la congiunzione, la concatenazione, la composizione, e così via. In questo modo viene permessa la completa gestione delle relazioni da creare tra gli automi delle diverse informazioni (o *features*).

In questo capitolo verrà illustrata in dettaglio questa procedura, mostrandola attraverso l'aggiunta di una *feature* da memorizzare per ogni parola.

5.1 Struttura morfologica

L'esempio che qui viene mostrato, riguarda la memorizzazione della struttura (`MStruct`) della forma generata dal lemma. Per struttura della forma intendiamo l'evidenziazione dei singoli morfemi che compongono la parola, messi in risalto, attraverso l'utilizzo di simboli separatori.

`riaddormentiamo` → `ri>ad>dorment-iamo`

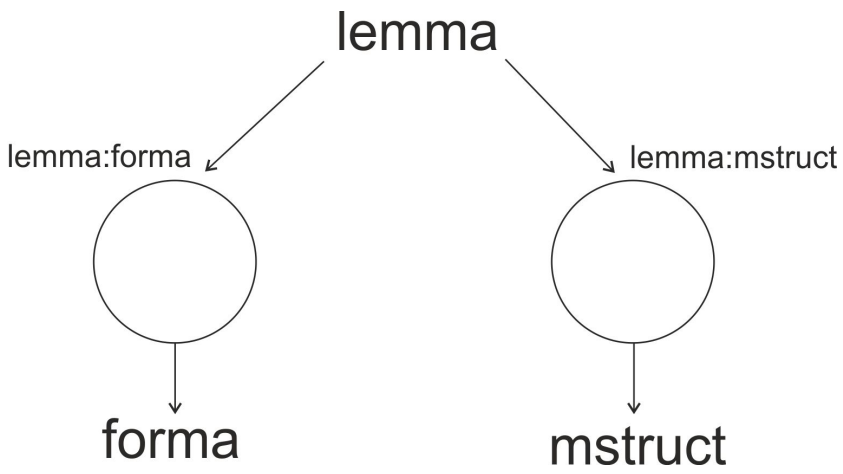

Figura 5.1: Generazione tramite l'utilizzazione di due automi

Idealmente, ciò che risulta necessario ottenere è, a fianco dell'automa già descritto che gestisce la relazione lemma:forma, l'implementazione di uno che gestisca la relazione lemma:mstruct. È importante, però, che l'informazione riguardante mstruct compaia sia durante una generazione a partire da un lemma, che durante l'analisi di una forma.

Avendo a disposizione i due automi, occorre gestirli in modo differente a seconda del task che si vuole risolvere: per eseguire la generazione di una forma a partire da un lemma, è necessario fornire lo stesso input ad entrambi gli automi, per ottenere le due diverse informazioni, forma e mstruct, relative allo stesso lemma.

Per eseguire invece l'analisi di una forma ed ottenere il relativo lemma, è prima necessario trovare tutti i lemmi che possono averla generata, utilizzando l'analizzatore, e poi fornirli in input all'automa che li mette in relazione con mstruct.

Analizzando la questione da un punto di vista più pratico, per riuscire ad eseguire l'operazione di generazione, si potrebbe sfruttare il tool HFST-CONCATENATE, che permette, come è stato già accennato nel paragrafo 3.1.3, di concatenare due trasduttori e "riconoscere" stringhe accettate dal primo automa concatenate a stringhe accettate dal secondo automa. In questo modo, si fornirebbe, così, l'output dei due trasduttori concatenato. Questa soluzione, però, presenta un inconveniente: è necessario fornire in input il lemma duplicato, in quanto il trasduttore risultante dalla concatenazione riconosce solamente la concatenazione degli input dei trasduttori di partenza (in questo caso entrambi i trasduttori riconoscono il lemma, per cui è necessario fornire due volte il lemma stesso). Ciò potrebbe causare disagi all'utente durante l'utilizzazione del tool, per cui è stato modificato il codice sorgente di HFST-LOOKUP, per effettuare direttamente e contemporaneamente la generazione della forma e della struttura corrispondenti al lemma in input. Il codice è stato quindi modificato in modo da prendere in input non più un solo trasduttore, ma due, ed effettuare perciò il riconoscimento del lemma richiesto in entrambi i trasduttori in input. In questo modo è possibile ottenere la generazione di entrambe le informazioni, mediante un unico tool che fornisce in risposta l'output di entrambi i trasduttori (vedi figura 5.1).

Per l'operazione di analisi, invece, si sfrutta il tool HFST-COMPOSE, che permette di effettuare una composizione tra due trasduttori, in modo da utilizzare l'output del primo come input del secondo (vedi figura 5.2).

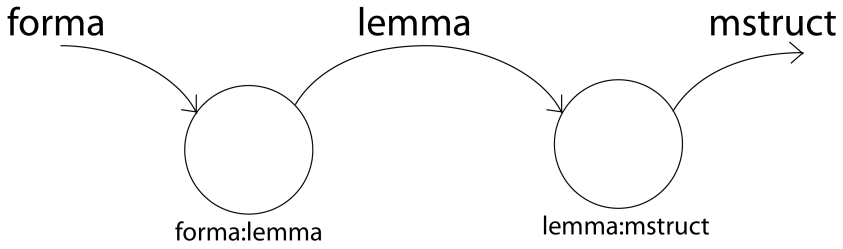

Figura 5.2: Composizione di due automi

A livello implementativo, la prima cosa da fare è la scrittura del trasduttore lemma:mstruct, su cui si baserà poi la creazione degli automi per la generazione e l'analisi.

Il trasduttore in oggetto è facilmente ottenibile a partire dal trasduttore lemma:forma. Risulta, infatti, avere la stessa struttura, ciò che cambia è il contenuto del lessico `root`. I morfemi iniziali della parola vengono modificati, aggiungendo i simboli separatori che indicano la struttura della parola stessa. Ciò che nel trasduttore lemma:forma viene indicato in questo modo:

```
l_riaddormentare:riaddorment+V_ARE SufVerbAre;
```

Nel trasduttore lemma:mstruct viene indicato con questo formato:

```
l_riaddormentare:ri%>ad%>dorment+V_ARE SufVerbAre;
```

Da notare l'uso del simbolo di *escape* %, per poter utilizzare il simbolo protetto >. Un ultimo dettaglio che è stato modificato all'interno del file .lexc, è il simbolo * che è stato sostituito con i caratteri - e <. Il primo viene utilizzato per evidenziare i suffissi flessivi, mentre il secondo è utilizzato nella

gestione dei suffissi di alterazione di sostantivi e aggettivi. Questa scelta è stata effettuata per mantenere una distinzione tra i due diversi tipi di suffisso.

```
LEXICON SufModMasc

    +SING+DIM:+SING+DIM<ino #;
    +PLUR+DIM:+PLUR+DIM<ini #;
    ...
    +SING+PEG:+SING+PEG<accio #;
    +PLUR+PEG:+PLUR+PEG<acci #;
```

La modifica richiesta, invece, per il file .twolc riguarda la necessità di evitare che i simboli, che vengono utilizzati per separare i morfemi (-, <, >), vengano trattati alla stregua di tutti i simboli usati per le varie caratteristiche delle forme lessicali: a differenza loro, infatti, questi simboli non vanno eliminati.

Per ottenere questo risultato è sufficiente, nella dichiarazione dell'alfabeto, definirili senza l'indicazione `:0`. Ciò che veniva, quindi, indicato in questo modo:

```
%*:0
```

Viene sostituito dalla seguente scrittura:

```
%- %< %>
```

Una volta implementato, il trasduttore lemma:mstruct viene compilato, analogamente al precedente, tramite il tool HFST-COMPOSE-INTERSECT. Una volta

ottenuto, esso può essere combinato, tramite i tool indicati precedentemente in questo capitolo, con il trasduttore lemma:forma.

Si ottengono così, partendo da un lemma, tutte le forme che corrispondono allo stesso, compresa l'indicazione della struttura; per il task dell'analisi, invece, si avrà come risultato il lemma di riferimento della forma e la struttura di quest'ultima.

Nell'applicazione bash che svolge la funzione di intermediario tra l'utente ed AnIta, presentata nel capitolo 3.2, viene gestita anche l'aggiunta di questa nuova *feature*, mstruct.

Eseguendo questo script per effettuare l'analisi di una forma, l'utente avrà, quindi, un output comprensivo dell'informazione derivata da mstruct:

```
cannone
cannone l_canna+NN+FEMM+PLUR+AUM cann<one
cannone l_cannone+NN+MASC+SING cann<on-e
```

Analogamente, per generare le forme relative ad un lemma, si otterrà il seguente risultato:

```
l_riaddormentare+V_FIN+IND+PRES+1+PLUR
l_riaddormentare+V_FIN+IND+PRES+1+PLUR riaddormentiamo
                                       ri>ad>dorment-iamo
```

Capitolo 6

Conclusioni

In questa tesi è stato descritto lo sviluppo di un software per l'analisi morfologica della lingua italiana. Questo progetto è nato dalla necessità di implementare uno strumento completamente open-source, che permettesse di gestire le caratteristiche morfologiche della lingua italiana.

Nella prima fase del lavoro per lo sviluppo di questo progetto, è stato effettuato uno studio approfondito della morfologia della lingua italiana, volto ad una completa comprensione del processo di formazione delle parole.

Sono stati studiati a fondo anche i principi che si trovano alla base della morfologia computazionale, basata sulle teorie degli automi a stati finiti, ed alcuni dei possibili strumenti a disposizione, per l'implementazione di quest'applicazione.

Una volta terminata la fase preliminare di progettazione, è stata eseguita l'effettiva implementazione di AnIta. Lo strumento utilizzato per il

suo sviluppo è HFST, un tool open-source per lo sviluppo di strumenti morfologici, distribuito dall'Università di Helsinki, che è risultato ottimale per quest'applicazione per le sue caratteristiche di efficienza, di libertà del suo codice sorgente e per il fatto che è un software recente, seguito ed aggiornato.

Sono state fornite, inoltre, le indicazioni per poter modificare il software sviluppato o per poter gestire l'inserimento di ulteriori informazioni (morfologiche, fonologiche, ...) relative alle parole riconosciute e generate dal lemmario.

Vi sono alcune caratteristiche che rendono interessante l'applicazione sviluppata in questo progetto:

- Il dizionario implementato possiede circa 120.000 lemmi, che permettono la corretta generazione ed il riconoscimento di più di due milioni di forme della lingua italiana; aspetto che rende l'applicazione effettivamente utilizzabile, in quanto capace di coprire gran parte della lingua, nel nostro caso dell'italiano.

- L'implementazione delle regole morfotattiche è stata effettuata in modo da rendere semplice per l'utente l'inserimento di nuovi lemmi, che richiedono, infatti, la scrittura di poche righe, per ognuno di essi.

- La gestione delle varie informazioni, abbinate ad ogni parola generata da AnIta, è facilmente ottenibile, in quanto ogni nuova informazione da gestire viene racchiusa in un trasduttore separato e autonomo, che

viene, poi, combinato con il lemmario di partenza, con le modalità illustrate nel capitolo 5 di questa tesi.

Infine, analizzando quello che potrebbe rappresentare un miglioramento di questa applicazione, vengono indicati alcuni suggerimenti che potrebbero portare a possibili sviluppi futuri:

- Essendo quello presentato in questa tesi un tool sviluppato per la gestione della lingua italiana, una lingua viva, l'ampliamento del lessico gestito, oltre che suggerire uno sviluppo futuro, è da considerarsi un'operazione di vera e propria manutenzione, per mantenere l'applicazione attuale e aggiornata. Tra i possibili sviluppi futuri in questo ambito, è ipotizzabile un ampliamento del lemmario con un insieme di termini specifici di una materia (ad es. aggiunta di termini giuridici, per l'analisi di testi appartenenti al campo della giurisprudenza).

- Un ulteriore possibile sviluppo è quello determinato dall'aggiunta di altre informazioni (ad es. di tipo fonologico) da associare alle parole generate da AnIta, in modo da renderne possibile un utilizzo in più campi di applicazione.

Appendice A

Tutorial

In questa appendice verranno illustrati i comandi di utilizzo di AnIta e verranno elencati i componenti di quest'applicazione.

Installazione

Il software HFST è disponibile per le piattaforme Unix e MacOS ed è scaricabile gratuitamente dal sito:

```
http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/
```

Per installarlo si utilizzano gli usuali comandi:

```
$ ./configure
$ make
$ make install
```

Compilazione del lessico

Una volta scritti i file del lessico e delle regole, che qui verranno chiamati italiano.lexc ed italiano.twolc, si utilizzano i seguenti comandi per ottenere i trasduttori del lessico (italiano.hfst) e delle regole (regole.hfst).

- L'istruzione che segue permette di creare il trasduttore italiano.hfst. L'opzione `-v` (verbose) fornisce informazioni sul processo di compilazione e risulta utile in fase di *debugging*. Per indicare il nome del file di output si utilizza l'opzione `-o`:

  ```
  $ hfst-lexc -o italiano.hfst italiano.lexc -v
  ```

- Analogamente, il seguente comando fornisce in output il trasduttore delle regole regole.hfst:

  ```
  $ hfst-twolc -o regole.hfst italiano.twolc -v
  ```

- Utilizzando questa istruzione, si ottiene il trasduttore che permette di generare una forma a partire da un lemma e dalle caratteristiche lessicali richieste. Per distinguere il trasduttore delle regole dal trasduttore del lessico, quest'ultimo viene indicato tramite l'opzione `-l`:

  ```
  $ hfst-compose-intersect -l italiano.hfst -o generatore.hfst
        regole.hfst
  ```

- Una volta creato il file del generatore, tramite il seguente comando si ottiene il trasduttore che effettua il task dell'analisi:

```
$ hfst-invert -o analizzatore.hfst generatore.hfst
```

In maniera analoga ai comandi appena descritti, è possibile compilare i trasduttori che gestiscono le informazioni aggiuntive (features), partendo dai file mstruct.lexc (che contiene il lessico) e mstruct.twolc (che contiene le regole di riaggiustamento):

```
$ hfst-lexc -o mstruct.hfst mstruct.lexc -v
$ hfst-twolc -o regolemstruct.hfst mstruct.twolc -v
$ hfst-compose-intersect -l mstruct.hfst -o gen_mstruct.hfst
       regolemstruct.hfst
```

Per comporre il trasduttore forma:lemma (analizzatore.hfst) ed il trasduttore lemma:mstruct (gen_mstruct.hfst) per ottenere un trasduttore che, presa una forma lessicale, ne restituisce la struttura, si sfrutta il seguente comando:

```
$ hfst-compose -o forma-mstruct.hfst -1 analizzatore.hfst
       -2 gen_mstruct.hfst -v
```

Per concatenare, invece, il trasduttore lemma:forma (generatore.hfst) ed il trasduttore lemma:mstruct (gen_mstruct.hfst) per ottenere un trasduttore che preso un lemma con le caratteristiche richieste, restituisca la forma e la sua struttura, si utilizza la seguente istruzione:

```
hfst-concatenate -o lemma-mstruct.hfst -1 generatore.hfst
     -2 Mstruct/gen_mstruct.hfst -v
```

Utilizzazione dell'analizzatore/generatore

Per analizzare una forma lessicale, sfruttando il trasduttore analizzatore.hfst, si utilizza il tool HFST-LOOKUP, nel seguente modo:

```
$ hfst-lookup analizzatore.hfst
> porta
porta l_porta+NN+FEMM+SING
porta l_portare+V_FIN+IND+PRES+3+SING
porta l_portare+V_FIN+IMP+PRES+2+SING
porta l_porgere+V_PP+PART+PAST+FEMM+SING
```

In maniera analoga, per generare una forma lessicale da un lemma si utilizza il seguente comando:

```
$ hfst-lookup generatore.hfst
> l_portare+V_FIN+IND+PRES+3+SING
l_portare+V_FIN+IND+PRES+3+SING porta
```

Al tool Hfst-lookup è possibile passare in input anche un file testuale che contiene una lista di interrogazioni. Si consideri di avere a disposizione un file prova.txt, contenente le stringhe:

```
medicina
portiere
saldamente
```

Il risultato dell'analisi sarà il seguente:

```
$ hfst-lookup -I prova.txt analizzatore.hfst
medicina l_medicina+NN+FEMM+SING
medicina l_medicinare+V_FIN+IND+PRES+3+SING
medicina l_medicinare+V_FIN+IMP+PRES+2+SING

portiere l_portiera+NN+FEMM+PLUR
portiere l_portiere+NN+MASC+SING

saldamente l_saldamente+ADV
```

Per effettuare, invece, operazioni di interrogazione ai trasduttori che gestiscono l'informazione aggiuntiva denominata mstruct, si consiglia l'utilizzo dello script bash AnIta.sh, nel seguente modo:

```
$ ./AnIta.sh

    Benvenuto su AnIta
    Digita A per l'analizzatore
    Digita G per il generatore
    Digita EXIT per uscire
```

```
> A
AnIta - Analizzatore
> cannone
cannone l_canna+NN+FEMM+PLUR+AUM cann-one
cannone l_cannone+NN+MASC+SING cann<on-e

> EXIT
> G
AnIta - Generatore
> l_vendere+V_FIN+IND+PAST+1+SING

l_vendere+V_FIN+IND+PAST+1+SING vendei vend-ei
l_vendere+V_FIN+IND+PAST+1+SING vendetti vend-etti
```

Scrittura del lessico

In questa sezione dell'appendice verranno elencati tutti i simboli multicarattere utilizzati in questo progetto e tutte le classi di continuazione create per gestire le regole morfotattiche.

Simboli multicarattere

Di seguito vengono elencati tutti i *tag* da assegnare alle forme lessicali riconosciute e generate da AnIta.

- `+NN`, `+NN_P`: indicano rispettivamente che la forma è un nome comune o un nome proprio.
- `+ADJ`: indica che la forma è un aggettivo.
- `+ADJ_DIM`, `+ADJ_IND`, `+ADJ_IES`, `+ADJ_NUM`, `+ADJ_POS`: indicano che la forma è un aggettivo specificando il suo tipo. Rispettivamente abbiamo aggettivi dimostrativi, indefiniti, interrogativi/esclamativi, numerali e possessivi.
- `+PRON`: indica che la forma è un pronome.
- `+PRON_DIM`, `+PRON_IND`, `+PRON_IES`, `+PRON_REL`, `+PRON_POS`, `+PRON_PER`: indicano che la forma è un pronome specificando il suo tipo. Rispettivamente abbiamo pronomi dimostrativi, indefiniti, interrogativi/esclamativi, relativi, possessivi e personali.
- `+ADV`: indica che la forma è un avverbio.
- `+ART`: indica che la forma è un articolo.
- `+INT`: indica che la forma è un'interiezione.
- `+PREP`, `+PREP_A`: indicano rispettivamente che la forma è un nome comune o un nome proprio.

- `+CONJ`, `+CONJ_S`: indicano rispettivamente che la forma è un congiunzione o un congiunzione articolata.

- `+V_ARE`, `+V_ERE`, `+V_IRE`: indicano che la forma è un verbo e ne specificano la sua declinazione.

- `+V_FIN`: indica che la forma è un verbo coniugato in un modo finito (indicativo, congiuntivo, condizionale, imperativo).

- `+V_NOFIN`: indica che la forma è un verbo coniugato in un modo indefinito, escluso il participio che viene gestito singolarmente (infinito, gerundio).

- `+V_PP`: indica che la forma è un verbo coniugato al participio, presente o passato.

- `+MASC`, `+FEMM`: indica il genere della forma (maschile o femminile).

- `+SING`, `+PLUR`: indica il numero della forma (singolare o plurale).

- `+DIM`, `+AUM`, `+VEZ`, `+PEG`, `+SUP`: indicano la presenza di suffissi valutativi nella forma. Rispettivamente abbiamo diminutivi, accrescitivi, vezzeggiativi, peggiorativi e superlativi.

- `+IND`, `+SUBJ`, `+COND`, `+IMP`, `+INF`, `+GER`, `+PART`: indicano il modo in cui è coniugata una forma verbale. Rispettivamente abbiamo indicativo, congiuntivo, condizionale, imperativo, infinito, gerundio e participio.

- `+PRES`, `+IMPERF`, `+PAST`, `+FUT`: indicano il tempo in cui è coniugata una forma verbale. Rispettivamente abbiamo presente, imperfetto, passato e futuro.

- `+1`, `+2`, `+3`: indicano la persona in cui è coniugata una forma verbale.

- `+GLI`: indica che la forma non ammette il raddoppio dell'ultima *i* della radice nel processo di flessione. Non compare nell'output del trasduttore, ma serve per poter applicare correttamente le regole grafotattiche.

- `+VEL_A`, `+VEL_Y`: indicano che la forma necessita dell'inserimento di un'*h* tra la radice ed il suffisso. Il primo si utilizza per le forme non verbali in cui la radice termina per *c* o *g* ed il suffisso inizia per *e* (sovietic-e), mentre il secondo si usa in tutte le forme in cui la radice termina per *c* o *g* ed il suffisso inizia per e oppure *i* (cuoc-i). Non compare nell'output del trasduttore, ma serve per poter applicare correttamente le regole grafotattiche.

Classi di continuazione

Di seguito vengono elencate tutte le classi di continuazione utilizzate per definire le regole morfotattiche.

- `SufVerbAre`, `SufVerbEre`, `SufVerbIre`: gestiscono la combinazione delle basi dei verbi regolari della prima, seconda e terza coniugazione con le relative desinenze.

- `r1Are`, `r1Ere`, `r1Ire`: contengono le desinenze da abbinare alle radici di tipo r1 dei verbi irregolari della prima, seconda e terza coniugazione.

- `r1EreNoInf`: contiene le desinenze da abbinare alle radici di tipo r1 dei verbi irregolari della seconda coniugazione, tranne quella relativa all'infinito. È utilizzato per quei verbi che presentano la forma all'infinito troncata.

- `r1EreBlock1`: contiene alcune desinenze da abbinare alle radici di tipo r1 dei verbi irregolari della seconda coniugazione. È usato per i verbi che ammettono solo le seguenti forme per la radice r1: la prima e seconda persona plurale del congiuntivo presente, la prima persona plurale dell'indicativo presente e per la prima persona plurale dell'imperativo presente. Un esempio è il verbo dispiacere nella radice dispiacc-.

- `r1EreBlock3`: contiene tutte e sole le desinenze relative alla terza persona, da abbinare alle radici di tipo r1 dei verbi irregolari della seconda coniugazione. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r2Are`, `r2Ere`, `r2Ire`: contengono le desinenze da abbinare alle radici di tipo r2 dei verbi irregolari della prima, seconda e terza coniugazione.

- `r2EreBlock3`: contiene tutte e sole le desinenze relative alla terza persona, da abbinare alle radici di tipo r2 dei verbi irregolari della seconda

coniugazione. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r3Ere`, `r3Ire`: contengono le desinenze da abbinare alle radici di tipo r3 dei verbi irregolari della seconda e terza coniugazione.

- `r3EreBlock3`: contiene tutte e sole le desinenze relative alla terza persona, da abbinare alle radici di tipo r3 dei verbi irregolari della seconda coniugazione. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r4Are`, `r4Ere`, `r4Ire`: contengono le desinenze da abbinare alle radici di tipo r4 dei verbi irregolari della prima, seconda e terza coniugazione.

- `r4EreBlock3`: contiene tutte e sole le desinenze relative alla terza persona, da abbinare alle radici di tipo r4 dei verbi irregolari della seconda coniugazione. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r5Are`, `r5Ere`, `r5Ire`: contengono le desinenze da abbinare alle radici di tipo r5 dei verbi irregolari della prima, seconda e terza coniugazione.

- `r5EreBlock3`: contiene tutte e sole le desinenze relative alla terza persona, da abbinare alle radici di tipo r5 dei verbi irregolari della seconda coniugazione. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r6Are`, `r6Ere`, `r6Ire`: contengono le desinenze da abbinare alle radici di tipo r6 dei verbi irregolari della prima, seconda e terza coniugazione.

- `rQ`: contiene le desinenze alternative (-etti, -ette, -ettero) per i verbi della seconda coniugazione che, nella prima e terza persona singolare e nella terza persona plurale del passato remoto, ammettono due forme verbali diverse.

- `rQBlock3`: contiene le stesse desinenze della classe rQ, limitate alla terza persona. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r4`: contiene le desinenze per la gestione del passato remoto dei verbi altamente irregolari, ovvero quelli che subiscono un processo di troncamento in alcune desinenze.

- `r4Block3`: contiene le desinenze per la gestione del passato remoto dei verbi altamente irregolari, ovvero quelli che subiscono un processo di troncamento in alcune desinenze. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r5`: contiene le desinenze per la gestione del futuro e del condizionale presente dei verbi altamente irregolari, ovvero quelli che subiscono un processo di troncamento in alcune desinenze.

- `r5Block3`: contiene le desinenze per la gestione del futuro e del condizionale presente dei verbi altamente irregolari, ovvero quelli che subis-

cono un processo di troncamento in alcune desinenze. È utilizzata per quei verbi che ammettono solo forme alla terza persona singolare e plurale.

- `r6`: contiene le desinenze per la gestione del participio passato dei verbi altamente irregolari, ovvero quelli che subiscono un processo di troncamento in alcune desinenze.

- `r7`: contiene le desinenze per la gestione dell'infinito dei verbi altamente irregolari, ovvero quelli che subiscono un processo di troncamento in alcune desinenze.

- `SufVerbEreNoPP`: contiene le desinenze dei verbi della seconda coniugazione che presentano un'irregolarità solo nel participio passato, che viene gestito a parte. Contiene, quindi, tutte le desinenze della coniugazione regolare, tranne quella relativa al participio passato.

- `SufVerbAreBlock3`, `SufVerbEreBlock3`, `SufVerbIreBlock3`: contengono le desinenze delle tre coniugazioni regolari per quei verbi che ammettono solo forme alla terza persona singolare o plurale.

- `SufVerbAreInf`, `SufVerbEreInf`, `SufVerbIreInf`: contengono le desinenze del modo infinito dei verbi nelle tre coniugazioni.

- `SufVerbAreGer`, `SufVerbEreGer`, `SufVerbIreGer`: contengono le desinenze del modo gerundio dei verbi nelle tre coniugazioni.

- `SufVerbArePPres`, `SufVerbErePPres`, `SufVerbIrePPres`: contengono le desinenze del participio presente dei verbi nelle tre coniugazioni.

- `SufNomMascA`, `SufNomMascE`, `SufNomMascO`, `SufNomFemmA`, `SufNomFemmE`, `SufNomFemmO`: contengono i suffissi per i sostantivi maschili e femminili terminanti in *a*, *e*, *o* e per tutti gli aggettivi il cui comportamento è assimilabile a quello dei sostantivi. Da queste classi di continuazione vengono generati anche tutti i suffissi valutativi accordati nel genere.

- `SufNomMascAr1`, `SufNomMascEr1`, `SufNomMascOr1`, `SufNomFemmAr1`, `SufNomFemmEr1`, `SufNomFemmOr1`: contengono i suffissi per i sostantivi maschili e femminili terminanti in *a*, *e*, *o* e per tutti gli aggettivi il cui comportamento è assimilabile a quello dei sostantivi, che ammettono solo la forma singolare. Da queste classi di continuazione vengono generati anche tutti i suffissi valutativi accordati nel genere.

- `SufNomMascAr2`, `SufNomMascEr2`, `SufNomMascOr2`, `SufNomFemmAr2`, `SufNomFemmEr2`, `SufNomFemmOr2`: contengono i suffissi per i sostantivi maschili e femminili terminanti in *a*, *e*, *o* e per tutti gli aggettivi il cui comportamento è assimilabile a quello dei sostantivi e che ammettono solo la forma plurale. Da queste classi di continuazione vengono generati anche tutti i suffissi valutativi accordati nel genere.

- `SufNomIrr`: contiene i suffissi per i sostantivi che ammettono una forma singolare e due plurali.

- `SufNomIrrNomp`: contiene i suffissi per i sostantivi che ammettono solamente una forma singolare maschile ed una plurale femminile.

- `SufNomMascANoMod`, `SufNomMascENoMod`, `SufNomMascONoMod`, `SufNomFemmANoMod`, `SufNomFemmENoMod`, `SufNomFemmONoMod`, `SufNomMascAr1NoMod`, `SufNomMascEr1NoMod`, `SufNomMascOr1NoMod`, `SufNomFemmAr1NoMod`, `SufNomFemmEr1NoMod`, `SufNomFemmOr1NoMod`, `SufNomMascAr2NoMod`, `SufNomMascEr2NoMod`, `SufNomMascOr2NoMod`, `SufNomFemmAr2NoMod`, `SufNomFemmEr2NoMod`, `SufNomFemmOr2NoMod`: in queste classi sono contenuti i suffissi per i sostantivi e aggettivi analoghi a quelli gestiti nelle classi appena mostrate, con la differenza che questa classe non genera i suffissi valutativi. Questa classe è utilizzata anche per tutte quelle particelle del discorso che si comportano come i sostantivi ma che non ammettono i valutativi (aggettivi, pronomi,...).

- `SufAggA`, `SufAggE`, `SufAggO`: contengono i suffissi per gli aggettivi che terminano in *a*, *e*, *o*. Vengono generati tutti i suffissi valutativi, gli aggettivi superlativi in -issimo e l'avverbio in -issimamente.

- `SufNomA`, `SufNomE`, `SufNomO`: contengono i suffissi per i sostantivi che terminano in *a*, *e*, *o*. In questa classe vengono generati solamente i suffissi valutativi.

- `SufAggOr1`: contiene i suffissi per la radice di tipo r1 per gli aggettivi irregolari.

- `SufAggOr2`: contiene i suffissi per la radice di tipo r2 per gli aggettivi irregolari.

- `SufAggIrr`: contiene i suffissi per la gestione di aggettivi terminanti in *e* che mostrano un'irregolarità nelle desinenze (ad es. cassier-e).

- `SufAggIrrxNom`: contiene i suffissi per la gestione di sostantivi terminanti in *e* che mostrano un'irregolarità nelle desinenze (ad es. cassier-e).

- `SufAggANoMod`, `SufAggENoMod`, `SufAggONoMod`: contengono i suffissi per gli aggettivi terminanti in *a*, *e*, *o* che non ammettono i suffissi valutativi.

- `SufAA`, `SufAE`, `SufAO`: contengono i suffissi per aggettivi, pronomi e avverbi terminanti in *a*, *e*, *o* che non ammettono né i valutativi, né nessun altro tipo di suffisso.

- `SufModMasc`, `SufModFemm`: contengono i suffissi valutativi per i generi maschile e femminile.

- `SufIssimo`: contiene i suffissimi per generare gli aggettivi superlativi.

Bibliografia

[1] *PC-KIMMO: a two-level processor for morphological analysis.* Evan L. Antworth. Occasional Publications in Academic Computing No. 16. Dallas, TX: Summer Institute of Linguistics, 1990.

[2] *User's Guide to PC-KIMMO version 2.* Evan L. Atworth, 1995. URL: http://www.sil.org/pckimmo/v2/doc/guide.html

[3] *Una piattaforma di morfologia computazionale per l'analisi e la generazione delle parole italiane.* Marco Battista, Vito Pirrelli. Technical Report Magic 1.0, 1999.

[4] *KIMMO: a general morphological processor.* Lauri Karttunen. *In Texas Linguistic Forum, vol. 22*, Dalrymple, Doron, Goggin, Goodman and McCarthy. University of Texas, Department of Linguistics, Austin, TX, 1983.

[5] *A compiler for two-level phonological rules.* Lauri Karttunen, Kimmo Koskenniemi and Ronald M. Kaplan. Technical re-

port, Xerox Palo Alto Research Center and Center for the Study of Language and Information, Stanford University, June 1987.

[6] *Two-Level Rule Compiler.* Lauri Karttunen and Kenneth R. Beesley. Technical Report, Xerox Palo Alto Research Center. Palo Alto, CA, October 1992.

[7] *Finite-State Lexicon Compiler.* Lauri Karttunen. Technical Report, Xerox Palo Alto Research Center. Palo Alto, CA, April 1993.

[8] *Constructing Lexical Transducers.* Lauri Karttunen. In Proceedings of the 15th International Conference on Computational Linguistics COLING 94, I, 406-411, 1994.

[9] *Applications of Finite-State Transducers in Natural Language Processing.* Lauri Karttunen. Xerox Research Centre Europe. Proceedings of CIAA-2000. Lecture Notes in Computer Science. Springer Verlag.

[10] *A Short History of Two-Level Morphology.* Lauri Karttunen, Xerox Palo Alto Research Center, Kenneth R. Beesley, Xerox Research Centre Europe. September 2001.

[11] *Two-level morphology: A general computational model for word-form recognition and production.* Kimmo Koskenniemi,

University of Helsinki, Department of General Linguistics, Helsinki 1983.

[12] *HFST Tools for Morphology - An Efficient Open-Source Package for Construction of Morphological Analyzers.* Krister Lindén, Miikka Silfverberg, Tommi Pirinen. University of Helsinki, Department of General Linguistics, Helsinki. In Proceedings of the Workshop on Systems and Frameworks for Computational Morphology 2009. September 2009, Zurich.

[13] *Monotonic Paradigmatic Schemata in Italian Verb Inflection.* Vito Pirrelli, Marco Battista. COLING '96 Proceedings of the 16th conference on Computational linguistics. 1996, ILC-CNR, Pisa.

[14] *Techniques in Natural Language Processing.* Graeme Ritchie, Chris Mellish. Course Notes, Autumn 2000.

[15] *La struttura delle parole.* Sergio Scalise, Antonietta Bisetto. Il Mulino, 2008.