Uli Bubenheimer
YALL — Yet Another Latin Lemmatizer:
A Morphological Analyzer for Latin and
A Reimplementation in Java
(Under the direction of Michael A. Covington)

This thesis describes a morphological analyzer for Latin and its reimplementation in the Java programming language. The analyzer is divided into two parts, the analysis of endings and a lexicon. Both components rely on tries (character–trees), data structures for efficient lexical lookup. For the analysis of endings, a directed acyclic graph (DAG) is used in place of the usual tree structure. The tries are complemented by feature structures that include disjunction and negation. Graph unification for feature structures is used as a main processing mechanism. The deficits of the analyzer's previous implementation in the C++ programming language are pointed out, a comparison of C++ and Java is undertaken, and a description of the new implementation is given.

Index words: Latin, morphological analyzer, inflectional morphology, lemmatizer, trie, character–tree, letter–tree, Java, C++

YALL — Yet Another Latin Lemmatizer

by

Uli Bubenheimer

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

1999

YALL — Yet Another Latin Lemmatizer

by

Uli Bubenheimer

Approved:

____

Major Professor

____

Date

Approved:

____

Graduate Dean

____

Date

Acknowledgements

Many thanks go to Michael Covington for his ongoing support and assistance. In addition, I want to thank Donald Nute and Rick LaFleur for their time and effort with this thesis. I am also grateful to the Artificial Intelligence Center at The University of Georgia for providing a stimulating research environment.

Contents

Acknowledgements iv

1 Introduction 1

2 Theory 3

2.1 Introduction 3

2.2 Analysis of endings 4

2.3 Lexicon 11

2.4 Summary 20

3 Java versus C++ 21

3.1 Problems of the original implementation 21

3.2 Some advantages of Java over C++ 24

3.3 The reimplementation process 33

3.4 New and improved program features 34

4 Conclusions and Outlook 45

4.1 Theoretical refinement 45

4.2 Practical refinement and reengineering 46

4.3 Future extensions 46

Bibliography 48

Appendices

A Unification with Negation and Disjunction 51

B A Feature System for Latin Morphology 53

C More Examples of Standard Lexical Entries 58

D Inflection Tables 60

D.1 Verbs 61

D.2 Nouns 65

D.3 Adjectives 66

E Endings Trie 67

F Source Code 72

G YALL Class Documentation 135

G.1 Overview 135

G.2 Package `yall` 136

G.3 Class `Yall` 138

Chapter 1

Introduction

This thesis describes YALL (Yet Another Latin Lemmatizer), a morphological analyzer and lemmatizer for Latin and its reimplementation in Java. Part of this thesis describes why the Java implementation is superior to an earlier implementation in C++ and summarizes the experiences made in the reimplementation process; it also includes a comparison of Java and C++ features from a software development point of view. In addition, the thesis has served as an opportunity for me to widen my knowledge and to learn Java, which has been widely talked about for a number of years. Java has a number of interesting features like its combination of traditional imperative programming with object–orientation, interpreted execution, platform–independence, and Internet accessibility. At the same time, Java is supposed to be similar to C++, which makes an investigation into their differences interesting.

Work on YALL started in the summer of 1994, leading to the first implementation in C++ described in Bubenheimer (1995). Since then, YALL has been refined, leading to an improved theoretical basis and a better approach to its implementation (through the stronger theory and by learning from mistakes made in the previous implementation). Most of this thesis describes the results of the refinement, with additional supplements translated from Bubenheimer (1995). One of the basic ideas behind YALL, namely to approach Latin morphology with tries (character–trees, letter–trees) was also discovered independently by Covington at about the same time as my original implementation took place (see Covington, 1999).

This thesis does not focus on how Latin morphology can be described in terms of the theory to be presented. This task has been done earlier and the result can be found in Bubenheimer (1995). However, the theory will be illustrated by extensive examples in Chapter 2 and Appendices B, C, D, and E. These appendices are largely based on German materials from Bubenheimer (1995), and represent an introduction in English to how exactly Latin morphology is described in terms of the theory.

Chapter 2 describes the present theory of YALL. Basic knowledge of Latin morphology will be helpful to understand this chapter. Chapter 3 analyzes some problems with the previous implementation, compares differences of Java and C++, and describes the reimplementation in Java. The chapter presupposes basic knowledge of programming languages, object–oriented programming concepts, and of the C++ and Java programming languages. The final Chapter 4 attempts an evaluation of the present state of development and possible future changes and improvements. Several appendices follow which primarily give additional examples and elaborations of concepts described earlier.

Chapter 2

Theory

2.1 Introduction

The morphological analyzer described in this thesis analyzes inflected Latin word forms and returns a description of a form's morphological properties and provides the word's base form, the lemma. Only inflectional word structure is analyzed, not derivational structure. Since the program deals with written Latin, vowel lengths are ignored. A sample run of the analyzer is given in Section 3.4.4, page 42.

To combine the goals of a linguistically sound morphological analysis and efficient execution, *tries* (also known as *character–trees* or *letter–trees*) are used as the main data structure to store and process both the system of Latin endings and the program's lexicon.

The morphological analysis is divided into two parts. The *analysis of endings* (Section 2.2) analyzes and strips off a word's inflectional ending. The remaining word stem is then looked up in the *lexicon* (Section 2.3). Typically, the analysis of endings, which does not have access to the lexicon, produces a great number of analyses, many of which are only hypothetical and rely on non–existent vocabulary. Lexical lookup then filters out those analyses that are legitimized by the existence of appropriate word stems and matching morphological features.

2.2 Analysis of endings

2.2.1 Features

Latin is a richly inflected language. At the same time, Latin inflection is almost exclusively restricted to the end of words and shows very regular inflectional patterns. For example, the final -m in silv-a-m (from silva, *forest*) and fili-u-m (from filius, *son*) indicates accusative case and singular number. -a- and -u- indicate *a–declension* (also named *first declension*) and *o–declension* (or *second declension*), respectively. Furthermore, nouns of a–declension can only have masculine or feminine gender, while there are examples for nouns in every gender for o–declension (Allen and Greenough, 1903).

To represent these insights formally, *features* in the form of *attribute–value pairs* are used (Johnson, 1991). For the task at hand, attributes represent morphological categories, while values represent elements from their corresponding categories. Appendix B describes the system of features used by the morphological analyzer.

Attribute–value structures (*feature structures*) are sets of attribute–value pairs. To represent the above examples, appropriate feature structures are:

$$\begin{bmatrix} case : acc \\ num : sg \end{bmatrix} \qquad \text{(for -m)}$$

$$\begin{bmatrix} decl : o \\ gen : \begin{Bmatrix} masc \\ fem \\ neutr \end{Bmatrix} \end{bmatrix} \qquad \text{(for -u-)}$$

$$\begin{bmatrix} decl : a \\ gen : \begin{Bmatrix} masc \\ fem \end{Bmatrix} \end{bmatrix} \qquad \text{(for -a-)}$$

The curly brackets in the second and the third example mark disjunctions of feature values. Such disjunctions can be expanded distributively into equivalent disjunctions of whole feature structures. This is demonstrated by the following rewriting of the last feature structure from above, where ∨ denotes logical "or." Expanded feature structures like these will be used again in Section 2.3.3.

$$\begin{bmatrix} decl : a \\ gen : masc \end{bmatrix} \vee \begin{bmatrix} decl : a \\ gen : fem \end{bmatrix}$$

As can be seen from a comparison of the expanded with its unexpanded form, attribute–value structures with disjunctions of feature values allow to keep the representation more compact and make it faster to process.

2.2.2 Unification

To combine features from different parts of the ending like -a- and -m in silv-a-m, *unification*, or more specifically, *graph unification* for attribute-value structures can be used (Johnson, 1991; Karttunen, 1984). For feature structures with disjunctions of feature values and only constants as elements of those disjunctions, unification is easy to sketch. To facilitate the following explanations, a single feature value will now more generally be considered a disjunction with a single element.

Two feature structures A, B are unified ($A \sqcup B$) in the following way, resulting, if successful, in a new feature structure:

- If an attribute occurs in exactly one of A, B, add it to the result, together with its value disjunction.

- If an attribute occurs in both A and B, compute the intersection of its two value disjunctions from A and B, and add the attribute to the result, together with the value intersection. If the intersection is empty, unification fails.

For example,

$$\begin{bmatrix} \mathit{decl} : o \\ \mathit{num} : \mathit{sg} \\ \mathit{gen} : \begin{Bmatrix} \mathit{masc} \\ \mathit{fem} \end{Bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \mathit{decl} : \begin{Bmatrix} a \\ o \end{Bmatrix} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \begin{Bmatrix} \mathit{masc} \\ \mathit{neutr} \end{Bmatrix} \end{bmatrix} = \begin{bmatrix} \mathit{decl} : o \\ \mathit{case} : \mathit{acc} \\ \mathit{num} : \mathit{sg} \\ \mathit{gen} : \mathit{masc} \end{bmatrix}$$

2.2.3 Tries

Tries (Knuth, 1973; Sproat, 1992) are one of the main concepts of the morphological analyzer. Tries are commonly described as trees (in a graph–theoretical sense) whose non–root nodes are annotated with letters. A trie stores a set of words which are represented in the paths from the trie's root node to its leaf nodes (Figure 2.1). One

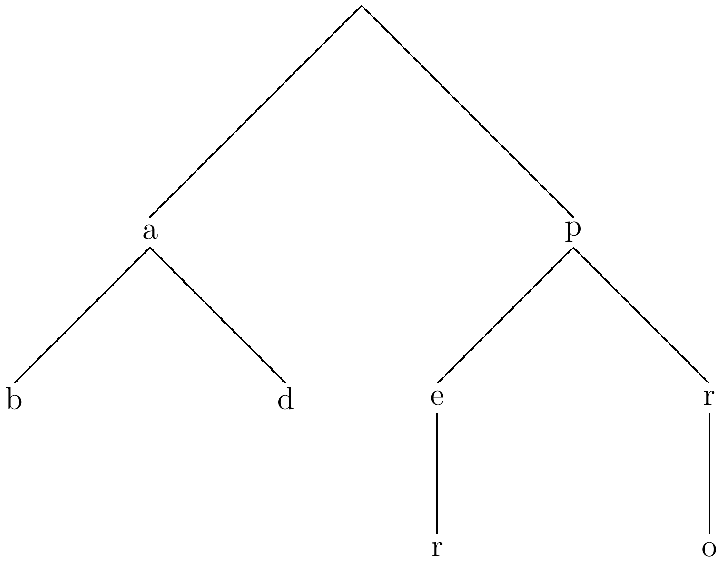


Figure 2.1: A simple trie representing the Latin prepositions ab (*from*), ad (*to*), per (*through*), and pro (*for*). Every path from the root node to a leaf node represents a word.

The second generalization is that nodes are annotated with strings of characters, not just single characters. This makes the representation of Latin endings more meaningful. The idea is that a node's feature structure approximately represents the morphological "meaning" of the node's string (or *morph*). The bottom half of Figure 2.3 shows good examples.

An additional goal in the development of the endings trie was not only to design a trie that accurately reflects endings, but which does so in a manner that captures the regularities of Latin endings and contains as little redundancy as possible. To this end, several nodes (annotated with empty character strings, ϵ) only serve to represent certain morphological features, and the trie's arcs define how they relate to other nodes, morphs, and features. For example, nodes 98 through 104 in Figure 2.2 ensure valid gender for nouns and adjectives; e.g. for a–declension (node 108), adjectives can only have feminine gender, while nouns can have feminine or masculine gender.

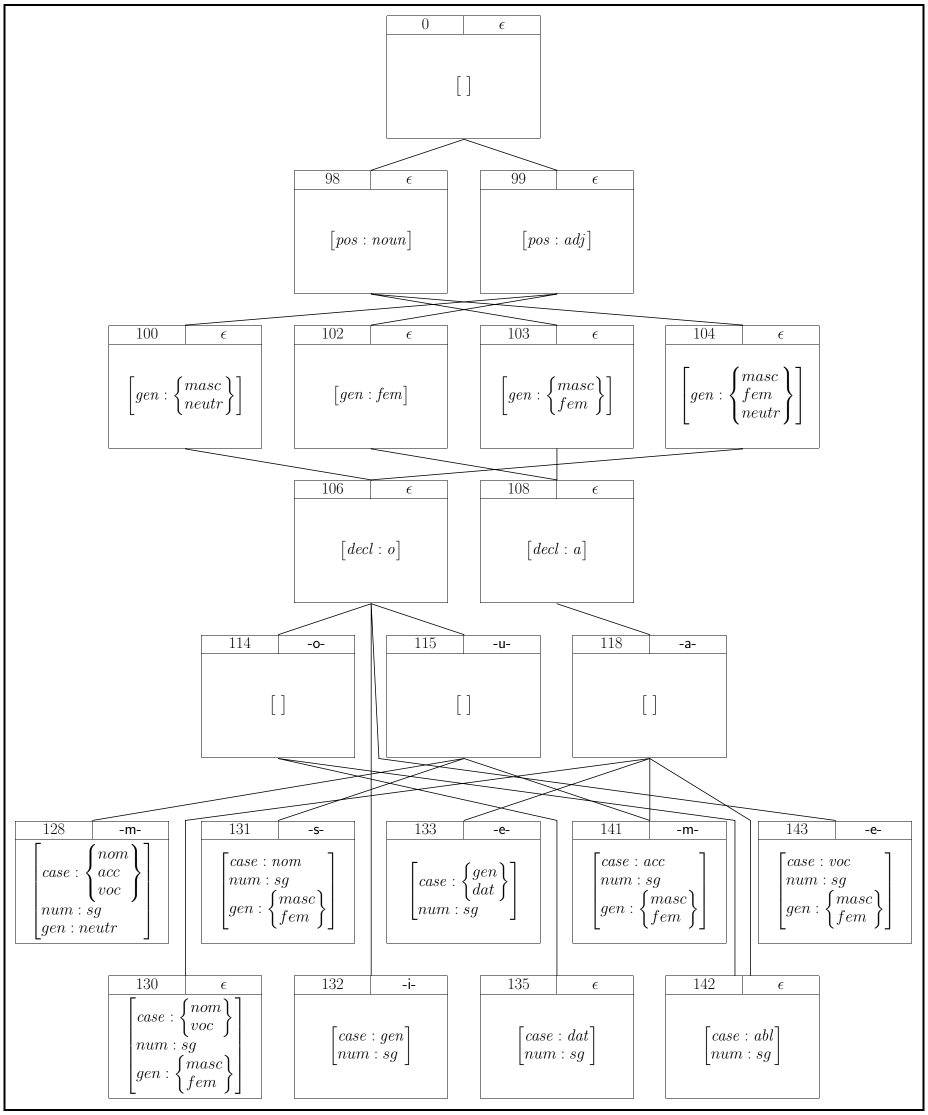


Figure 2.2: A trie representing the regular singular declensional endings of nouns and adjectives from a–declension (first declension) and o–declension (second declension). It is a detail of the complete analysis trie. Each node contains a character string, a feature structure, and a reference number (for documentation purposes here). The character string can be empty (marked by ϵ).

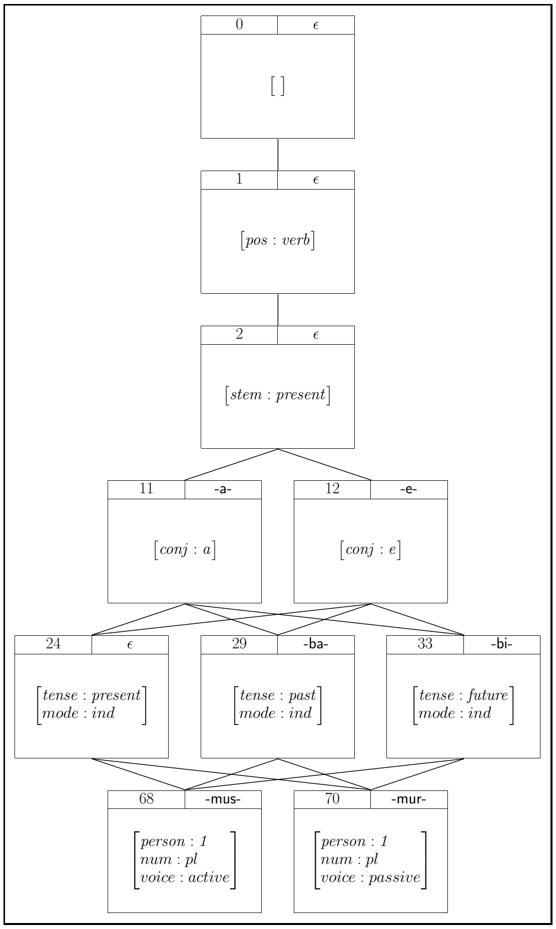

0 ϵ
[ ]
1 ϵ
[pos : verb]
2 ϵ
[stem : present]
11 -a-
[conj : a]
12 -e-
[conj : e]
24 ϵ
[tense : present mode : ind]
29 -ba-
[tense : past mode : ind]
33 -bi-
[tense : future mode : ind]
68 -mus-
[person : 1 num : pl voice : active]
70 -mur-
[person : 1 num : pl voice : passive]

Figure 2.3: Representation of first person plural in the present stem tenses for a– and e–conjugation.

However, empty character strings can also designate actual manifestations of morphological structure (often called *null morphs*) which are expressed by the absence of a morph on the surface. An example is node 24 in Figure 2.3.

2.2.4 Algorithm

The program analyzes word endings from right to left. Consequently, the endings trie is traversed along its arcs from the leaf nodes at the bottom toward the root node at the top. The ending substrings of the nodes are matched with the analyzed word's ending from right to left in a linear manner. If a node's string does not match the word ending in the right place, an analysis using the path that led to this node from a leaf node is not possible. An analysis is accepted if it reaches the root node. In general, several alternative analyses (i.e., alternative paths) are found for an ending.

While the trie is traversed, the feature structures from the visited nodes are unified with each other, providing one single, unified feature structure per analysis. If unification fails on some node, then an analysis along the present trie path is not possible, similarly to a mismatch of node substring and ending substring described above.

For a valid analysis, the part of the word that is matched in the trie is the word's ending, the unmatched part is its stem. The stem, together with the analysis' feature structure, is subsequently matched against the lexicon to eliminate invalid analyses (cf. Section 2.3).

As an example, consider the previously mentioned silvam. One successful analysis leads through nodes 141, 118, 108, 103, 98, and 0. It finds silv- as the word stem, and the following morphological feature structure description:

$$\begin{bmatrix} pos : noun \\ decl : a \\ case : acc \\ num : sg \\ gen : \begin{Bmatrix} masc \\ fem \end{Bmatrix} \end{bmatrix}$$

Another valid path is $< 141, 118, 108, 102, 99, 0 >$. However, this s78.1(er,)-72ter,-71(i72t)-277.

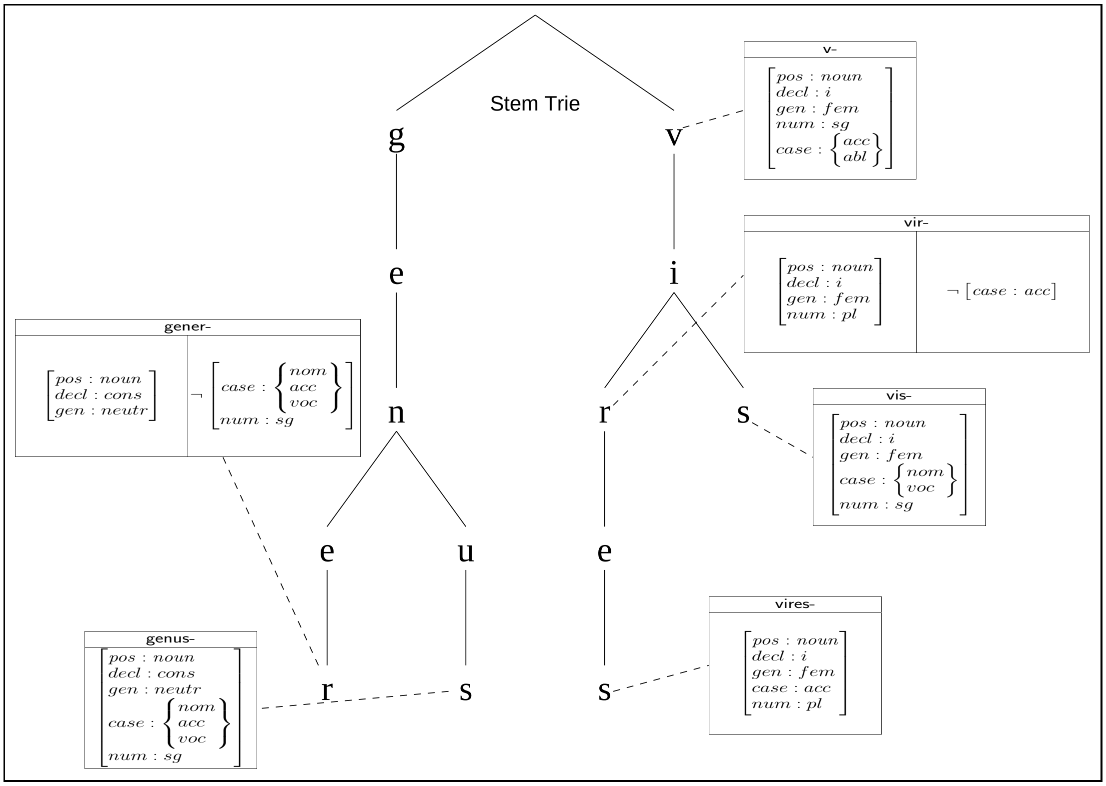


Figure 2.4: A sample lexicon trie with stems for vis, *force*, and genus, *gender*. The dashed lines indicate connections between word stems in the trie and their stem entries (in a separate file). vis is listed with four stems in this lexicon (v-, vis-, vir-, and vires-), genus with two (genus- and gener-). Each stem is associated with a feature specification, described in Section 2.3.4.

2.3.3 Unification with negation

Lexical entries can be made more concise by using negation in feature structures (Karttunen, 1984; Pereira, 1987; Johnson, 1991), even though the same information can be expressed more elaborately without negation. Negation in a feature structure specifies for which forms the feature structure is *not* suited; it does not make a direct statement for which forms the feature structure *is* suited.

Unlike other aspects of unification, if an intuitive semantics for negation is used, results can vary depending on the time when a negation is evaluated (cf. Pereira, 1987). Fortunately, this limitation does not pose a problem here, since the lexicon component evaluates negations as the very last step, on feature structures that do not become further instantiated afterwards.

It turns out that in the case of Latin morphology, no relevant loss of expressiveness is encountered when negation is limited to be applied to entire feature structures rather than parts of them, as in the following:

$$\neg \begin{bmatrix} \mathit{case} : \mathit{acc} \\ \mathit{gen} : \mathit{fem} \end{bmatrix} \tag{2.1}$$

Unification of this negative feature structure with a regular, positive feature structure only succeeds if the latter does not contain accusative case and feminine gender. For feature structures without disjunctions of feature values, unification of a positive feature structure P and a negative feature structure N fails if and only if the set of features in N is a subset of the set of features in P (for the restricted kinds of feature structures that are used here). For example, unification of negative feature structure (2.1) with the feature structure below fails:

$$\begin{bmatrix} \mathit{decl} : \mathit{i} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \mathit{fem} \end{bmatrix}$$

	positive	negative 1
vir–	$\begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : i \\ \mathit{gen} : \mathit{fem} \\ \mathit{num} : \mathit{pl} \end{bmatrix}$	$\neg \left[\mathit{case} : \mathit{acc} \right]$

Figure 2.5: Lexical entry for the stem vir- of the noun vis; the negative feature structure indicates that this entry is not applicable to the accusative plural (because the rare form vir-is is assumed not to occur). For the purpose of this and following examples, which only illustrate formal properties of the analyzer and do not necessarily represent optimal analyses of Latin morphology, I assume an analysis of vis which leaves out the uncommon forms vis and vi for gen. and dat. sg., and assumes a single form vir-es for acc. pl. (and not the less common vir-is from regular i–declension, cf. Wilson (1968); Glare (1982) even mentions a rare f-1-TfflΩ4.1802-0-TDfl7flΩ/F03-0-TDflΩ0.0009

onthe ern, eiv

mmo rfimoss du(e)-309.9tgo (pltg)-4ess rg(o)-567(n)312[s

Subsequently, the lexicon is searched for an entry containing the found stem. This search is efficient due to the lexicon's trie structure. If no appropriate entry is found, the analysis is discarded. If one is found, the entry's positive feature structure is unified with the feature structure from the analysis. If this unification succeeds, each of the lexical entry's negative feature structures are unified with the result. If one of the unifications fails, then the analysis is discarded, otherwise the final result of all unifications is a validated analysis.

For example, suppose that the following feature structure, together with a stem vir-, is found in the endings trie when the word virium is analyzed:[3]

$$\begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : i \\ \mathit{case} : \mathit{gen} \\ \mathit{num} : \mathit{pl} \\ \mathit{gen} : \left\{ \begin{matrix} \mathit{masc} \\ \mathit{fem} \\ \mathit{neutr} \end{matrix} \right\} \end{bmatrix} \tag{2.2}$$

vir- is then looked up in the lexicon, and the entry from Figure 2.5 is found. Unification of feature structure (2.2) and the lexical entry's positive features produces:

$$\begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : i \\ \mathit{case} : \mathit{gen} \\ \mathit{num} : \mathit{pl} \\ \mathit{gen} : \mathit{fem} \end{bmatrix} \tag{2.3}$$

Unification with the lexical entry's negative feature structure also succeeds, approving the analysis with stem vir- and feature structure (2.3) as valid.

[3] The ending analysis produces a general analysis which is not restricted to a particular gender. Gender is a *paradigm category* and peculiar to a noun. It is listed in the lexical entry. Using this information from the lexical entry, the unification process restrains the original analysis to the right gender.

2.3.5 Paradigm entries

A *word paradigm* comprises all word forms that belong to a particular base form. A paradigm also comprises one or more word stems from which its word forms are built. While the previous sections took a simplified view on the lexicon's structure to highlight other concepts, the lexicon structure is actually a little more complex than described so far to take care of common properties that members of the same word paradigm share.

The lexicon lists information common to a word paradigm in a single *paradigm entry.* In addition, *stem entries* list information specific to particular word stems and the word forms built from them. Figure 2.6 sketches a lexicon consisting of a trie for stem entries and a trie for paradigm entries.

This differentiation into two kinds of lexical entries reduces redundancy (just compare the feature structures for stems in Figure 2.4 with those in Figure 2.6) and is in accordance with a linguistic distinction of *paradigm* and *unit* categories. Features from paradigm categories like *part of speech* or — for nouns — *declension* and *gender* are intrinsic to a whole word paradigm; their values do not vary. They are stored in paradigm entries. Values of features from unit categories like *number* or — for nouns — *case* are imposed by inflection. Stem entries specify which stem is appropriate for particular values of unit categories. Figure 2.7 shows the complete lexical entry for the word paradigm vis. It contains the paradigm entry together with all its stem entries.

The division of entries into a stem part and a paradigm part is transparent (invisible) to other components. When the lexicon component is queried to return the (composite) entry for a particular stem, the stem entry is looked up in the stem trie and combined with its corresponding paradigm entry (which is found through the link from the stem entry as illustrated in Figure 2.6). The positive feature

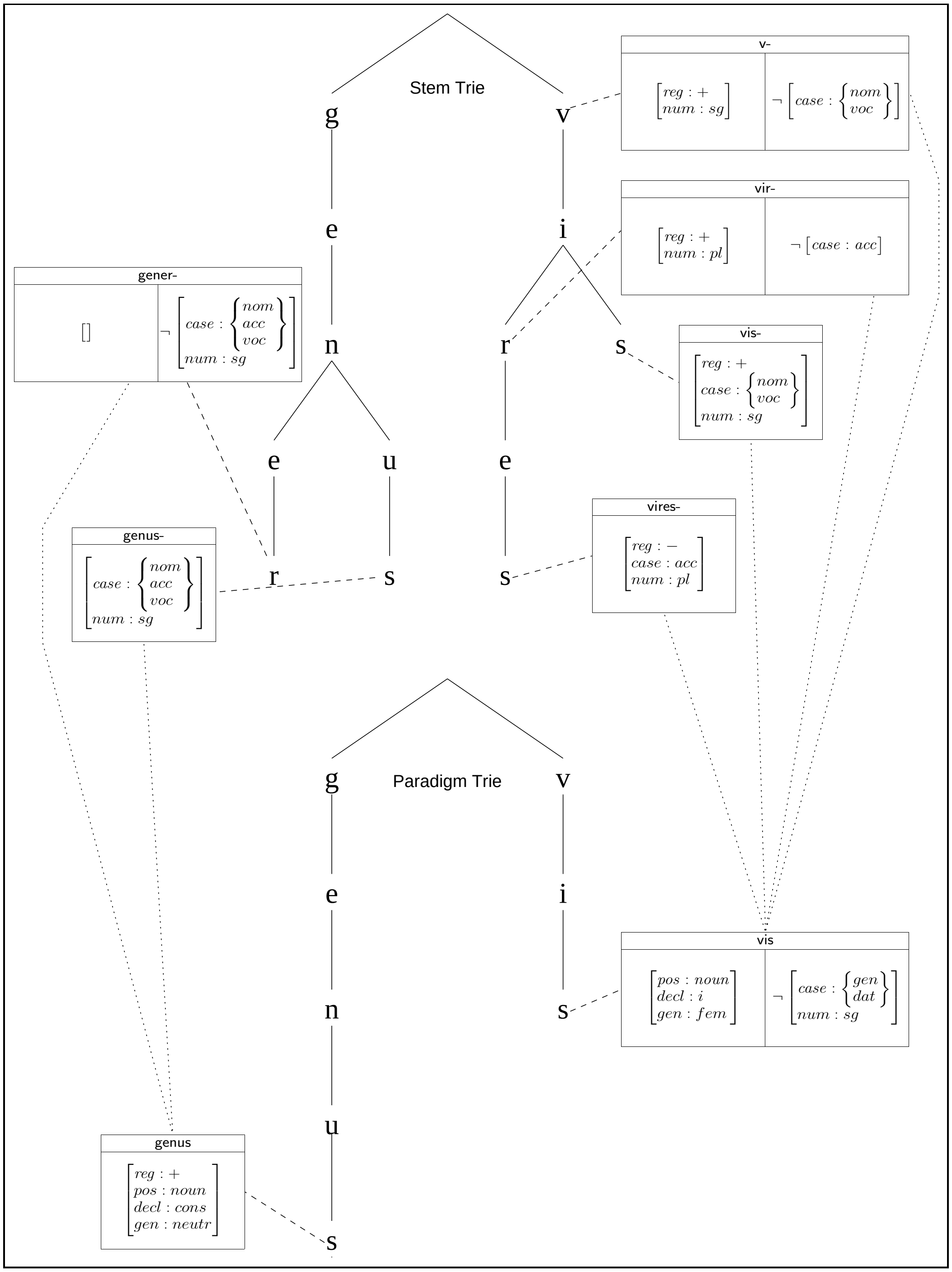

Stem Trie
g
e
n
e
r
u
s
v
i
r
e
s
s
v-
[reg : + num : sg]
¬ [case : {nom voc}]
vir-
[reg : + num : pl]
¬ [case : acc]
gener-
[]
¬ [case : {nom acc voc} num : sg]
vis-
[reg : + case : {nom voc} num : sg]
genus-
[case : {nom acc voc} num : sg]
vires-
[reg : − case : acc num : pl]
Paradigm Trie
g
e
n
u
s
v
i
s
vis
[pos : noun decl : i gen : fem]
¬ [case : {gen dat} num : sg]
genus
[reg : + pos : noun decl : cons gen : neutr]

Figure 2.6: A lexicon consisting of a trie for paradigm entries and a trie for stem entries. The dashed lines indicate the connections between lexical entries and their stems or lemmas in the trie. The dotted lines denote links between stem entries and paradigm entries. Each paradigm (entry) is connected to one or more stem entries. The feature *reg* is explained in Section 2.3.7.

	positive	negative 1
vis	$\begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : i \\ \mathit{gen} : \mathit{fem} \end{bmatrix}$	$\neg \begin{bmatrix} \mathit{case} : \begin{Bmatrix} \mathit{gen} \\ \mathit{dat} \end{Bmatrix} \\ \mathit{num} : \mathit{sg} \end{bmatrix}$
vis–	$\begin{bmatrix} \mathit{reg} : + \\ \mathit{case} : \begin{Bmatrix} \mathit{nom} \\ \mathit{voc} \end{Bmatrix} \\ \mathit{num} : \mathit{sg} \end{bmatrix}$	
v–	$\begin{bmatrix} \mathit{reg} : + \\ \mathit{num} : \mathit{sg} \end{bmatrix}$	$\neg \begin{bmatrix} \mathit{case} : \begin{Bmatrix} \mathit{nom} \\ \mathit{voc} \end{Bmatrix} \end{bmatrix}$
vir–	$\begin{bmatrix} \mathit{reg} : + \\ \mathit{num} : \mathit{pl} \end{bmatrix}$	$\neg \begin{bmatrix} \mathit{case} : \mathit{acc} \end{bmatrix}$
vires	$\begin{bmatrix} \mathit{reg} : - \\ \mathit{case} : \mathit{acc} \\ \mathit{num} : \mathit{pl} \end{bmatrix}$	

Figure 2.7: Lexical entry for the word paradigm vis. The first row shows the paradigm entry, the following rows show the stem entries. See Figure 2.5 for assumptions about the analysis of vis here. Section 2.3.7 explains the meaning of the *reg* (*regular*) feature.

structures of the two entries are unified, and the union of the two negative feature structure sets is built. The response to the query returns the unified positive feature structure and the union of negative feature structures.

2.3.6 Additional features and lemmatization

Paradigm entries are also convenient to store additional information of a morphological or non–morphological nature about a word. External applications can use this information after the morphological analysis is complete.

Moreover, paradigm entries provide an easy way to do *lemmatization* (deriving a word's base form (the *lemma*

	positive	negative 1	negative 2
domus	$\begin{bmatrix} pos : noun \\ decl : u \\ gen : fem \end{bmatrix}$		
dom–	$\begin{bmatrix} reg : + \\ case : \begin{Bmatrix} nom \\ voc \end{Bmatrix} \\ num : sg \end{bmatrix}$	$\neg \begin{bmatrix} case : abl \\ num : sg \end{bmatrix}$	$\neg \begin{bmatrix} case : acc \\ num : pl \end{bmatrix}$
domo	$\begin{bmatrix} reg : - \\ case : abl \\ num : sg \end{bmatrix}$		
domos	$\begin{bmatrix} reg : - \\ case : acc \\ num : pl \end{bmatrix}$		

Figure 2.8: A lexical entry for the noun domus. See the text for additional explanations.

For all "regular" analyses, the ending analysis adds the feature <*regular*:+>, and lexical entries for regular word forms also contain this feature. Figure 2.8 shows the lexical entry for domus. Figure 2.7 is another example that also demonstrates how forms with a smaller degree of irregularity are captured by adding additional "regular" word stems to a lexical entry. Finally, Appendix C contains several more illustrative (but "regular") lexical entries.

2.4 Summary

In this chapter, the main ideas underlying the theory of YALL, a morphological analyzer for Latin, were presented. Trie structures are used for both the analysis of endings and the representation of the lexicon. They are supported by unification mechanisms which include handling of disjunctive and negated feature structures. Several methods help to capture irregular word forms and to avoid overgeneration of inaccurate analyses.

Chapter 3

Java versus C++

This chapter describes deficiencies in YALL's original implementation, outlines why the choice of Java as a programming language alone remedies some of these deficiencies, briefly reviews the reimplementation process, and provides some of the more interesting implementation–specific details.

3.1 Problems of the original implementation

YALL had originally been implemented in C++. While this implementation was a very functional prototype of the system, it had potential for improvement. Some problems are described in the following.

- **Overly complex C++ code.** The C++ code of the original implementation is quite sophisticated. At the time of implementation, the Gnu C/C++ compiler did not yet recognize some concepts now documented in the ANSI C++ standard (ANSI/ISO/IEC), such as exception handling and templates, even though those concepts had already been implemented in many other C++ compilers. It also had several bugs that had to be circumvented. This made part of the code appear ugly and difficult to maintain.

  In addition, extensive use of advanced C++ features such as multiple inheritance or friend methods and friend classes made the code harder to understand and maintain. Even though such concepts are recommended to use for C++

programs, it seems to me that Java facilitates programming by reducing the number of such bells and whistles.

Also, standard data structures like lists, sets, and vectors had to be implemented from scratch in YALL (cf. Section 3.2.3). This led to vastly increased debugging demands. Furthermore, the graphical user interface, due to missing built–in support for windows and graphical user interface (GUI) programming (see below), suffered from these same complications of needing much low–level programming and debugging.

A crucial problem with C++ was loss of memory. Because of C++'s strong reliance on dynamic memory allocation by the programmer, some allocated memory is easily overlooked and never deallocated. The final version of YALL in C++ still suffered from memory losses. Although these were errors on the part of the programmer, the language made them easy to make and hard to detect.

- **New Insights.** During and after the original implementation of YALL, many ideas for improvements appeared which had not been implemented in the original version. Some of these are experiences gained from the first implementation (e.g., to use standard class libraries for abstract data types, or to use a different dynamic memory management mechanism); some are consequences of continued refinement of the theory (e.g., some changes to the endings trie, or the decomposition of feature structures with value disjunctions into feature structures without such disjunctions to facilitate the evaluation of negative feature structures, cf. Section 2.3.3); some are insights from redesigning the Java implementation (e.g., to use a declarative lexicon specification, or to use an external file for the endings trie specification; see Sections 3.4.1 and 3.4.3).

- **Only usable under MS{DOS.** Some emphasis had been put on machine independence during the development of C++–YALL. However, the development took place under MS–DOS and despite a relatively machine–independent programming style, as described elsewhere, some changes would have had to be made to port the software to a different architecture such as UNIX. However, such a port has never been attempted.

- **Descriptions in German.** Program code and documentation of the original implementation are in German, which is not surprising since the project was undertaken in Germany. However, this evidently limits the range of people able to use the software or to adapt the morphological analyzer to their own needs. To provide wider access to the analyzer, the reimplementation and documentation in English is helpful.

- **Unattractive user interface.** YALL was developed under MS–DOS, using the `djgpp` C/C++ compiler (D. J. Delorie, 1995), an MS–DOS port of the Gnu C/C++ compiler `gcc`. One of YALL's original goals was to be easily portable to other platforms. For that reason, the `Curses` C library, an elementary graphics and textual input/output library for text terminals, was used for the user interface. Variants of `Curses` are available for various platforms, including MS–DOS and UNIX. However, Curses does not provide the higher–level functions, such as support to manage windows or a mouse, that are necessary to develop an attractive and easy–to–use graphical user interface.

The reimplementation in Java seeks to approach some of the problems in the C++ version. It concentrates on the previous version's most interesting part, the morphological analyzer and lemmatizer. Additionally, due to structural changes in the software, the need for a graphical user interface has vanished.

3.2 Some advantages of Java over C++

YALL's reimplementation in Java appeared considerably easier than the original implementation in C++. Some of this ease can probably be attributed to the circumstance that after a first implementation the problem domain is better understood, and application design is facilitated by being able to build upon the results and insights from the previous design and implementation. However, there are also differences between the two programming languages that have an effect on implementation difficulty. This section describes, from an empirical or experience–based point of view, the differences that have had an important impact on the reimplementation of YALL . There are certainly other differences between C++ and Java, which can be found in standard textbooks comparing the two languages (for example, Boone, 1996; Chew, 1998), but those are not relevant in the present context. Remarkably, C++ has developed further since YALL's first implementation, but the differences outlined below are mostly unaffected by these changes.

This section makes use of object–oriented terminology and presupposes knowledge of programming languages in general, and of C++ and Java more specifically. Readers unfamiliar with these subjects may consider an introduction to object–oriented programming, such as Stroustrup (1997), which is also the standard introduction to C++. For an introduction to Java, numerous textbooks exist, as well as a free online course (Sun Microsystems, 1999a). For a general introduction to the theory of programming languages I recommend Louden (1993).

3.2.1 Better handling of class interface and class implementation

In C++ it is recommended that interface and implementation of a class and its methods be stored in separate files (typically a header file with extension `.h` for the class interface, and an implementation file with extension `.cc`). The abstract, declarative class interface mainly serves to specify the class's member functions and member variables, as well as the class's integration into a class inheritance hierarchy, in order to make this knowledge available to the user of the class who can then see how he is supposed to use the class (comments for the class user should also be part of the class interface). The class implementation presupposes the class interface and implements the methods and some variables specified in the interface.

An inherent problem with this approach is that whenever a method or variable declaration changes, it needs to be changed in two places, which leads to consistency problems during program development.

Another problem is that all public, protected, and private class members need to be declared together in the same place. From the perspective of data encapsulation, this is not very fortunate: since the user of the class should only see public (and possibly protected) members, he should not have to or be able to view the private members. And when the internal structure (private members) of a class changes, there should not be a need to change the public class declaration as in C++, since the usage of the class remains the same.

Java remedies these problems. A Java class source file (a `.java` file) is the source for both the class interface and the class implementation. Primarily, a `.java` file constitutes the class implementation. However, the class interface and comments are generated automatically by the `javadoc` utility from the class source code, typically in `HTML` format. By default, only public and protected classes are documented in this way, which leaves the documentation unchanged if changes are made that do not affect the use of the class. An example for documentation generated by `javadoc` is the documentation for the Java version of YALL in Appendix G.

Usually, the proper way to design a class in object–oriented programming (on the code level) is to specify its interface first, and to write the implementation as the last step. Java takes care of this approach by enabling `javadoc` to generate class interface files from *code stubs* in the `.java` file, which only contain declarations of member variables and declarations (heads) of member functions. Hence, a Java class source file should first contain only the information necessary for the class interface. At a later stage of program development, the interface can be implemented. In practice, the class interface tends to undergo change at this point due to new insights and requirements from the implementation. When the programmer makes such changes in the class implementation, Java takes responsibility for corresponding modifications to the class interface.

3.2.2 Automatic memory management

C++ offers memory management capabilities that do not go substantially beyond those of C. Essentially, the only improvement on the surface is the ability to allocate and deallocate heap space for single dynamic objects via the `new` and `delete` operators.

Java's memory management is more powerful. Java adds a garbage collection mechanism which includes automatic deallocation of memory for objects that are no longer in use. The programmer does not need to worry about explicit deallocation of objects.

This feature is a very beneficial improvement over C++. Object–oriented programming relies substantially on the dynamic creation of objects, which also requires their deallocation at some point. If deallocation has to be done by hand, a very rigorous approach to the construction and destruction of objects is required. Objects in general do not only contain members of primitive data types (for which no memory allocation in addition to memory for the object itself is needed), but also contain other objects or references to other objects; these objects require dynamic allocation and deallocation in turn. In addition, several objects can share identical member objects which makes the responsibility for deallocation quite sophisticated. In C++, allocation of memory for member objects is usually done in the *constructor* of the containing object, a special method called upon creation of the containing object. Deallocation of memory for member objects is performed in the *destructor*, a method called upon destruction of the containing object. However, not all member objects can be created when the containing object is created (since their content may not be known yet at the time), and not all member objects can be destroyed when the containing object is destroyed (due to shared member objects as described above). Consequently, keeping track of all allocated resources can be extraordinarily error–prone.

In the C++ version of YALL, it turned out not to be feasible to track down and remove all such occurrences of "memory leakage." Even after months of debugging, memory loss was still present. An additional complication in the debugging of memory leaks is the difficulty in detecting the source of the leak. Memory leaks only come about in a decrease in available memory during program runs, but except maybe with sophisticated debugging tools, the point of the leak in the program is not easy to find. In C++–YALL, many leaks could be detected and removed, but some still remained.

Java's automatic memory management consequently has several advantages for the programmer. Less rigor in memory management is required, memory leaks are avoided, and debugging is substantially shortened. Having shared member objects and leaving member objects unspecified after the creation of the containing object also becomes painless. While the existence of a large number of interacting object classes can become a complexity problem in C++ due to possible complications in terms of memory management, there is no such headache in Java, and the use of objects can be a pleasure, maybe comparable in ease of use to parameterized *procedures* in non–object–oriented imperative programming.

However, there are also at least two disadvantages in Java's approach. First, automatic garbage collection requires some time to execute, which slows down the execution speed of the software. Second, it can easily lead the programmer to forget about the overhead involved in dynamic memory allocation and deallocation and make him program in a style that he might not have used if he had had to undertake memory allocation more explicitly.

To me, both of these problems seem negligible compared to the benefits. If memory management slows down execution, then this is a low price paid for improved software reliability and shortened development time in an age of ever–increasing speed and ever–decreasing price of processing power. This same counterargument also applies to the point of a wasteful programming style; and in addition, as known from program optimization, such a style is usually only critical in crucial places, frequently executed passages of code; these, if programmed inefficiently, lead to an over–proportional slowdown of execution. If execution speed is important, it is often sufficient to simply locate such places with standard software development tools (*profilers*) and to optimize them.

3.2.3 Standard libraries and general standardization

Java 2 (or, more specifically, the Java 2 Platform, v. 1.2) provides a rather extensive set of ready–to–use standard data structures such as sets, maps, lists, and arrays in many different flavors. In addition, the so–called Swing classes to support graphical user interfaces are part of the language. Some of these libraries have been part of Java since the first released version in 1995, but essential parts have been added more recently, in particular with the introduction of Java 2 in December 1998.

In contrast, the first version of C++ was officially released about 1985 (and documented in Stroustrup, 1986). A library as part of the language containing standard data structures has only been added over the last few years, together with the development of an official C++ standard (ANSI/ISO/IEC). It does not contain any graphical user interface support.

This also illustrates a more general point. Sun Microsystems' central control over the development of Java has the advantage that standardization is not a problem. While there have been several drafts of a proposed ANSI C++ standard over many years, Sun with their *100% pure Java* policy can more easily define the exact Java language and produce similarly standardized extensions to the "core" Java language at (reasonable) will.

For application development, this has important consequences. One goal in the development of YALL in C++ was to keep the code as independent from a particular development environment and operating system as possible, since a morphological analyzer naturally does not have much to do with a particular computer system. In 1994, development of a C++ standard library for abstract data types was only starting. Many development environments had such libraries, also for graphical user interface development, but using those would have meant to become dependent on a particular version of a particular development environment on a particular machine. Consequently, I had to develop my own data structure classes and graphical (text–terminal–based) user interface routines, which took some effort.[1] In contrast, only three years after its introduction, Java has already become more mature than C++ in these respects.

There is yet another problem with standardization of C++. While a language standard has finally been accepted (ANSI/ISO/IEC), there is no guarantee whatsoever that any given compiler adheres to the standard for reasons such as cost or difficulty in implementing the standard, or incompatibilities of new features in

[1] Due to a redesign of YALL's lexicon component and to a focus on the morphological analyzer and lemmatizer component of YALL in this thesis, a graphical user interface has become unnecessary for the current implementation. However, future applications based on YALL will benefit from Java's provisions for graphical user interface development.

the standard with traditional features of the compiler. Hence one needs to study the exact limitations of the compiler one uses before any code is written (let alone the need to become acquainted with the proprietary class libraries coming with the compiler). For example, when YALL was developed with the GNU C++ compiler, the compiler's manual claimed support for C++ *class templates*, a language feature that had not been part of C++ originally, but which had been around for a while in 1994. After a good portion of the class design had been completed making extensive use of class templates, the code got to the point to be compiled for the first time. Surprisingly, the templates as described in Stroustrup (1991) did not compile due to a different implementation in the Gnu compiler. The effort needed to redesign the code was disheartening.

Java avoids this problem by having a single "reference" compiler that sets the standard. When a programmer changes to a different development environment or hardware platform, the need to relearn the features and limitations of the new environment is avoided. In addition, code and libraries developed for the old environment can quite painlessly be reused, and there is no need to start over from "near–scratch."

3.2.4 Additional support from IDE or external libraries

C++ requires considerable proprietary support from a development environment (often an *integrated development environment*, IDE) beyond what is defined as part of the language. One example is debugging of dynamic memory allocation. As described before, without either an additional automatic garbage collection mechanism or strong support for debugging dynamic memory allocation, this task can overwhelm a developer. Another example concerns more general debugging needs. C++, as an imperative programming language in the tradition of C, encourages a programming style that makes abundant use of pointers and pointer–based structures such as arrays. However, as every programmer knows, pointers tend to point to the wrong places, and array elements tend to be accessed beyond the limits of the array. In order to easily locate such problems, additional run–time support is required from the development environment, and the ease of debugging hence depends substantially on that environment.

Java requires little or no additional support at all from a development environment. Dynamic memory management is already built into Java and does not pose a problem. Pointers are not available in Java anymore; they are replaced by *references* to objects and more "abstract" arrays that cannot be referenced through pointers as in C++. Basically, in order to refer to objects, one *has to* use references; objects cannot be bound to a variable directly. While in C++ one often has the burden of deciding between two very similar concepts, namely whether to use a reference or a pointer to an object (a choice which has to be remembered later when the reference or pointer is used), Java does not leave this confusing choice between two almost identical concepts and does not even have variables that stand for objects directly. This can facilitate object handling considerably. In addition, type–checking at compile time is very strict and restrictive compared to C++, capturing many incorrect variable assignments early. Later on, typical run–time problems are monitored, such as references that do not point to proper places.

In general, development environments for Java do not need to provide much; Sun's free Java distribution already contains most of what is needed: class libraries, a compiler, an interpreter, a (command–line) debugger, a documentation tool (*javadoc*), several other development tools, and complete documentation of the language and the distribution. A development environment may just want to integrate these components with a visually appealing and easy–to–use graphical user interface, and to provide an editor with the additional convenience of source–level debugging.[2] All components in the Java distribution are also well–integrated with each other. An example is the `toString` method from the Java "root" class `Object`. Every class should redefine this method to make a class instance return some kind of instance "state" in string format. The method can be used by other classes within the application itself (and YALL does so), but it is also used by the debugger as a special routine to provide information about the content of objects.

3.2.5 Platform independence and Internet accessibility

Two of the best–known features of Java are its platform independence ("write once, run anywhere": code only needs to be written once and runs on all platforms that support Java) and its usability for Internet–related software development, which is in part due to its platform independence. As mentioned before, one of YALL's original goals has been platform–independence, which makes Java an excellent implementation language. In case the desire arises to make YALL or an application using YALL available online, it is already well–equipped for Internet access. C++ does not have much to offer in these regards.

3.2.6 Java problems

Java also has disadvantages compared to C++. One of the foremost is execution speed. Java programs tend to run considerably slower than C++ programs. Since Java is a (partially) interpreted language and uses garbage collection, this disadvantage is unlikely to ever disappear completely. Hence (as usual), the decision on a programming language depends on the requirements of the developed application. If a fast application for a specific platform is required, Java should not be used. If

[2] These low requirements probably explain the abundance of high–quality, free and commercial development environments available for Java.

however speed is not so critical or cross–platform availability is desired, Java brings along convincing qualities to be the language of choice.

3.3 The reimplementation process

Java is generally said to be similar to C++. Hence I had initially expected to be able to make a rough automatic translation from C++ to Java, to refine the result a little further by hand, and then to arrive at acceptable Java code. However, an attempt to use an automatic translator (Tilevich, 1997) ended in Java code which was so unsatisfactory that it was easier to reimplement the system, even though the mentioned translator tool (or rather an earlier version of it) appears to be the most widely referenced tool of its kind.

I chose to use the newly released Java 2 Platform for the reimplementation, even though it meant an anticipated risk because of Java 2's novelty. However, there were two important reasons in its favor. Java 2 introduces the so–called `Collection` classes as part of the standard Java class library (the Java *API*), which provide lists, sets, hash–tables, and similar standard data structures. Second, having future extensions of YALL in mind, I wanted to use the Java version that would remain current for the longest time, in order to avoid potential incompatibilities with future Java versions for as long as possible.

The risk in using Java 2 turned out to include missing, incomplete, or error–prone support for Java 2 from most integrated development environments (IDEs). Sun's own Java development environment, Java Workshop (Sun Microsystems, 1998), did not support Java 2 yet when I started the reimplementation. I eventually turned to XEmacs (XEmacs), a versatile text editor, in combination with JDE (Kinnucan, 1999), the "Java Development Environment" for XEmacs. Since JDE had not had a good visual debugger, I later tried a few other IDEs: NetBeans (NetBeans, 1999b) and a new version of Sun's Java Workshop (Sun Microsystems, 1999b), both of which claimed to support Java 2. However, both failed to support debugging properly. I tried a beta version of Gandalf (NetBeans, 1999a), a more recent version of NetBeans, which seemed to be capable of proper Java 2 debugging, but it was too slow to be useful. Eventually, I chose Kawa (Tek–Tools, 1999), an unsophisticated but fast IDE which supports visual debugging of Java 2 code well.

3.4 New and improved program features

The transition from C++ to Java includes a variety of structural changes to the program. Some of these changes are of minor importance, like the use of different and more efficient standard data structures, minor improvements to algorithmic efficiency and structure, or the exact program code (which is given in Appendix F). Other changes are of great relevance to the user or to the system architecture and will be described in the following.

3.4.1 Declarative lexicon specification

Previous problems

In the C++ implementation, the user had to enter the system's lexicon interactively. Using a sophisticated text–based user interface, lexical entries could be entered by interactively providing lemma and stem grapheme and specifying the entry's features one by one through selection from context–sensitive menus. Changes to lexical entries had to follow the same procedure. While this ensured syntactically correct lexical entries (by only providing contextually valid choices to the user), it made the already sophisticated process of entering correct lexical entries overly tedious and lengthy by requiring the user to navigate through menus.

In addition, the lexicon was stored in a proprietary binary format (namely just the trie format) which made it difficult to quickly view lexical entries, to check their validity (displaying lexical entries was only possible by navigating through menus, basically one feature at a time), to generally display what's in the lexicon at all, to make quick changes, or to quickly enter series of words with similar morphological characteristics. The lexicon structure also made it harder to convert (import) other lexicons to YALL's format or to convert (export) YALL's format to other formats. In addition, making modifications to the lexicon storage structure usually meant having to discard the old lexicon and starting a new lexicon from scratch (unless appropriate conversion routines were written, see below). Furthermore, a reimplementation (like the present one) or another software wishing access to the existing data in the lexicon had to adhere exactly to the existing trie and entry storage format, even when storage requirements changed (for example, for some applications a *hash table* may be preferable over a trie structure) or new, additional (non–morphological) information needed to be stored in the entries (which was made impossible). Alternatively, a lengthy conversion program could have been written to transform the existing lexicon into a new format, or the complete lexicon could have been reentered.

For these reasons, the present implementation uses an additional declarative lexicon definition file containing the entire lexicon in text format. From this declaration the actual lexicon trie is automatically generated.

The current implementation

The syntax of the declarative lexicon specification is captured by the following *context–free grammar* in an extended *BNF* notation (for BNF, cf. Aho et al., 1986). The square brackets designate an optional occurrence of the enclosed symbol string, the curly braces stand for any number (including zero) of iterations of the enclosed symbol string. A string enclosed in single quotes stands for its literal occurrence. Commas are always used literally, i.e. they are part of the object language (which, in this case, is the (meta–) language used to specify the lexicon).

$$\begin{array}{rcl}
\textit{lexicon} & \longrightarrow & \{\textit{lemmaentry}\} \\
\textit{lemmaentry} & \longrightarrow & \text{‘[’}\textit{lemma}, \textit{posfeatures}, \textit{negfeaturesets}, \textit{stementries}\text{‘]’} \\
\textit{lemma} & \longrightarrow & \textit{grapheme} \\
\textit{posfeatures} & \longrightarrow & \textit{features} \\
\textit{negfeaturesets} & \longrightarrow & \text{‘[’}\{\textit{features}\}\text{‘]’} \\
\textit{features} & \longrightarrow & \text{‘[’}[\textit{featurevalue}\{, \textit{featurevalue}\}]\text{‘]’} \\
\textit{stementries} & \longrightarrow & \text{‘[’}\{\textit{stementry}\}\text{‘]’} \\
\textit{stementry} & \longrightarrow & \text{‘[’}\textit{stem}, \textit{posfeatures}, \textit{negfeaturesets}\text{‘]’} \\
\textit{stem} & \longrightarrow & \textit{grapheme}
\end{array}$$

grapheme can be a lemma or a stem like laudare or laud. *featurevalue* is a feature value such as *feminine* or *subjunctive.* It is not necessary to specify category–value pairs, since each feature value is unique over all categories, and the parser automatically determines the right category for a value.

Instead of giving a precise semantic definition for the grammar symbols above, their use is illustrated by the following (toy–size) declarative lexicon file (representing entries for laudare (cf. Appendix C) and vis (cf. Figure 2.7). *Whitespace* such as spaces, tabs, or linefeeds is not significant between terminal symbols.

```
[
  laudare,
  [regular,verb,a_conj],
  [],
  [
    [
      laud,
      [prs_stem],
      []
    ]
    [
      laudav,
      [perf_stem],
      []
    ]
    [
      laudat,
      [sup_stem],
      []
    ]
  ]
]
[
  vis,
  [noun,i_decl,feminine],
  [[genitive,dative,singular]],
  [
    [
      vis,
      [regular,nominative,vocative,singular],
      []
    ]
    [
      v,
      [regular,singular],
      [[nominative,vocative]]
    ]
    [
      vir,
      [regular,plural],
      [[accusative]]
    ]
    [
      vires,
      [exception,accusative,plural],
      []
    ]
  ]
]
```

The parser for the grammar is implemented as a recursive–descent parser with one character lookahead (cf. Aho et al., 1986).

Generation of the lexicon trie files from the declarative lexicon specification is triggered by the command

```
java Lexicon
```

This generates six lexicon trie files from the specification file `yalllext.txt`.

Advantages of the new approach

The declarative definition has several advantages:

- **Easy parsing and conversion.** Since the declarative lexicon file is based on a simple context–free grammar, it can be easily parsed and "compiled" into a trie format or converted into any other desired file format. This allows other programs besides the morphological analyzer easy access to the lexicon as well. In addition, if the structure of the trie file format changes in the future, the lexicon can easily be regenerated in the new format from the declaration.

- **Easier conversion of other lexicons.** A routine to convert an existing Latin lexicon into YALL's format would only need to produce the simple context–free grammar format, not the more sophisticated trie format. The conversion into trie format is done by YALL.

- **Easy viewing, updating, and inserting.** These standard lexicon operations can now be performed with any text editor or similar text processing tool, which, for example, could also be used to systematically correct certain mistakes in the lexicon through the editor's replacement function. In addition, one could, for example, come up with a standard entry *pattern* for, say, nouns of a–declension, specifying all the standard features. This pattern could then be copied in the text editor as often as needed, and only lemma and stem graphemes would need to be filled in manually for each entry. As a future improvement, a small application could be written to automate this use of *templates*.

3.4.2 Simplified lexicon structure

The implementation of the lexicon tries in the C++ version was designed to be efficient in terms of access time as well as in terms of disk space consumption. However, the methods needed to access the lexicon were very complex. There were three different kinds of data in the lexicon tries. When a node in the trie had more than five child nodes, it was stored in an array to achieve optimal access efficiency. The array contained one element for each letter of the Latin alphabet to designate each possible child node; “absent” child nodes were marked by a `NIL` (Not In List) symbol in the array. With five children or less, a linked list was used with one list element per child node; this saved memory compared to an array representation. In addition, a node in the lexicon trie could also represent a list of pointers to lexical entries. Consequently, the trie access procedures were rather complicated.

A theoretical assessment of the achieved space gain by using this overly complex trie implementation shows that the gain is limited approximately by an upper constant factor of 27 (26 characters of the English alphabet plus a special element pointing to entries), but is actually far lower in the average case.

The new implementation generally uses arrays for the storage of each node in the trie rather than lists and arrays. In addition, the lists of pointers to lexical entries have been moved to a separate file. From every node there is now a pointer to a position in that separate file, denoting the beginning of a list of pointers to lexical entries connected to the trie node in question. The pointer can have the value `NIL` to denote an empty list. This approach makes the implementation more straightforward and less error–prone. The complete lexicon trie structure now consists of six files:

the paradigm trie file (yalllexl.yt), the paradigm entry lists file (yalllexl.yel), the paradigm entry file (yalllexl.ye), the stem trie file (yalllexs.yt), the stem entry lists file (yalllexs.yel), and the stem entry file (yalllexs.ye).

3.4.3 Simplified endings trie specification

Similarly to the lexicon specification described in Section 3.4.1, the trie for the ending analysis is also stored in a declarative format. In the C++ version of YALL, the endings trie had not been stored completely declaratively, but in a kind of declarative–procedural format, with several lines of program code for every node in the trie. Here is an illustration of two entries for the personal endings (nodes 68 and 70) from Figure 2.3 (modified to correspond to the figure):

```
mat[68]=new extaffix(
  "mus" ,new featuremap(new feature(plural),new feature(active),
                        new feature(one),(feature *)0),
  mat[24],mat[29],mat[33],(extaffix *)0);

mat[70]=new extaffix(
  "mur" ,new featuremap(new feature(plural),new feature(passive),
                        new feature(one),(feature *)0),
  mat[24],mat[29],mat[33],(extaffix *)0);
```

The main disadvantage of this approach is that it makes the coding of the endings trie specification into the program quite cumbersome. Not surprisingly, this representation had to be completely discarded for the reimplementation, due to Java's different (and simplified) syntax. As a consequence, the endings trie is now stored in its own (data) file, and read in whenever the application is started. This makes it necessary to be able to read and parse the trie quickly. Consequently, the trie representation is optimized for speed and simplicity. It is not nice to read, but this is quite unnecessary since the endings trie does not need to be changed once it is entered correctly.[3] The above example now becomes the following (again modified from the original file to correspond to Figure 2.3):

```
68
mus
plural
active
one

24
29
33

70
mur
plural
passive
one

24
29
33
```

The general structure of the endings trie file is defined by the following rules:

1. The file can have any number of entries for nodes, but must have at least one.

2. The last entry is the "start node" of the trie.

3. Each entry consists of several lines:

    (a) a textual identifier for the node (some string; currently number strings are used)

    (b) the node's grapheme

[3] Because of the simple endings trie file format, it was possible to generate the more readable specification of the endings trie in Appendix E from the endings trie file semi–automatically via regular expression replacement.

(c) one line for each feature value; as with the lexicon specification, only feature values need to be specified, no categories; a blank line marks the end of the feature value list

(d) one line for each successor node identifier; a blank line marks the end of this list of links

An important difference to the conceptual description in Chapter 2 is that the trie can now be regarded "upside–down." When the analysis starts, the traversal of the trie graph begins at a *single leaf node*, from where it proceeds to the former leaf nodes from Chapter 2, and then further into the trie towards the root node. However, since the root node does not have any purpose left except maybe to "conceptually dominate" the whole trie, it is now simply left out (which brings the endings trie even closer to an acyclic directed graph and further away from a tree).

The entire endings trie file is given in Appendix E in a form modified for readability. Compared to the endings trie from Bubenheimer (1995), it also contains minor corrections in content.

3.4.4 User input and output

The user specifies the word to be analyzed as a command–line argument to the analyzer, as in the following example to analyze laudare (the command `java` starts the Java interpreter, the argument `Yall` invokes class `Yall`, and `laudare` is passed as an argument to `Yall`):

```
java Yall laudare
```

The output from YALL is a textual display of the found analyses together with the found word stems and morphological features. For laudare, the output is the following:

```
Found 2 analyses for laudare.


Lemma: laudare
Stem: laud
[
voice : passive
conjugation : a_conj
mode : imperative
number : singular
tense : present
person : two
partofspeech : verb
regularity : regular
stem : prs_stem
]


Lemma: laudare
Stem: laud
[
infinitivity : infinitive
voice : active
conjugation : a_conj
person : infinite
partofspeech : verb
regularity : regular
stem : prs_stem
]
```

The displayed feature structures do not contain disjunctions. During the different unification operations, feature structures with disjunctions of feature values are eventually broken up into disjunctions of feature structures without value disjunctions. This is necessary to check the validity of found feature structures against negative feature structures from the lexicon (see Section 2.3.3). Even though an attempt could be made to at least partially "reassemble" split–up feature structures, the result, in the general case, would only be a partial reassembly and could still contain a disjunction of feature structures in addition. The example in Appendix A shows a case where it is not possible to obtain a single positive feature structure as the result of unification with negation.

Chapter 4

Conclusions and Outlook

This thesis provided theoretical and practical refinements of a morphological analyzer and lemmatizer for Latin, originally developed by Bubenheimer (1995). At the same time, it demonstrated differences in the use of Java and C++ for software development, and found a remarkable facilitation of small–scale software development with Java. A reimplementation of the morphological analyzer in Java now serves as a clean basis for future improvements to the analyzer.

4.1 Theoretical refinement

While the original theory behind the analyzer was comparably weak and proprietary, it has been shown in this thesis that it can be cleanly and uniformly reformulated in terms of current linguistic theory using tries, feature structures, and graph unification.

Tries are used to represent both stems and endings. This leads to a uniform computational model for Latin inflectional morphology.

Feature structures and unification supplement this model and allow to specify morphological properties of inflected word forms. They are also fundamental in providing the connection between analysis of word ending and lexical lookup of word stems; they ensure that only valid combinations of these two processes are admitted.

Feature structures can include disjunctions and negations to provide greater flexibility and more concise expressiveness. These extensions are in line with common linguistic theory.

4.2 Practical refinement and reengineering

The thesis has presented observations gained from redesign and reimplementation of the morphological analyzer in Java. The reimplementation was considerably easier than the original implementation in C++; experiences from the first implementation helped to avoid mistakes and provided guidance for the reimplementation; in addition, differences between C++ and Java in their features and their abilities to facilitate program development were contributing factors. However, an attempt to automatically convert C++ code to Java failed. The redesign of the morphological analyzer also brought about greater declarativeness of the used data structures and a reduction of the user interface from complex semi–graphical user interaction to a simple command–line tool.

A well–known problem of Java, its execution speed, has not turned out to be a problem in the present application. The morphological analyzer runs at an acceptable speed.

4.3 Future extensions

An additional motivation for attempting a reimplementation in Java was to provide a clean basis for future improvements of YALL. While these improvements could already have been applied to YALL's C++ version, and many of them had already been described as possible extensions in Bubenheimer (1995), I did not feel comfortable to add such extensions to a program that suffered from a number of deficiencies as discussed in Chapter 3.

The following extensions could be added to YALL for a more comprehensive morphological analysis system:

- **Complete Latin morphology.** The current implementation of the analyzer handles nouns, verbs, and adjectives in the positive form (as well as non–inflected *particles*). The first three parts of speech are crucial to show the validity of the theory since they provide the most variety in inflection. Further development must extend the morphological analyzer to the remaining parts of speech, but this extension is not likely to cause crucial changes in the theory.

  More specifically, adverbs, numbers, pronouns, clitics, and comparison of adjectives are still missing from the analyzer. In addition, better support should be added for a limited number of high–frequency irregular verbs such as esse and ire as indicated in Appendix B.

- **Lexicon.** The size of the lexicon, which currently only consists of a few words for testing purposes, needs to be extended to a large subset of Latin vocabulary to make the morphological analyzer useful. The most promising approach appears to be to convert an existing Latin lexicon to YALL's format. Entering the lexicon manually does not seem appealing.

Bibliography

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, Reading, Mass., 1986.

J. H. Allen and J. B. Greenough. *New Latin Grammar*. Aristide D. Caratzas, New Rochelle, New York, 1903.

ANSI/ISO/IEC. *Information Technology — Programming Languages — C++*. American National Standards Institute (ANSI), International Organization for Standardization (ISO), and International Electrotechnical Commission (IEC), September 1998. ANSI/ISO/IEC 14882.

B. Boone. *Java Essentials for C and C++ Programmers*. Addison–Wesley, Reading, Mass., 1996.

U. Bubenheimer. *YALL. Eine morphologische Analysekomponente für das Lateinische zum Einsatz in einem lehrunterstützenden System*. Universität Koblenz–Landau, Institut für Computerlinguistik, Koblenz, Germany, 1995. Studienarbeit.

F. F. Chew. *The Java/C++ Cross–Reference Handbook*. Prentice Hall, Upper Saddle River, New Jersey, 1998.

M. A. Covington. Converging transition networks and sub–morphemic regularities in Latin noun inflection. Unpublished paper, Artificial Intelligence Center, The University of Georgia, February 1999.

D. J. Delorie. djgpp 1.12M4. Computer software, available from http://www.delorie.com/djgpp, 1995.

P. G. W. Glare, editor. *Oxford Latin Dictionary*. Oxford University Press, New York, 1982.

H. J. Hillen. *Krüger Lateinisches Unterrichtswerk. Zweiter Teil. Grammatisches Beiheft*. Verlag Moritz Diesterweg, Frankfurt am Main, Germany, 1986.

M. Johnson. Features and formulae. *Computational Linguistics*, 17(2):131–151, 1991.

L. Karttunen. Features and values. In *Proceedings of COLING 84*, pages 28–33, 1984.

P. Kinnucan. JDE 2.1.4 — Java Development Environment for Emacs. Computer software, available from http://sunsite.auc.dk/jde, 1999.

D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison–Wesley, Reading, Mass., 1973.

K. C. Louden. *Programming Languages: Principles and Practice*. PWS Publishing Company, Boston, Mass., 1993.

NetBeans. Gandalf Beta 1. Computer software, available from http://www.netbeans.com, April 1999a.

NetBeans. NetBeans DeveloperX2 2.1. Computer software, available from http://www.netbeans.com, March 1999b.

F. C. N. Pereira. Grammars and logics of partial information. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, volume 2, pages 989–1013, Cambridge, Mass., 1987. MIT Press.

H. Rubenbauer, J. B. Hofmann, and R. Heine. *Lateinische Grammatik*. C. C. Buchners Verlag, Bamberg, Germany, 1989.

R. W. Sproat. *Morphology and Computation*. MIT Press, Cambridge, Mass., 1992.

B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, Mass., first edition, 1986.

B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, Mass., second edition, 1991.

B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, Reading, Mass., third edition, 1997.

Sun Microsystems. Java Workshop 2.0a. Computer software, previously available from http://www.sun.com/workshop/java, 1998.

Sun Microsystems. The Java tutorial. World Wide Web page, http://java.sun.com/docs/books/tutorial, 1999a.

Sun Microsystems. Java Workshop 3.0. Computer software, available from http://www.sun.com/workshop/java, 1999b.

Tek–Tools. Kawa 3.21. Computer software, available from http://tek–tools.com/kawa, 1999.

I. Tilevich. C2J++. Computer software, available from http://sol.pace.edu/˜tilevich/c2j.html, 1997.

W. M. Wilson. *An Essential Latin Grammar*. Macmillan, London, 1968.

XEmacs. XEmacs 20.4. Computer software, available from http://www.xemacs.org, 1998.

Appendix A

Unification with Negation and Disjunction

This appendix gives a longer example including negated and disjunctive feature structures for the unification method described in Section 2.3.3.

$$
\begin{aligned}
a &= \neg \begin{bmatrix} \mathit{decl} : \mathit{i} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \begin{Bmatrix} \mathit{masc} \\ \mathit{fem} \end{Bmatrix} \end{bmatrix} \\
&\equiv \neg \left(\begin{bmatrix} \mathit{decl} : \mathit{i} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \mathit{masc} \end{bmatrix} \vee \begin{bmatrix} \mathit{decl} : \mathit{i} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \mathit{fem} \end{bmatrix} \right)
\end{aligned}
$$

$$
\begin{aligned}
b &= \begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : \mathit{i} \\ \mathit{num} : \mathit{pl} \\ \mathit{case} : \begin{Bmatrix} \mathit{nom} \\ \mathit{acc} \end{Bmatrix} \\ \mathit{gen} : \begin{Bmatrix} \mathit{masc} \\ \mathit{neutr} \end{Bmatrix} \end{bmatrix} \\
&\equiv \begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : \mathit{i} \\ \mathit{num} : \mathit{pl} \\ \mathit{case} : \mathit{nom} \\ \mathit{gen} : \mathit{masc} \end{bmatrix} \vee \begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : \mathit{i} \\ \mathit{num} : \mathit{pl} \\ \mathit{case} : \mathit{nom} \\ \mathit{gen} : \mathit{neutr} \end{bmatrix} \vee \begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : \mathit{i} \\ \mathit{num} : \mathit{pl} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \mathit{masc} \end{bmatrix} \vee \begin{bmatrix} \mathit{pos} : \mathit{noun} \\ \mathit{decl} : \mathit{i} \\ \mathit{num} : \mathit{pl} \\ \mathit{case} : \mathit{acc} \\ \mathit{gen} : \mathit{neutr} \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
a \sqcup b = & \begin{bmatrix} \textit{pos} : \textit{noun} \\ \textit{decl} : \textit{i} \\ \textit{num} : \textit{pl} \\ \textit{case} : \textit{nom} \\ \textit{gen} : \begin{Bmatrix} \textit{masc} \\ \textit{neutr} \end{Bmatrix} \end{bmatrix} \vee \begin{bmatrix} \textit{pos} : \textit{noun} \\ \textit{decl} : \textit{i} \\ \textit{num} : \textit{pl} \\ \textit{case} : \textit{acc} \\ \textit{gen} : \textit{neutr} \end{bmatrix} \\
\equiv & \begin{bmatrix} \textit{pos} : \textit{noun} \\ \textit{decl} : \textit{i} \\ \textit{num} : \textit{pl} \\ \textit{case} : \textit{nom} \\ \textit{gen} : \textit{masc} \end{bmatrix} \vee \begin{bmatrix} \textit{pos} : \textit{noun} \\ \textit{decl} : \textit{i} \\ \textit{num} : \textit{pl} \\ \textit{case} : \textit{nom} \\ \textit{gen} : \textit{neutr} \end{bmatrix} \vee \begin{bmatrix} \textit{pos} : \textit{noun} \\ \textit{decl} : \textit{i} \\ \textit{num} : \textit{pl} \\ \textit{case} : \textit{acc} \\ \textit{gen} : \textit{neutr} \end{bmatrix}
\end{aligned}
$$

Appendix B

A Feature System for Latin Morphology

Tables B.1, B.2, and B.3 show the system of features for the analysis of nouns, adjectives, verbs, and particles, the parts of speech that are currently covered by the morphological analyzer. It is based on the classification of Hillen (1986), a Latin textbook grammar. Abbreviations in parentheses for attribute names correspond to abbreviations used in the text. Abbreviations in brackets for value names stand for abbreviations used in the endings and lexicon trie specification files. Appendix D lists all regular forms recognized by the morphological analyzer. In addition, uninflected particles are recognized. A brief explanation of features that need explanation follows.

regularity The category *regularity* serves to separate regular from irregular forms and is further described in Section 2.3.7.

declension The feature values in the category *declension* may seem unusual to some readers. Instead of numbering the declensional classes from one to five, first declension is called *a–declension*, second *o–declension*, fourth *u–declension*, and fifth *e–declension*. In addition, instead of having five main declensional classes, second declension is split into two distinct paradigms and third declension into three. For second declension, this is in accordance with the approach of most (German) school grammars (e.g. Hillen, 1986; Rubenbauer et al., 1989) and captures the regular deviations of nouns ending in -er (e.g. puer, *boy*, in contrast to filius, *son*). For third declension, the split is based on the classification

Table B.1: Features for more than one part of speech

attribute	value
part of speech (pos)	noun adjective verb particle
number (num)	singular plural
regularity (reg)	+ [regular] − [exception]

Table B.2: Features for declined forms

attribute	value
declension (decl)	a–declension [a_decl] o–declension [o_decl] o–declension, -er stems [o_puer_decl] consonant declension [cons_decl] i–declension [i_decl] mixed declension [mixed_decl] u–declension [u_decl] e–declension [e_decl]
gender (gen)	masculine feminine neuter
case	nominative genitive dative accusative ablative vocative

Table B.3: Features for verbs

attribute	value
conjugation (conj)	a–conjugation [a_conj] e–conjugation [e_conj] consonant conjugation [cons_conj] consonant conjugation, 1^{st} sg. -io [cons_i_conj] i–conjugation [i_conj] 0–conjugation, 1^{st} sg. -o, [null_conj_o] 0–conjugation, 1^{st} sg. -m, [null_conj_m]
person (pers)	1 [one] 2 [two] 3 [three] infinite
mode	indicative subjunctive imperative
stem	present stem [prs_stem] perfect stem [perf_stem] supine stem [sup_stem]
tense	present past future
voice	active passive
infinitivity (inf)	infinitive gerund gerundive participle supineI supineII

of Hillen (1986). In the case of nouns, lex, *law*, and tempus, *time*, are examples for *consonant declension*, turris, *tower*, and mare, *sea*, for *i–declension*, and urbs, *city*, and cor, *heart*, for *mixed declension* which shares characteristics of consonant declension and i–declension. For adjectives, Hillen distinguishes between *consonant declension* (dives, *rich*), and *i–declension* (acer, *sharp*). These two declensions are slightly different from their noun counterparts, although they bear the same names. All the subtypes of third declension do not show big differences, but in order to capture the few differences as a regular phenomenon in the morphological system, a further differentiation of the declensional paradigms is convenient. For an exact list of the forms in each declension see Appendix D.

conjugation Similarly to the treatment of declensions, conjugations are named a–conjugation (first conjugation), e–conjugation (second conjugation), consonant conjugation (third conjugation), and i–conjugation (fourth conjugation). Consonant conjugation has a variant for verbs like capere, *to grab*, which resembles i–conjugation (for details, see Appendix D). In addition, there are two "null" conjugations. These are artificial constructions, and they are only defined for forms of the present stem. Their purpose is to facilitate the description of irregular verbs in the lexicon, like esse, *to be* or ire, *to go*. A more sophisticated approach could encode these verbs into the endings trie directly, and would eliminate the need for two "ad–hoc" conjugations. Another advantage of a "direct encoding" approach would be that verbs like adesse or adire would not need to duplicate the complex esse and ire lexical entries but could use the special esse and ire inflectional paradigms in the endings trie in order to vastly facilitate the lexical entries for adesse, adire, and any other verbs derived from and ending in esse or ire.

person In addition to the three standard values for the *person* category, it can have the value *infinite* which designates infinite verb forms. Those forms are the only ones that receive a feature from the *infinitivity* category.

tense Instead of six different tenses as usual, only three are distinguished, but these are combined with the present stem and the perfect stem features to practically result in the six common tenses.

Appendix C

More Examples of Standard Lexical Entries

This appendix gives additional examples of typical lexical entries. Several of the examples also appear in Bubenheimer (1995).

	positive
templum	$\begin{bmatrix} reg : + \\ pos : noun \\ decl : o \\ gen : neutr \end{bmatrix}$
templ–	

An example for nouns from o– and a–declension, with usually only one stem and regular inflection.

	positive	negative 1genus{

	positive
asper	$\begin{bmatrix} \mathit{reg} : + \\ \mathit{pos} : \mathit{adj} \\ \mathit{decl} : \begin{Bmatrix} a \\ o\_puer \end{Bmatrix} \end{bmatrix}$
asper–	

An example for some of the adjectives from o– and a–declensions with –er ending. Only one stem is needed.

	positive	**negative 1**	**negative 2**
utilis	$\begin{bmatrix} \mathit{reg} : + \\ \mathit{pos} : \mathit{adj} \\ \mathit{decl} : i \end{bmatrix}$		
utilis–	$\begin{bmatrix} \mathit{case} : \begin{Bmatrix} \mathit{nom} \\ \mathit{voc} \end{Bmatrix} \\ \mathit{num} : \mathit{sg} \\ \mathit{gen} : \begin{Bmatrix} \mathit{masc} \\ \mathit{fem} \end{Bmatrix} \end{bmatrix}$		
utile–	$\begin{bmatrix} \mathit{case} : \begin{Bmatrix} \mathit{nom} \\ \mathit{acc} \\ \mathit{voc} \end{Bmatrix} \\ \mathit{num} : \mathit{sg} \\ \mathit{gen} : \mathit{neutr} \end{bmatrix}$		
util–		$\neg \begin{bmatrix} \mathit{case} : \begin{Bmatrix} \mathit{nom} \\ \mathit{voc} \end{Bmatrix} \\ \mathit{num} : \mathit{sg} \end{bmatrix}$	$\neg \begin{bmatrix} \mathit{case} : \mathit{acc} \\ \mathit{num} : \mathit{sg} \\ \mathit{gen} : \mathit{neutr} \end{bmatrix}$

An example for adjectives from i–declension with more than one nominative singular form. Usually, one stem entry is needed for every nominative singular form and one common stem entry for most other forms.

	positive
laudare	$\begin{bmatrix} \mathit{reg} : + \\ \mathit{pos} : \mathit{verb} \\ \mathit{conj} : a \end{bmatrix}$
laud–	$\begin{bmatrix} \mathit{stem} : \mathit{present} \end{bmatrix}$
laudav–	$\begin{bmatrix} \mathit{stem} : \mathit{perfect} \end{bmatrix}$
laudat–	$\begin{bmatrix} \mathit{stem} : \mathit{supine} \end{bmatrix}$

An example for verbs from a–declension, with usually three stems and very regular inflection.

Appendix D

Inflection Tables

The following tables list the word forms recognized by the morphological analyzer. This appendix is mainly a translation of material from Bubenheimer (1995) and largely follows Hillen (1986).

Finite verb forms are listed starting from first person singular through third person plural. Declined word forms are listed in the following order of cases with the exception of the gerund, for which nominative and vocative forms are not listed.

nominative
genitive
dative
accusative
ablative
vocative

D.1 Verbs

Table D.1: present stem active

	a–conj.	e–conj.	cons. conj.	cons. conj. (-io–stem)	i–conj.
indicative					
present	am–o am–as am–at am–amus am–atis am–ant	mon–eo mon–es mon–et mon–emus mon–etis mon–ent	reg–o reg–is reg–it reg–imus reg–itis reg–unt	cap–io cap–is cap–it cap–imus cap–itis cap–iunt	aud–io aud–is aud–it aud–imus aud–itis aud–iunt
past	am–abam am–abas am–abat am–abamus am–abatis am–abant	mon–ebam mon–ebas mon–ebat mon–ebamus mon–ebatis mon–ebant	reg–ebam reg–ebas reg–ebat reg–ebamus reg–ebatis reg–ebant	cap–iebam cap–iebas cap–iebat cap–iebamus cap–iebatis cap–iebant	aud–iebam aud–iebas aud–iebat aud–iebamus aud–iebatis aud–iebant
future	am–abo am–abis am–abit am–abimus am–abitis am–abunt	mon–ebo mon–ebis mon–ebit mon–ebimus mon–ebitis mon–ebunt	reg–am reg–es reg–et reg–emus reg–etis reg–ent	cap–iam cap–ies cap–iet cap–iemus cap–ietis cap–ient	aud–iam aud–ies aud–iet aud–iemus aud–ietis aud–ient
subjunctive					
present	am–em am–es am–et am–emus am–etis am–ent	mon–eam mon–eas mon–eat mon–eamus mon–eatis mon–eant	reg–am reg–as reg–at reg–amus reg–atis reg–ant	cap–iam cap–ias cap–iat cap–iamus cap–iatis cap–iant	aud–iam aud–ias aud–iat aud–iamus aud–iatis aud–iant
past	am–arem am–ares am–aret am–aremus am–aretis am–arent	mon–erem mon–eres mon–eret mon–eremus mon–eretis mon–erent	reg–erem reg–eres reg–eret reg–eremus reg–eretis reg–erent	cap–erem cap–eres cap–eret cap–eremus cap–eretis cap–erent	aud–irem aud–ires aud–iret aud–iremus aud–iretis aud–irent
imperative					
present	am–a am–ate	mon–e mon–ete	reg–e reg–ite	cap–e cap–ite	aud–i aud–ite
future	am–ato am–ato am–atote am–anto	mon–eto mon–eto mon–etote mon–ento	reg–ito reg–ito reg–itote reg–unto	cap–ito cap–ito cap–itote cap–iunto	aud–ito aud–ito aud–itote aud–iunto
infinitive	am–are	mon–ere	reg–ere	cap–ere	aud–ire

Table D.2: present stem passive

	a–conj.	e–conj.	cons. conj.	cons. conj. (-io–stem)	i–conj.
indicative					
present	am–or am–aris am–atur am–amur am–amini am–antur	mon–eor mon–eris mon–etur mon–emur mon–emini mon–entur	reg–or reg–eris reg–itur reg–imur reg–imini reg–untur	cap–ior cap–eris cap–itur cap–imur cap–imini cap–iuntur	aud–ior aud–iris aud–itur aud–imur aud–mini aud–iuntur
past	am–abar am–abaris am–abatur am–abamur am–abamini am–abantur	mon–ebar mon–ebaris mon–ebatur mon–ebamur mon–ebamini mon–ebantur	reg–ebar reg–ebaris reg–ebatur reg–ebamur reg–ebamini reg–ebantur	cap–iebar cap–iebaris cap–iebatur cap–iebamur cap–iebamini cap–iebantur	aud–iebar aud–iebaris aud–iebatur aud–iebamur aud–iebamini aud–iebantur
future	am–abor am–aberis am–abitur am–abimur am–abimini am–abuntur	mon–ebor mon–eberis mon–ebitur mon–ebimur mon–ebimini mon–ebuntur	reg–ar reg–eris reg–etur reg–emur reg–emini reg–entur	cap–iar cap–ieris cap–ietur cap–iemur cap–iemini cap–ientur	aud–iar aud–ieris aud–ietur aud–iemur aud–iemini aud–ientur
subjunctive					
present	am–er am–eris am–etur am–emur am–emini am–entur	mon–ear mon–earis mon–eatur mon–eamur mon–eamini mon–eantur	reg–ar reg–aris reg–atur reg–amur reg–amini reg–antur	cap–iar cap–iaris cap–iatur cap–iamur cap–iamini cap–iantur	aud–iar aud–iaris aud–iatur aud–iamur aud–iamini aud–iantur
past	am–arer am–areris am–aretur am–aremur am–aremini am–arentur	mon–erer mon–ereris mon–eretur mon–eremur mon–eremini mon–erentur	reg–erer reg–ereris reg–eretur reg–eremur reg–eremini reg–erentur	cap–erer cap–ereris cap–eretur cap–eremur cap–eremini cap–erentur	aud–irer aud–ireris aud–iretur aud–iremur aud–iremini aud–irentur
imperative					
present	am–are am–amini	mon–ere mon–emini	reg–ere reg–imini	cap–ere cap–imini	aud–ire aud–imini
future	am–ator am–ator am–antor	mon–etor mon–etor mon–entor	reg–itor reg–itor reg–untor	cap–itor cap–itor cap–iuntor	aud–itor aud–itor aud–iuntor
infinitive	am–ari	mon–eri	reg–i	cap–i	aud–iri

Table D.3: additional infinite forms for the present stem

	a–conj.	e–conj.	cons. conj.	cons. conj. (-io–stem)	i–conj.
participle (participle present active)	am–ans am-antis etc. (see below, mixed declension)	mon–ens mon–entis etc. (see below, mixed declension)	reg–ens reg–entis etc. (see below, mixed declension)	cap–iens cap–ientis etc. (see below, mixed declension)	aud–iens aud–ientis etc. (see below, mixed declension)
gerund	am–andi am–ando am–andum am–ando	mon–endi mon–endo mon–endum mon–endo	reg–endi reg–endo reg–endum reg–endo	cap–iendi cap–iendo cap–iendum cap–iendo	aud–iendi aud–iendo aud–iendum aud–iendo
gerundive	am–andus, a,um etc. (a–/o–decl. for adj.)	mon–endus, a,um etc. (a–/o–decl. for adj.)	reg–endus, a,um etc. (a–/o–decl. for adj.)	cap–iendus, a,um etc. (a–/o–decl. for adj.)	aud–iendus, a,um etc. (a–/o–decl. for adj.)

Table D.4: perfect stem active

indicative	
present (perfect)	amav–i amav–isti amav–it amav–imus amav–istis amav–erunt
past (pluperfect)	amav–eram amav–eras amav–erat amav–eramus amav–eratis amav–erant
future (future perfect)	amav–ero amav–eris amav–erit amav–erimus amav–eritis amav–erint
subjunctive	
present (perfect)	amav–erim amav–eris amav–erit amav–erimus amav–eritis amav–erint
past (pluperfect)	amav–issem amav–isses amav–isset amav–issemus amav–issetis amav–issent
infinitive	amav–isse

Table D.5: supine stem

participle present (participle perf. passive)	amat–us,a,um etc. (a–/o–decl. for adjectives)
participle future (participle fut. active)	amat–urus,a,um etc. (a–/o–decl. for adjectives)
supine I	amat–um
supine II	amat–u

Table D.6: declension of present participle

	singular		plural	
	masc., fem.	neutr.	masc., fem.	neutr.
mixed decl.	am–ans	am–ans	am–antes	am–antia
(3^{rd} decl.)	am–antis	am–antis	am–antium	am–antium
	am–anti	am–anti	am–antibus	am–antibus
	am–antem	am–ans	am–antes/–antis	am–antia
	am–ante	am–ante	am–antibus	am–antibus
	am–ans	am–ans	am–antes	am–antia

Table D.7: 0–conjugation paradigms (only present stem)

	1^{st} sg. -o		1^{st} sg. -m	
	active	passive	active	passive
indicative,	–o	–or	–m	–r
subjunctive	–s	–ris	–s	–ris
	–t	–tur	–t	–tur
	–mus	–mur	–mus	–mur
	–tis	–mini	–tis	–mini
	–unt	–untur	–nt	–ntur
imperative				
present	–ϵ	–re	–ϵ	–re
	–te	–mini	–te	–mini
imperative				
future	–to	–tor	–to	–tor
	–to	–tor	–to	–tor
	–tote		–tote	
	–unto	–untor	–nto	–ntor

D.2 Nouns

Table D.8: noun declension

	masc.,fem.		neutr.	
	singular	plural	singular	plural(-

D.3 Adjectives

Table D.9: adjective declension

	singular			plural		
	masc.	fem.	neutr.	masc.	fem.	neutr.
a–/o–decl.	iust–us	iust–a	iust–um	iust–i	iust–ae	iust–a
(fem.:	iust–i	iust–ae	iust–i	iust–orum	iust–arum	iust–orum
a–decl.;	iust–o	iust–ae	iust–o	iust–is	iust–is	iust–is
masc./neutr.:	iust–um	iust–am	iust–um	iust–os	iust–as	iust–a
o–decl.)	iust–o	iust–a	iust–o	iust–is	iust–is	iust–is
	iust–e	iust–a	iust–um	iust–i	iust–ae	iust–a
a–/o–decl.	pulcher	pulchr–a	pulchr–um	pulchr–i	pulchr–ae	pulchr–a
(-er stem)	pulchr–i	pulchr–ae	pulchr–i	pulchr–orum	pulchr–arum	pulchr–orum
(fem.:	pulchr–o	pulchr–ae	pulchr–o	pulchr–is	pulchr–is	pulchr–is
a–decl.;	pulchr–um	pulchr–am	pulchr–um	pulchr–os	pulchr–as	pulchr–a
masc./neutr.:	pulchr–o	pulchr–a	pulchr–o	pulchr–is	pulchr–is	pulchr–is
o–decl. (-er))	pulcher	pulchr–a	pulchr–um	pulchr–i	pulchr–ae	pulchr–a
cons. decl.	dives	dives	dives	divit–es	divit–es	divit–a
	divit–is	divit–is	divit–is	divit–um	divit–um	divit–um
	divit–i	divit–i	divit–i	divit–ibus	divit–ibus	divit–ibus
	divit–em	divit–em	dives	divit–es	divit–es	divit–a
	divit–e	divit–e	divit–e	divit–ibus	divit–ibus	divit–ibus
	dives	dives	dives	divit–es	divit–es	divit–a
i–decl.	acer	acris	acre	acr–es	acr–es	acr–ia
	acr–is	acr–is	acr–is	acr–ium	acr–ium	acr–ium
	acr–i	acr–i	acr–i	acr–ibus	acr–ibus	acr–ibus
	acr–em	acr–em	acre	acr–es/–is	acr–es/–is	acr–ia
	acr–i	acr–i	acr–i	acr–ibus	acr–ibus	acr–ibus
	acer	acris	acre	acr–es	acr–es	acr–ia

Appendix E

Endings Trie

The following list represents YALL's complete endings trie. Compared to Bubenheimer (1995), a clearer format is used and a few trie nodes have changed. Each line represents one trie node, including the node's number, covered grapheme, features, and parent nodes (for an explanation of the actual endings trie structure see Section 3.4.3); it has the form [*node number*,*morph*,*feature values*,*parent nodes*]. This is not the format of the endings trie specification file `trie.ini`, which is not equally readable. For a list of the used features see Appendix B. The last node listed is the trie's "start node."

```
[0,ϵ,verb,]
[1,ϵ,prs_stem,0]
[2,ϵ,perf_stem active,0]
[3,ϵ,sup_stem,0]
[4,ϵ,a_conj,1]
[5,ϵ,e_conj,1]
[6,ϵ,cons_conj,1]
[7,ϵ,cons_i_conj,1]
[8,ϵ,i_conj,1]
[9,ϵ,null_conj_o,1]
[10,ϵ,null_conj_m,1]
[11,a,,4]
[12,e,,5]
[13,i,,6]
[14,e,,6]
[15,i,,7]
[16,e,,7]
[17,i,,8]
```

```
[18,e,,15 17]
[19,re,,11 12 14 16 17]
[20,isse,,2]
[21,er,,2]
[22,ϵ,indicative subjunctive,9]
[23,ϵ,indicative subjunctive,10]
[24,ϵ,present indicative,11 12 13 15 17]
[25,ϵ,present indicative,4 6 12 15 17]
[26,ϵ,present indicative,541539.634.9(15)-509.9(17])]TJ'Ω-2.4301␣71.21␣TD'Ω([24,)

     [24,ϵ
```

```
[61,ris,singular passive two,22 23 28 29 32 34 35 36 37]
[62,ϵ,singular two,40 47 48]
[63,re,singular two passive,38]
[64,t,singular active three,22 23 24 27 29 32 33 35 36 50 52 53]
[65,it,singular active three,49]
[66,tur,singular passive three,22 23 24 29 32 33 35 36 37]
[67,ϵ,singular three,47 48]
[68,mus,plural active one,22 23 24 29 32 33 35 36 37 50 52 53]
[69,imus,plural active one,49]
[70,mur,plural passive one,22 23 24 29 32 33 35 36 37]
[71,tis,plural active two,22 23 24 29 32 33 35 36 37 50 52 53]
[72,istis,plural active two,49]
[73,mini,plural passive two,22 23 24 29 32 33 35 36 37 39]
[74,te,plural two,40 47]
[75,u,,22 26 30]
[76,nt,plural active three,23 27 29 32 35 36 37 50 52 53 75]
[77,erunt,plural active three,49]
[78,ntur,plural passive three,23 27 29 32 35 36 37 75]
[79,ϵ,three plural,47 48]
[80,ϵ,infinite infinitive active,19 20]
[81,r,,11 12 17]
[82,i,infinite infinitive passive,6 7 81]
[83,n,infinite participle masculine feminine neuter,11 12 14 18]
[84,s,nominative vocative singular masculine feminine,83]
[85,s,nominative accusative vocative singular neuter,83]
[86,t,singular genitive dative ablative,83]
[87,t,singular accusative masculine feminine,83]
[88,t,plural,83]
[89,ϵ,infinite participle present,3]
[90,ur,infinite participle future,3]
[91,nd,infinite gerund,11 12 14 18]
[92,i,genitive,91]
[93,o,dative ablative,91]
[94,um,accusative,91]
[95,nd,infinite gerundive,11 12 14 18]
[96,um,infinite supineI,3]
[97,u,infinite supineII,3]
[98,ϵ,noun,]
[99,ϵ,adjective,]
[100,ϵ,masculine neuter,89 90 95 99]
[101,ϵ,masculine neuter,99]
[102,ϵ,feminine,89 90 95 99]
[103,ϵ,masculine feminine,98]
```

```
[104,ϵ,masculine feminine neuter,98]
[105,ϵ,masculine feminine neuter,99]
[106,ϵ,o_decl,100 104]
[107,ϵ,o_puer_decl,101 103]
[108,ϵ,a_decl,102 103]
[109,ϵ,e_decl,103]
[110,ϵ,u_decl,104]
[111,ϵ,cons_decl,104 105]
[112,ϵ,i_decl,104 105]
[113,ϵ,mixed_decl,84 85 86 87 88 104]
[114,o,,106]
[115,u,,106]
[116,o,,107]
[117,u,,107]
[118,a,,108]
[119,e,,109]
[120,u,,110]
[121,i,,111]
[122,e,,111]
[123,i,,112]
[124,e,,112]
[125,i,,113]
[126,e,,113]
[127,ϵ,nominative accusative vocative singular neuter,
 111 112 113 120]
[128,m,nominative accusative vocative singular neuter,115 117]
[129,s,nominative vocative singular masculine feminine,119 120]
[130,ϵ,nominative vocative singular masculine feminine,
 107 111 112 113 118]
[131,s,nominative singular masculine feminine,115]
[132,i,genitive singular,106 107 119]
[133,e,genitive singular,118]
[134,s,genitive singular,120 121 123 125]
[135,ϵ,dative singular,114 116]
[136,e,dative singular,118]
[137,i,dative singular,111 112 113 119 120]
[138,ϵ,dative singular neuter,120]
[139,ϵ,noun,123]
[140,ϵ,adjective,124]
[141,m,accusative singular masculine feminine,
 115 117 118 119 120 122 126 139 140]
[142,ϵ,ablative singular,114 116 118 119 120 122 123 126]
[143,e,vocative singular masculine feminine,106]
```

```
[144,ϵ,verb,125]
[145,ϵ,noun,113]
[146,a,nominative accusative vocative plural neuter,
 106 107 111 120 123 144 145]
[147,s,nominative accusative vocative plural masculine feminine,
 119 120 122 124 126]
[148,i,nominative vocative plural masculine feminine,106 107]
[149,e,nominative vocative plural,118]
[150,r,,114 116 118 119]
[151,um,genitive plural,111 120 123 125 150]
[152,i,,110]
[153,is,dative ablative plural,106 107 108]
[154,bus,dative ablative plural,119 121 123 125 152]
[155,s,accusative masculine feminine plural,114 116 118 123 125]
[156,ϵ,particle,]
[157,ϵ,regular,
 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 76 77 78
 79 80 82 92 93 94 96 97 127 128 129 130 131 132 133 134 135 136 137
 138 141 142 143 146 147 148 149 151 153 154 155 156]
[158,ϵ,exception,]
[159,ϵ,,157 158]
```

Appendix F

Source Code

This appendix presents the complete source code of YALL's reimplementation in Java.

Affix.java

```
package yall;

/**
 * Combines a morph and a feature structure into a morphological
 * description of the "morph meaning".
 */
class Affix {
    public String grapheme;
    public FeatureMap affixfeaturemap;

    /*
      public Affix() {
      }
    */

    /*
      public Affix(String graphemestring, FeatureMap thefeaturemap) {
      this();
      grapheme = graphemestring;
      affixfeaturemap = thefeaturemap;
      }
    */
}
```

AnalysisResult.java

```
package yall;

/**
 * Data structure to specify the results of a morphological analysis
 * with the the found lemma, the found stem, and the found features.
 */
class AnalysisResult {
    String lemmaname;
    String stemname;
    SimpleFeatureMap features;

    public String toString () {
        return "Lemma: " + lemmaname + "\nStem: " + stemname + "\n" +
            features.toString();
    }
}
```

Automat.java

```
package yall;

import java.util.LinkedList;
import java.util.ListIterator;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;

/**
 * The main class for the endings trie.
 */
final class Automat{
    ExtAffix mat;

    /*
      file format of automat file:
      1. 1-n entries
      2. Each entry consists of several lines:
      1. identifier for entry (standard String)
      2. affix at node
      3. one line for each feature value,
      blank line indicates end of feature values
      4. one line for each successor node identifier,
      blank line indicates end of successor nodes
      3. The last entry is the start node of the trie
    */

    public Automat(){
        LinkedList tempentrylist = new LinkedList();

        try {
            BufferedReader trieini =
                new BufferedReader(new FileReader("trie.ini"));
            try {
                do {
                    TempEntry newentry = new TempEntry();
                    newentry.identifier = trieini.readLine();
                    newentry.node = new ExtAffix();
                    newentry.node.successors = new LinkedList();
                    newentry.node.grapheme = trieini.readLine();

                    newentry.node.affixfeaturemap = new FeatureMap();
                    String valuestring;
                    while ((valuestring =
                            trieini.readLine()).length() > 0) {
                        newentry.node.affixfeaturemap.
                            put(new Feature(valuestring));
                    }

                    newentry.nextentries = new LinkedList();
                    String idstring;
                    while ((idstring = trieini.readLine()).length() > 0) {
                        newentry.nextentries.add(idstring);
                    }
                    tempentrylist.addLast(newentry);

                } while(trieini.ready());
                // do-while: at least one entry (start node) required
                // in file!
                trieini.close();

            } catch (IOException e) {
                throw new Error("Wrong file format in trie.ini!");
            }
        } catch (FileNotFoundException e) {
            throw new Error("trie.ini could not be found!");
        }


        ListIterator entryiterator = tempentrylist.listIterator(0);
        do {
            TempEntry theentry = (TempEntry) entryiterator.next();
            ListIterator iditerator =
                theentry.nextentries.listIterator(0);
            while(iditerator.hasNext()) {
                String theid = (String) iditerator.next();
                boolean idfound = false;
                ListIterator otherentryiterator =
                    tempentrylist.listIterator(0);
                do {
                    TempEntry theotherentry =
                        (TempEntry) otherentryiterator.next();
                    if(theid.equals(theotherentry.identifier)) {
                        theentry.node.successors.add(theotherentry.node);
                        idfound = true;
                        break;
                    }
                } while (otherentryiterator.hasNext());
                if(!idfound) {
```

```
                    throw new Error("unknown node identifier: " + theid);
                }
            }
        } while (entryiterator.hasNext());

        mat = ((TempEntry) (tempentrylist.getLast())).node;
    }

    public LinkedList analyze(final String word) throws NoMatch {
        return mat.partmatch(word, (byte) (word.length() - 1),
                             new FeatureMap());
    }

}
```

BaseTrie.java

```
package yall;

import java.io.RandomAccessFile;
import java.io.IOException;
import java.io.FileNotFoundException;

import java.util.LinkedList;
import java.util.Iterator;

/**
 * The fundamental class for all lexicon tries.
 */
class BaseTrie {
    // and here comes the ugly workaround...
    private RandomAccessFile superFile;

    private EntryListFile entrylistfile;
    public EntryFile entryfile;

    private static final byte arrayentries = 27;
    // 26 characters of the Western Alphabet plus one "empty" letter

    private String fileSuffix = ".yt";
    // it's important to have at least one array in file

    public BaseTrie() {
        // only for creating an empty BaseTrie, needs to be followed
        // by create()
        entrylistfile = new EntryListFile();
        entryfile = new EntryFile();
    }

    public BaseTrie(String filename) {
        try {
            superFile = new RandomAccessFile(filename + fileSuffix, "r");
        }
        catch(FileNotFoundException e) {
            throw new Error("File " + filename + fileSuffix +
                            " does not exist yet. " +
                            "Please create it first.");
        }
        entrylistfile = new EntryListFile(filename);
        entryfile = new EntryFile(filename);
    }
```

```
public void create(String filename) {
    try {
        superFile = new RandomAccessFile(filename + fileSuffix, "rw");
        superFile.setLength(0);
        appendBlankArray();
    }
    catch(IOException f) {
        throw new Error("Couldn’t create new file" + filename +
                        fileSuffix);
    }
    entrylistfile.create(filename);
    entryfile.create(filename);
}

private int appendBlankArray() {
    try {
        int appendpos = (int) superFile.length();
        superFile.seek(appendpos);
        for(int i = 1; i <= arrayentries; i++) {
            superFile.writeInt(-1);
        }
        // I’m not sure if this works or if I have to
        // extend the size of the file first
        return appendpos;
    }
    catch(IOException e) {
        throw new Error("An error occured while trying to append an" +
                        "entry array to a trie file.\n" + e);
    }
}

public int arraypos(final String affixstring, final byte aktchar,
                    final int startpos) {
    return startpos + 4 * (affixstring.charAt(aktchar) - ’a’ + 1);
    // needs to be corrected!
}

/*
  exactness:
  0: exact find
  1: find prefixes of findaffix
  2: find completions of findaffix

  When trie file is created, every list in it must have the
  "empty" character as the character for the first element (makes
  retrieval easier)
*/


/* an old find() with lists
   public LinkedList find(final String findaffix, final int aktchar,
   final long findpos, final byte exactness)
   throws IOException {
   LinkedList poslist = new LinkedList();

   if(findpos == 0)
   return poslist;

   seek(findpos);
   byte infixchar = readByte();
   long aktelepos = readLong();
   long nextpos = readLong();

   if (infixchar == 0) {
   if ((exactness == 0 && aktchar == findaffix.length()) ||
   (exactness == 1 && aktchar <= findaffix.length()) ||
   (exactness == 2 && aktchar >= findaffix.length())) {
   poslist.addAll(getentrypositions(aktelepos));
   }
   if (exactness == 2 ||
   (exactness == 1 && aktchar < findaffix.length())) {
   poslist.addAll(find(findaffix, aktchar, nextpos, exactness));
   }
   } else if ((aktchar < findaffix.length() &&
   infixchar == findaffix.charAt(aktchar)) ||
   // some "casting" needed here!!!
   exactness == 2 ) {
   poslist.addAll(find(findaffix, aktchar + 1, aktelepos, exactness));
   } else {
   poslist.addAll(find(findaffix, aktchar, nextpos, exactness));
   }
   return poslist;
   }
*/

public LinkedList findEntries (final String findaffix,
                               final byte exactness) {
    LinkedList poslist = find(findaffix, exactness);
    Iterator positerator = poslist.iterator();
    LinkedList entryList = new LinkedList();
    for (int i = 0; i < poslist.size(); i++) {
        entryList.add(entryfile.find(((Integer) positerator.next()).
                                     intValue()));
    }
    return entryList;
}

public LinkedList find(final String findaffix, final byte exactness) {
    return find(findaffix, (byte) 0, 0, exactness);
}

private LinkedList find(final String findaffix, final byte aktchar,
                        final int findpos, final byte exactness) {
    LinkedList poslist = new LinkedList();
    try {
        if(findpos != -1) {
            if ((exactness == 0 && aktchar == findaffix.length()) ||
                (exactness == 1 && aktchar <= findaffix.length()) ||
                (exactness == 2 && aktchar >= findaffix.length())) {
                superFile.seek(findpos);
                int listpos = superFile.readInt();
                poslist.addAll(entrylistfile.
                               getentrypositions(listpos));
            }
            if (aktchar < findaffix.length()) {
                superFile.seek(arraypos(findaffix, aktchar, findpos));
                int nextpos = superFile.readInt();
                poslist.addAll(find(findaffix, (byte) (aktchar + 1),
                                    nextpos, exactness));
            }
            else if (exactness == 2)
            {
                for(int i = 0; i < arrayentries; i++) {
                    int nextpos = superFile.readInt();
                    int oldpos = (int) superFile.getFilePointer();
                    poslist.addAll(find(findaffix,(byte)(aktchar + 1),
                                        nextpos, exactness));
                    superFile.seek(oldpos);
                }
            }
        }
    }
    catch(IOException e) {
        throw new Error("BaseTrie.find() found a problem " +
                        "with the lexicon!");
    }
    return poslist;
```

```
}

public void updateEntry(LexEntry entry) {
    // only meant to update the lexicon entry’s file positions of
    // the linked lemmas/stems!!! so don’t change anything else in
    // entry!!! And leave the number of linked lemmas/stems the
    // same!!!
    try {
        entryfile.update(entry);
    }
    catch (IOException e) {
        throw new Error("BaseTrie.update(LexEntry) found a problem " +
                        "with the lexicon!");
    }
}

// this insert also sets entry.entrypos to the right position and
// returns that same position
public void insert(LexEntry entry) {
    try {
        int pos = entryfile.append(entry);
        insert(entry.name, pos);
    }
    catch (IOException e) {
        throw(new Error("BaseTrie.insert(LexEntry) found a problem " +
                        "with the lexicon!"));
    }
}

public void insert(final String affixstring, final int lexipos) {
    insert(affixstring, lexipos, (byte) 0, 0);
}

private int insert(final String affixstring, final int lexipos,
                   final byte aktchar, final int insertpos) {

    int realinsertpos;

    if(insertpos != -1) {
        realinsertpos = insertpos;
    }
    else {
        realinsertpos = appendBlankArray();
    }

    try {
        if(aktchar == affixstring.length()) {
            superFile.seek(realinsertpos);
            int listpos = superFile.readInt();
            int listrealpos = entrylistfile.insert(listpos, lexipos);
            if(listpos == -1) {
                superFile.seek(realinsertpos);
                superFile.writeInt(listrealpos);
            }
        } else {
            int charinsertpos = arraypos(affixstring, aktchar,
                                         realinsertpos);
            superFile.seek(charinsertpos);
            int readinsertpos = superFile.readInt();
            int returninsertpos = insert(affixstring, lexipos,
                                         (byte) (aktchar + 1),
                                         readinsertpos);
            if(readinsertpos == -1) {
                superFile.seek(charinsertpos);
                superFile.writeInt(returninsertpos);
            }
        }
    }
    catch (IOException e) {
        throw new Error("BaseTrie.insert encountered a problem " +
                        "with the lexicon!");
    }
    return realinsertpos;
}

public void delete(final String affixstring, final int lexipos) {
    delete(affixstring, lexipos, (byte) 0, 0);
}

private void delete(final String affixstring, final int lexipos,
                    final byte aktchar, final int triepos) {
    if(triepos != -1) {
        try {
            if(affixstring.length() == aktchar) {
                superFile.seek(triepos);
                int listpos = superFile.readInt();
                int deletepos =
                    entrylistfile.delete(listpos, lexipos);
                if(deletepos != -1) {
                    superFile.seek(triepos);
                    superFile.writeInt(deletepos);
                }
                }
                else {
                    int chardeletepos = arraypos(affixstring, aktchar,
                                                 triepos);
                    superFile.seek(chardeletepos);
                    int readdeletepos = superFile.readInt();
                    delete(affixstring, lexipos, (byte) (aktchar + 1),
                           readdeletepos);
                }
            }
            catch (IOException e) {
                throw new Error("BaseTrie.delete found a problem " +
                                "with the lexicon!");
            }
        }
    }
}
```

Category.java

```
package yall;

/**
   Used to specify feature categories.
 */
final class Category {
    private byte index;

    private static Category [] object = {new Category((byte)0),
                                         new Category((byte)1),
                                         new Category((byte)2),
                                         new Category((byte)3),
                                         new Category((byte)4),
                                         new Category((byte)5),
                                         new Category((byte)6),
                                         new Category((byte)7),
                                         new Category((byte)8),
                                         new Category((byte)9),
                                         new Category((byte)10),
                                         new Category((byte)11),
                                         new Category((byte)12),
                                         new Category((byte)13)};

    private Category (byte index) {
        this.index = index;
    }

    public byte index () {
        return index;
    }

    public static Category object (byte index) {
        return object[index];
    }
}
```

CategoryValue.java

```
package yall;

/**
 * The class to specify category-value pairs. Contains fields for all
 * possible categories and values.
 */
class CategoryValue {
    // categories
    public static int partofspeech = 1;
    public static int conjugation = 2;
    public static int person = 3;
    public static int number = 4;
    public static int infinitivity= 5;
    public static int mode = 6;
    public static int stem = 7;
    public static int tense = 8;
    public static int voice = 9;
    public static int declension = 10;
    public static int casus = 11; // casus since case is Java term
    public static int gender = 12;
    public static int regularity = 13;

    // partofspeech
    public static int noun = 1;
    public static int verb = 2;
    public static int adjective = 3;
    public static int particle = 4;

    // conjugation
    public static int a_konj = 1;
    public static int e_konj = 2;
    public static int kons_konj = 3;
    public static int kons_i_konj = 4;
    public static int i_konj = 5;
    public static int null_konj_o = 6;
    public static int null_konj_m = 7;

    // person
    public static int one = 1;
    public static int two = 2;
    public static int three = 3;
    public static int infinite = 4;

    // number
```

```
public static int singular = 1;
public static int plural = 2;

// infinitivity
public static int infinitive = 1;
public static int gerund = 2;
public static int gerundive = 3;
public static int participle = 4;
public static int supineI = 5;
public static int supineII = 6;

// mode
public static int indicative = 1;
public static int subjunctive = 2;
public static int imperative = 3;

// stem
public static int prs_stem = 1;
public static int pref_stem = 2;
public static int sup_stem = 3;

// tense
public static int present = 1;
public static int past = 2;
public static int future = 3;

// voice
public static int active = 1;
public static int passive = 2;

// declension
public static int a_decl = 1;
public static int o_decl = 2;
public static int o_puer_decl = 3;
public static int e_decl = 4;
public static int u_decl = 5;
public static int cons_decl = 6;
public static int i_decl = 7;
public static int mixed_decl = 8;

// casus
public static int nominative = 1;
public static int genitive = 2;
public static int dative = 3;
public static int accusative = 4;
public static int ablative = 5;
```

```
    public static int vocative = 6;

    // gender
    public static int masculine = 1;
    public static int feminine = 2;
    public static int neuter = 3;

    // regularity
    public static int regular = 1;
    public static int exception = 2;

    public int category;
    public int value;

    public CategoryValue(int cat, int val){
        category = cat;
        value = val;
    }
}
```

EntryFile.java

```
package yall;

// here we get to a big, big problem in Java... IT IS NOT POSSIBLE TO
// CATCH AN EXCEPTION FROM THE CONSTRUCTOR IN THE SUPERCLASS IN THE
// CONSTRUCTOR OF THE DERIVED CLASS! ALL DERIVED CLASSES HAVE TO THROW
// THE SAME EXCEPTION. ALSO: REQUIRED INVOCATION OF CONSTRUCTOR OF
// SUPERCLASS AS FIRST THING IN THE CONSTRUCTOR OF DERIVED CLASS

// another braindead idea is to make the classes String, Byte, etc.
// final, so they can't be used for derivations

import java.io.IOException;
import java.io.FileNotFoundException;

/**
 * The class to handle the lexicon entry files (.ye).
 */
class EntryFile {

    ExtRandomAccessFile superFile;
    // workaround: I don't
    // derive it from the superclass, I make the superclass a
    // member...

    private static String fileSuffix = ".ye";

    class FilePosLink {
        int startpos, headlinkpos;
    }

    public EntryFile() {
        // only for creating an empty EntryFile, needs to be followed
        // by create()
    }

    public EntryFile(String name) {
        try {
            superFile = new ExtRandomAccessFile(name + fileSuffix, "r");
        }
        catch(FileNotFoundException e) {
            throw new Error("File " + name + fileSuffix +
                            " does not exist yet. " +
                            "Please create it first.");
        }
    }

    public void create(String filename) {
        try {
            superFile = new ExtRandomAccessFile(filename + fileSuffix,
                                                "rw");
            superFile.setLength(0);
        }
        catch(IOException f) {
            throw new Error("Couldn’t create new file" + filename +
                            fileSuffix);
        }
    }

    public LexEntry find (final int findpos) {
        LexEntry anentry = new LexEntry();
        try {
            superFile.seek(findpos);
            anentry.fileRead(superFile);
        }
        catch(IOException e) {
            throw new Error("Something is wrong with the lexicon " +
                            "files; couldn’t read lexicon entry at" +
                            "specified location.");
        }
        return anentry;
    }

    // append also sets newentry.entrypos to the correct position
    // and returns that position
    int append (LexEntry newentry) throws IOException {
        newentry.entrypos = new Integer((int) superFile.length());
        superFile.seek(newentry.entrypos.intValue());
        newentry.fileWrite(superFile);
        return newentry.entrypos.intValue();
    }

    void update (LexEntry theentry) throws IOException {
        superFile.seek(theentry.entrypos.intValue());
        theentry.fileWrite(superFile);
    }
}
```

EntryListFile.java

```
package yall;

import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.util.LinkedList;

/**
 * The class to handle the files with lists of pointers to lexicon
 * entries in the lexicon entry files. The lexicon tries contain
 * pointers to heads of lists in the entry list files.
 */
class EntryListFile {
    // and another one of these workarounds...
    RandomAccessFile superFile;

    private static String fileSuffix = ".yel";

    public EntryListFile() {
        // only for creating an empty EntryListFile, needs to be
        // followed by create()
    }

    public EntryListFile (String filename) {
        try {
            superFile = new RandomAccessFile(filename + fileSuffix, "r");
        }
        catch (FileNotFoundException e) {
            // this should only occur when opening for reading
            throw new Error("File " + filename + fileSuffix +
                            " does not exist yet. " +
                            "Please create it first.");
        }
    }

    public void create(String filename) {
        try {
            superFile = new RandomAccessFile(filename + fileSuffix, "rw");
            superFile.setLength(0);
        }
        catch(IOException f) {
            throw new Error("Couldn’t create new file" + filename +
                            fileSuffix);
        }
    }

    public LinkedList getentrypositions(final int entriespos)
        throws IOException {
        if (entriespos == -1) {
            return new LinkedList();
        } else {
            superFile.seek(entriespos);
            int nextpos = superFile.readInt();
            int lexentrypos = superFile.readInt();
            LinkedList returnlist = getentrypositions(nextpos);
            returnlist.add(new Integer(lexentrypos));
            return returnlist;
        }
    }

    public int insert(final int insertpos, final int lexentrypos)
        throws IOException {
        if(insertpos == -1) {
            int length = (int) superFile.length();
            superFile.seek(length);
            superFile.writeInt(-1);
            superFile.writeInt(lexentrypos);
            return(length);
        } else {
            superFile.seek(insertpos);
            int nextpos = superFile.readInt();
            int realinsertpos = insert(nextpos, lexentrypos);
            if(nextpos == -1) {
                superFile.seek(insertpos);
                superFile.writeInt(realinsertpos);
            }
            return insertpos;

            /* old stuff
               long runentrypos = insertpos;
               long oldpos;
               do {
               seek(runentrypos);
               oldpos = runentrypos;
               } while((runentrypos = readLong()) != -1);
               long length = length();
               seek(oldpos);
               writeLong(length);
               seek(length);
```

```
                writeLong(-1);
                writeLong(lexentrypos);
                return insertpos;
            */
        }
    }

    public int delete(final int insertpos, final int lexentrypos)
        throws IOException {
        if(insertpos == -1) {
            return -1;
        } else {
            superFile.seek(insertpos);
            int nextpos = superFile.readInt();
            int thislexpos = superFile.readInt();
            if(thislexpos == lexentrypos) {
                return nextpos;
            } else {
                int deletepos = delete(nextpos, lexentrypos);
                if(deletepos != -1) {
                    superFile.seek(insertpos);
                    superFile.writeInt(deletepos);
                }
                return -1;
            }
        }
    }
}
```

ExtAffix.java

```
package yall;

import java.util.LinkedList;
import java.util.Iterator;

/**
 * Objects of this class represent nodes in the endings trie which
 * include pointers to successor nodes.
 */
class ExtAffix extends Affix {
    LinkedList successors;
    //the following nodes in the trie

    LinkedList partmatch(String input, byte aktchar,
                         final FeatureMap solong) throws NoMatch {
        if(input.regionMatches(aktchar - grapheme.length() + 1,grapheme,0,
                               grapheme.length())) {
            FeatureMap newsolong;

            try {
                newsolong = solong.matchmap(affixfeaturemap);
            }
            catch(NotUnifiable e) {
                throw new NoMatch();
            }

            LinkedList helplist = new LinkedList();

            if(successors.size() == 0) {
                StringFeatureMap resultmap = new StringFeatureMap();
                resultmap.analysis = newsolong;
                resultmap.stem =
                    input.substring(0,aktchar - grapheme.length() + 1);
                helplist.add(resultmap);
                return helplist;
            }
            else {
                Iterator succiter = successors.listIterator(0);
                boolean somematch = false;

                for(int i = successors.size(); i > 0; i--) {
                    try {
                        helplist.
                            addAll(((ExtAffix)succiter.next()).
                                   partmatch(input, (byte)
                                             (aktchar-grapheme.length()),
                                             newsolong));
                        somematch = true;
                    }
                    catch(NoMatch e) {
                    }
                }
                if(somematch) {
                    return helplist;
                }
                else {
                    throw new NoMatch();
                }

            }
        }
        else {
            throw new NoMatch();
        }
    }
}
```

ExtLexEntry.java

```
package yall;

/**
 * This class represents lexicon entries and includes the entry’s
 * lemma string.
 */
class ExtLexEntry extends LexEntry {
    String lemmagrapheme;

    public ExtLexEntry(LexEntry oldentry, EntryFile lemmafile) {
        entrypos = oldentry.entrypos;
        name = oldentry.name;
        links = oldentry.links;

        LexEntry lemmaEntry =
            lemmafile.find(((Integer)links.getFirst()).intValue());
        lemmagrapheme = lemmaEntry.name;
        try {
            features = oldentry.features.matchmap(lemmaEntry.features);
        }
        catch(NotUnifiable e) {
            throw new Error("The positive feature sets for the stem " +
                            "entry " + name + " and the lemma entry " +
                            lemmaEntry.name + " are not compatible!");
        }
        nofeatures = new FeatureMapList(oldentry.nofeatures);
        nofeatures.addAll(lemmaEntry.nofeatures);
    }

    /*
      LinkedList matchanalysis(final FeatureMap analysis)
      throws NotUnifiable {
      // versucht, eine morphologische Analyse analysis mit einem
      // kombinierten Lemma-Stamm-Eintrag zu matchen; die so
      // gewonnenen gueltigen Analysen werden in einer Liste von
      // SimpleFeatureMaps zurueckgegeben

      // I don’t think this is a good way to approach the problem;
      // will start from domatch first!!!

      LinkedList biglist = new LinkedList();
      FeatureMap helpmap = analysis.matchmap(features);
      LinkedList tempresult =
      helpmap.matchnegmaplist(stementry.nofeatures);
      // etc. etc. (need to do matchnegmap/matchnegmaplist first)

      }
    */
}
```

ExtRandomAccessFile.java

```
package yall;

import java.io.RandomAccessFile;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.io.FileNotFoundException;

/**
 * This class is an extension of the standard {@link RandomAccessFile}
 * class and adds reading and writing of {@link String} objects.
 */
class ExtRandomAccessFile extends RandomAccessFile {
    public ExtRandomAccessFile(String name, String mode)
        throws FileNotFoundException {
        super(name,mode);
    }

    String readString() throws IOException {
        byte strlength = readByte();
        byte [] bytearray = new byte[strlength];
        readFully(bytearray);
        String returnstring;
        try{
            // encoding should have been ASCII but Kawa debugging
            // won't let me... UTF8 should include all ASCII chars
            // with same numbers (Java Developers Connection)
            returnstring = new String(bytearray, "UTF8");}
        catch(UnsupportedEncodingException e) {
            throw new Error("unsupported encoding");
        }
        return returnstring;
    }

    void writeString(final String outputstring) throws IOException {
        writeByte(outputstring.length());
        writeBytes(outputstring);
    }

    /* has been done in class Feature
       Feature readFeature() throws IOException {
       returnfeature = new Feature();
       returnfeature.category = readByte();
       returnfeature.value = readByte();
       return returnfeature;
       }

       void writeFeature(Feature outputfeature) throws IOException {
       writeByte(outputfeature.category);
       writeByte(outputfeature.value);
       }

       FeatureMap readFeatureMap() throws IOException {

       }

       void writeFeatureMap() throws IOException {
       }
    */
}
```

Feature.java

```
package yall;

import java.io.IOException;

// the java.lang object classes like Byte, Integer, etc., suck
// insofar as they don’t work together with the collection classes
// from java.util very well; e.g. in a set, it cannot be tested if a
// certain numeric value is in the set, unless one has made a unique
// object for every possible number (this is because equals() in Byte,
// Integer, etc., checks for real Object identity, not number
// identity!)
//
// Hence I will make unique Byte objects for every possible number for
// categories and values.

/**
 * This class implements simple attribute-value pairs.
 */
class Feature implements Filable {
    public Category category;
    public Value value;

    // class Byte instead of byte because of easier handling elsewhere

    public Feature() {
    }

    public Feature(String valuestring) {
        // constructs a feature from the given valuestring
        for (byte cat = 1; cat <= 13; cat++) {
            for (byte val = 1; val <= FeatureNames.valuenumbers[cat - 1];
                 val++) {
                if(valuestring.equals(FeatureNames.
                                      valuenames[cat - 1][val - 1])) {
                    category = Category.object(cat);
                    value = Value.object(val);
                    return;
                }
            }
        }
        throw new Error("Unknown feature value " + valuestring);
    }

    public void fileWrite(ExtRandomAccessFile f) throws IOException {
        f.writeByte(category.index());
        f.writeByte(value.index());
    }

    public void fileRead(ExtRandomAccessFile f) throws IOException {
        category = Category.object(f.readByte());
        value = Value.object(f.readByte());
    }
}
```

FeatureDescription.java

```
package yall;

/**
 * Objects of this class contain a textual description of an
 * attribute-value pair that is used in the specification files for
 * the endings trie and the lexicon.
 */
class FeatureDescription {
    int category;
    int value;
    String description;

    public FeatureDescription(int cat, int val, String desc) {
        category = cat;
        value = val;
        description = desc;
    }
}
```

FeatureMap.java

```
package yall;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Iterator;

import java.util.LinkedList;


import java.io.IOException;

/**
 * A class to implement feature structures with features containing
 * disjunctive feature values.
 */
class FeatureMap extends HashMap implements Filable {

    FeatureMap() {
        super();
    }

    FeatureMap(FeatureMap t) {
        super(t);
    }

    public void fileWrite (ExtRandomAccessFile f) throws IOException {
        Set mapset = entrySet();
        f.writeByte((byte) mapset.size());
        Iterator setiterator = mapset.iterator();
        while (setiterator.hasNext())
        {
            Map.Entry entry = (Map.Entry) setiterator.next();
            f.writeByte(((Category) entry.getKey()).index());
            byte valueno = (byte) ((ValueSet) entry.getValue()).size();
            f.writeByte(valueno);
            Iterator valueiterator =
                (((ValueSet) entry.getValue()).iterator());
            for (; valueno > 0; valueno--) {
                f.writeByte(((Value) valueiterator.next()).index());
            }
        }
    }
```

```
public void fileRead (ExtRandomAccessFile f) throws IOException {
    byte mapsize = f.readByte();
    for(; mapsize > 0; mapsize--)
    {
        Category category = Category.object(f.readByte());
        ValueSet newvalueset = new ValueSet();
        byte valueno = f.readByte();
        for(; valueno > 0; valueno--) {
            newvalueset.add(Value.object(f.readByte()));
        }
        put(category, newvalueset);
    }
}

public void put(Feature newfeature) {
    if(containsKey(newfeature.category)) {
        ValueSet thevalueset = (ValueSet) get(newfeature.category);
        thevalueset.add(newfeature.value);
        put(newfeature.category, thevalueset);
    }
    else {
        ValueSet newvalueset = new ValueSet();
        newvalueset.add(newfeature.value);
        put(newfeature.category, newvalueset);
    }
}

public FeatureMap matchmap (final FeatureMap othermap)
    throws NotUnifiable {

    FeatureMap newmap = new FeatureMap();

    Set thiscollection = entrySet();

    Iterator runner = thiscollection.iterator();

    while(runner.hasNext()) {
        Map.Entry thisentry = (Map.Entry) runner.next();
        Category thiscategory = (Category) thisentry.getKey();
        ValueSet thisvalueset = ((ValueSet) thisentry.getValue());

        if (othermap.containsKey(thiscategory)) {
            ValueSet othervalueset =
                ((ValueSet) othermap.get(thiscategory));
            ValueSet newvalueset = thisvalueset.match(othervalueset);
            newmap.put(thiscategory, newvalueset);
        } else {
            // category was not in the other FeatureMap:
            newmap.put(thiscategory, thisvalueset);
        }
    }

    Set othercollection = othermap.entrySet();

    Iterator otherrunner = othercollection.iterator();

    while(otherrunner.hasNext()) {
        Map.Entry otherentry = (Map.Entry) otherrunner.next();
        Category othercategory = (Category) otherentry.getKey();
        ValueSet othervalueset = ((ValueSet) otherentry.getValue());

        if (! newmap.containsKey(othercategory)) {
            newmap.put(othercategory, othervalueset);
        }
    }
    return newmap;

    /*
      Iterator otherrunner = othercollection.iterator();
      otherloop:
      while(otherrunner.hasNext()) {
      Map.Entry otherentry = (Map.Entry) otherrunner.next();
      byte othercategory = ((Byte) otherentry.getKey()).byteValue();
      if(thiscategory == othercategory) {
      ValueSet othervalueset = ((ValueSet) otherentry.getValue());
      ValueSet newvalueset = thisvalueset.match(othervalueset);
      newmap.put(thiscategory, newvalueset);
      continue thisloop;
      }
      }
      // category was not in the other FeatureMap:
      newmap.put(thiscategory, thisvalueset);
      }

      Collection newcollection = newmap.values();
      Iterator otherrunner = othercollection.iterator();

      outerloop:
      while(otherrunner.hasNext()) {
      Map.Entry otherentry = (Map.Entry) otherrunner.next();
      byte othercategory = ((Byte) otherentry.getKey()).byteValue();
      ValueSet othervalueset = ((ValueSet) otherentry.getValue());
```

```
      Iterator newrunner = newcollection.iterator();
      innerloop:
      while(newrunner.hasNext()) {
      Map.Entry newentry = (Map.Entry) newrunner.next();
      byte newcategory = ((Byte) newentry.getKey()).byteValue();
      if(othercategory == newcategory) {
      // category is already in new FeatureMap
      continue outerloop;
      }
      }
      // category was not in the new FeatureMap:
      newmap.put(othercategory, othervalueset);
      }
      return newmap;
    */
}

public LinkedList splitmap() {
    // splits up a FeatureMap into SimpleFeatureMaps with at
    // most one value per category
    //
    // needed for matchnegmap, etc.
    //
    // this procedure is a little computationally expensive (could
    // be improved for the case that a category has only one value
    // (but it's harder to program then))

    Set thiscollection = entrySet();
    Iterator thisrunner = thiscollection.iterator();

    LinkedList prevcollection = new LinkedList();
    prevcollection.add(new SimpleFeatureMap());
    do {
        LinkedList newcollection = new LinkedList();
        Map.Entry thisentry = (Map.Entry) thisrunner.next();
        Iterator thisvaluerunner =
            ((ValueSet) thisentry.getValue()).iterator();
        do {
            Value thisvalue = (Value) thisvaluerunner.next();
            Iterator prevrunner = prevcollection.iterator();
            while(prevrunner.hasNext()) {
                SimpleFeatureMap amap =
                    (SimpleFeatureMap) prevrunner.next();
                SimpleFeatureMap newmap;
                // we want to save some copying effort, so for the
                // last value in the valuelist we just use the
```

```
                    // FeatureMaps from prevcollection instead of
                    // copying them (without this measure they would
                    // get discarded and we would waste a lot of time
                    // when the FeatureMaps are very simple)
                    if(thisvaluerunner.hasNext()) {
                        newmap = new SimpleFeatureMap(amap);
                    }
                    else {
                        newmap = amap;
                    }
                    newmap.put((Category) thisentry.getKey(), thisvalue);
                    newcollection.add(newmap);
                }
            } while(thisvaluerunner.hasNext());
            prevcollection = newcollection;
        } while(thisrunner.hasNext());
        return prevcollection;
    }

    public String toString() {
        StringBuffer text = new StringBuffer("[\n");
        Set mapset = entrySet();
        Iterator setiterator = mapset.iterator();
        while (setiterator.hasNext()) {
            Map.Entry entry = (Map.Entry) setiterator.next();
            byte catindex = ((Category) entry.getKey()).index();
            ValueSet values = (ValueSet) entry.getValue();
            text.append(FeatureNames.categorynames[catindex - 1] + " : " +
                        values.toString(catindex) + "\n");
        }
        text.append("]\n");
        return new String(text);
    }
}
```

FeatureMapList.java

```java
package yall;

import java.util.LinkedList;
import java.util.Iterator;

import java.io.IOException;

/**
 * A class to implement lists of feature structures containing
 * features with disjunctive values.
 */
class FeatureMapList extends LinkedList implements Filable {

    public FeatureMapList() {
        super();
    }

    public FeatureMapList(FeatureMapList l) {
        super(l);
    }

    public void fileRead (ExtRandomAccessFile f) throws IOException {
        byte listlength = f.readByte();
        for (int i = 0; i < listlength; i++) {
            FeatureMap newmap = new FeatureMap();
            newmap.fileRead(f);
            add(newmap);
        }
    }

    public void fileWrite (ExtRandomAccessFile f) throws IOException {
        f.writeByte(size());
        Iterator runner = iterator();
        for (int i = 0; i < size(); i++) {
            ((FeatureMap) runner.next()).fileWrite(f);
        }
    }
}
```

FeatureNames.java

```
package yall;

/**
 * Maps numeric feature descriptions to names of attributes and values
 * for displaying feature structures.
 */
final class FeatureNames {
    // categories
    public final static int partofspeech = 1;
    public final static int conjugation = 2;
    public final static int person = 3;
    public final static int number = 4;
    public final static int infinitivity= 5; // very tentative...
    public final static int mode = 6;
    public final static int stem = 7;
    public final static int tense = 8;
    public final static int voice = 9;
    public final static int declension = 10;
    public final static int casus = 11; // casus since case is Java term
    public final static int gender = 12;
    public final static int regularity = 13;

    // partofspeech
    public final static int noun = 1;
    public final static int verb = 2;
    public final static int adjective = 3;
    public final static int particle = 4;

    // conjugation
    public final static int a_conj = 1;
    public final static int e_conj = 2;
    public final static int cons_conj = 3;
    public final static int cons_i_conj = 4;
    public final static int i_conj = 5;
    public final static int null_conj_o = 6;
    public final static int null_conj_m = 7;

    // person
    public final static int one = 1;
    public final static int two = 2;
    public final static int three = 3;
    public final static int infinite = 4;

    // number
```

```
public final static int singular = 1;
public final static int plural = 2;

// infinitivity
public final static int infinitive = 1;
public final static int gerund = 2;
public final static int gerundive = 3;
public final static int participle = 4;
public final static int supineI = 5;
public final static int supineII = 6;

// mode
public final static int indicative = 1;
public final static int subjunctive = 2;
public final static int imperative = 3;

// stem
public final static int prs_stem = 1;
public final static int perf_stem = 2;
public final static int sup_stem = 3;

// tense
public final static int present = 1;
public final static int past = 2;
public final static int future = 3;

// voice
public final static int active = 1;
public final static int passive = 2;

// declension
public final static int a_decl = 1;
public final static int o_decl = 2;
public final static int o_puer_decl = 3;
public final static int e_decl = 4;
public final static int u_decl = 5;
public final static int cons_decl = 6;
public final static int i_decl = 7;
public final static int mixed_decl = 8;

// casus
public final static int nominative = 1;
public final static int genitive = 2;
public final static int dative = 3;
public final static int accusative = 4;
public final static int ablative = 5;
public final static int vocative = 6;

// gender
public final static int masculine = 1;
public final static int feminine = 2;
public final static int neuter = 3;

// regularity
public final static int regular = 1;
public final static int exception = 2;

final static String [] categorynames = {
    "partofspeech",
    "conjugation",
    "person",
    "number",
    "infinitivity",
    "mode",
    "stem",
    "tense",
    "voice",
    "declension",
    "case",
    "gender",
    "regularity"
};

final static String [][] valuenames = {
    {
        "noun",
        "verb",
        "adjective",
        "particle"
    },
    {
        "a_conj",
        "e_conj",
        "cons_conj",
        "cons_i_conj",
        "i_conj",
        "null_conj_o",
        "null_conj_m"
    },
    {
        "one",
        "two",
        "three",
        "infinite"
    },
    {
        "singular",
        "plural"
    },
    {
        "infinitive",
        "gerund",
        "gerundive",
        "participle",
        "supineI",
        "supineII"
    },
    {
        "indicative",
        "subjunctive",
        "imperative"},
    {
        "prs_stem",
        "perf_stem",
        "sup_stem"
    },
    {
        "present",
        "past",
        "future"
    },
    {
        "active",
        "passive"
    },
    {
        "a_decl",
        "o_decl",
        "o_puer_decl",
        "e_decl",
        "u_decl",
        "cons_decl",
        "i_decl",
        "mixed_decl"
    },
    {
        "nominative",
        "genitive",
            "dative",
            "accusative",
            "ablative",
            "vocative"
        },
        {
            "masculine",
            "feminine",
            "neuter"
        },
        {
            "regular",
            "exception"
        }
    };
    final static int [] valuenumbers =
    {4, 7, 4, 2, 6, 3, 3, 3, 2, 8, 6, 3, 2};
}
```

Filable.java

```
package yall;

import java.io.IOException;

/**
 * Denotes classes that implement methods to write their contents to
 * an {@link ExtRandomAccessFile} and to read it back in.
 */
interface Filable {
    public void fileWrite(ExtRandomAccessFile f) throws IOException;
    public void fileRead(ExtRandomAccessFile f) throws IOException;
}
```

LexEntry.java

```java
package yall;

import java.util.LinkedList;
import java.util.Iterator;


import java.io.IOException;

/**
 * Objects of this class can contain a complete lexicon entry, but not
 * the entry’s lemma string.
 */
class LexEntry implements Filable {
    Integer entrypos;
    // an Integer object in order to make interweaving with links
    // field of other lexEntries easier
    String name;
    FeatureMap features;
    FeatureMapList nofeatures;
    LinkedList links;

    public void fileRead (ExtRandomAccessFile f) throws IOException {
        entrypos = new Integer((int) f.getFilePointer());
        name = f.readString();
        features = new FeatureMap();
        features.fileRead(f);
        nofeatures = new FeatureMapList();
        nofeatures.fileRead(f);

        byte listlength = f.readByte();
        links = new LinkedList();
        for (int i = 0; i < listlength; i++) {
            links.add(new Integer(f.readInt()));
        }
    }

    public void fileWrite (ExtRandomAccessFile f) throws IOException {
        f.writeString(name);
        features.fileWrite(f);
        nofeatures.fileWrite(f);
        Iterator runner = links.iterator();
        f.writeByte(links.size());
        for(int i = 0; i < links.size(); i++) {
            f.writeInt(((Integer) runner.next()).intValue());
        }
    }
}
```

Lexicon.java

```java
package yall;

import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.LinkedList;
import java.util.Iterator;

/**
 * This class handles YALL’s lexicon, including the generation of the
 * lexicon from a textual lexicon specification file
 * <code>yalllext.txt</code>. To do the generation type<p><code>java
 * Lexicon</code>
 */
class Lexicon {
    public BaseTrie lemmatrie, stemtrie;
    private FileReader textfile;
    private long textpos = 0;

    private static String lemmastring = "l";
    private static String stemstring = "s";
    private static String textstring = "t.txt";

    private char lookahead;

    public Lexicon() {
        // only to create new Lexicon; needs to be followed by
        // create()
        lemmatrie = new BaseTrie();
        stemtrie = new BaseTrie();
    }

    public Lexicon(final String basename) {

        lemmatrie = new BaseTrie(basename + lemmastring);
        stemtrie = new BaseTrie(basename + stemstring);
    }

    public static void main(final String [] args) {
        // the main procedure for the application to convert the lexicon
        Lexicon thelexicon = new Lexicon();
        thelexicon.create(Yall.lexiconBaseName);
    }
```

```
public void create(final String basename) {
    try {
        textfile = new FileReader(basename + "t.txt");
    }
    catch(FileNotFoundException e) {
        throw new Error("The lexicon text file " + basename +
                        textstring + " does not exist yet. " +
                        "Please create it first.");
    }
    lemmatrie.create(basename+lemmastring);
    stemtrie.create(basename+stemstring);

    parse();
}

private void parse() {
    look();
    lemmaentries();
}

private void look() {
    while(Character.isWhitespace(lookahead = reallyRead())) {
    }
}

private char reallyRead() {
    textpos++;
    try {
        return (char) textfile.read();
    }
    catch(IOException e) {
        throw new Error("An IO error occured while parsing the " +
                        "lexicon text file.");
    }
}

private void match(char character) {
    if(lookahead != character) {
        parseError();
    }
    look();
}

private void parseError() {
    throw new Error("The lexicon text file is incorrect at " +
                    "character " + textpos);
```

```
}

private void lemmaentries() {
    while(lookahead == '[') {
        lemmaentry();
    }
    match((char) -1);
}

private void lemmaentry() {
    LexEntry newEntry = new LexEntry();
    match('[');
    newEntry.name = string();
    match(',');
    newEntry.features = features();
    match(',');
    newEntry.nofeatures = featuremaplist();
    match(',');
    LinkedList stementries = stemEntries();

    newEntry.links = new LinkedList();
    for(int i = 1; i <= stementries.size(); i++) {
        // "make room" for later update;
        // required so the size of the lemma
        // entry doesn't change when updated
        newEntry.links.add(new Integer(0));
    }
    // also changes newEntry.entrypos
    lemmatrie.insert(newEntry);
    // clear list before storing real values
    newEntry.links.clear();
    Iterator stemiterator = stementries.iterator();
    while(stemiterator.hasNext()) {
        LexEntry currentstem = (LexEntry) stemiterator.next();
        currentstem.links = new LinkedList();
        currentstem.links.add(newEntry.entrypos);
        stemtrie.insert(currentstem);
        newEntry.links.add(currentstem.entrypos);
    }
    lemmatrie.updateEntry(newEntry);

    match(']');
}

private String string() {
    StringBuffer returnstring = new StringBuffer();
```

```
        while(Character.isLetterOrDigit(lookahead) || lookahead == '_') {
            returnstring.append(lookahead);
            look();
        }
        return new String(returnstring);
    }

    private FeatureMap features() {
        FeatureMap newmap = new FeatureMap();
        match('[');
        while(lookahead != ']') {
            newmap.put(this.feature());
            if(lookahead == ',') {
                //I'm a little too generous about parsing invalid
                //grammar, but it's ok
                match(',');
            }
        }
        match(']');
        return newmap;
    }

    private Feature feature() {
        String valuestring = string();
        return new Feature(valuestring);
    }

    private FeatureMapList featuremaplist() {
        FeatureMapList newlist = new FeatureMapList();
        match('[');
        while(lookahead == '[') {
            newlist.add(features());
        }
        match(']');
        return newlist;
    }

    private LinkedList stemEntries() {
        LinkedList returnlist = new LinkedList();
        match('[');
        while(lookahead == '[') {
            LexEntry newentry = stemEntry();
            returnlist.add(newentry);
        }
        match(']');
        return returnlist;
    }

    private LexEntry stemEntry() {
        LexEntry newEntry = new LexEntry();
        match(’[’);
        newEntry.name = string();
        match(’,’);
        newEntry.features = features();
        match(’,’);
        newEntry.nofeatures = featuremaplist();
        match(’]’);
        return newEntry;
    }

    /* this is not needed *yet*
       void open() {
       lemmatrie.open();
       stemtrie.open();
       }
    */
}
```

NoMatch.java

```
package yall;

/**
 * An exception to be thrown when at some point no match is possible
 */
class NoMatch extends Exception {
}
```

NotUnifiable.java

```
package yall;

/**
 * An exception to be thrown when two feature structures cannot be
 * unified.
 */
class NotUnifiable extends Exception {
}
```

SimpleFeatureMap.java

```
package yall;

import java.util.HashMap;
import java.util.Set;
import java.util.Iterator;
import java.util.Map;

/**
 * A class to implement feature structures with features containing
 * non-disjunctive feature values.
 */
class SimpleFeatureMap extends HashMap {

    public SimpleFeatureMap() {
        super();
    }

    public SimpleFeatureMap(final SimpleFeatureMap f) {
        super(f);
    }

    boolean containsCategory (final Category category) {
        return containsKey(category);
    }

    boolean containsFeature (final Feature f) {
        return (containsKey(f.category) && get(f.category) == f.value);
    }

    Value getValue(final Category category) {
        return (Value) get(category);
    }

    public Value put(Feature f) {
        // Returns previous value associated with specified key, or
        // null if there was no mapping for key. A null return can
        // also indicate that the HashMap previously associated null
        // with the specified key.

        return (Value) put(f.category,f.value);
    }

    // public Object put(Category category, Value value)
    // already available from superclass
```

```
public boolean matchnegmap(final FeatureMap negmap) {

    // here we have to make an implementation decision: when there
    // is a conflict between positive and negative set, we can
    // either cut up the resulting sets into "basic" feature
    // structures with at most one value per category, or we
    // accept the possibility of duplicate analyses represented in
    // more than one of the resulting feature sets; I find the
    // second choice better, otherwise we would get so many
    // analysis feature sets sometimes; that wouldn't be too
    // "insightful" (see explanation of cutting up feature
    // structures into basic ones on p. 15 of my Studienarbeit);
    // well, no I'll change my mind this time: it is so messy to
    // have several clusters that also may contain duplicates and
    // that get further fragmented and duplicated by
    // matchnegmaplist; so I will use "basic" feature sets for the
    // results; this also makes the implementation of matchmap
    // *much* easier! And we can always reverse the splitting up
    // with a procedure that clusters the feature sets back
    // together.
    //
    // but before this point, "clustered" feature sets make sense,
    // because they're a lot more efficient for the program and
    // are not a big additional burden for the programmer
    //
    // well, but now that I think about it, programming might be a
    // little easier with "basic" feature maps throughout


    // this procedure is completely redone, using the insights
    // from the Studienarbeit and above

    //      LinkedList result = splitmap();
    //      LinkedList realresult = new LinkedList();
    Set negset = negmap.entrySet();

    //      Iterator positivemapsrunner = result.iterator();
    //      while(positivemapsrunner.hasNext()) {
    //          SimpleFeatureMap currentmap =
    //              (SimpleFeatureMap) positivemapsrunner.next();
    Iterator negativemaprunner = negset.iterator();

    // a nested class declaration
    //      class CurrentMapPassed extends Exception {};
    //      try {
    while(negativemaprunner.hasNext()) {
        Map.Entry currentnegpair =
            (Map.Entry) negativemaprunner.next();
        Category currentnegcategory =
            (Category) currentnegpair.getKey();
        if(! containsCategory(currentnegcategory) ||
           ! (((ValueSet) currentnegpair.getValue()).
              contains(get(currentnegcategory)))) {
            return true;
        }
        //                  throw new CurrentMapPassed();
    }
    //          }
    //          catch(CurrentMapPassed e) {
    //              realresult.add(currentmap);
    //          }
    //      }

    //      if(realresult.size() == 0)
    //          throw new NotUnifiable();

    //      return realresult;

    return false;
}


//     public boolean matchnegmaplist (final LinkedList negmaplist) {
//      Iterator negmaplistrunner = negmaplist.iterator();
//      while(negmaplistrunner.hasNext()) {
//          if(!matchnegmap((FeatureMap)negmaplistrunner.next()))
//              return false;
//      }
//      return true;
//     }

public String toString() {
    StringBuffer text = new StringBuffer("[\n");
    Set mapset = entrySet();
    Iterator setiterator = mapset.iterator();
    while (setiterator.hasNext()) {
        Map.Entry entry = (Map.Entry) setiterator.next();
        byte catindex = ((Category) entry.getKey()).index();
        Value value = (Value) entry.getValue();
        text.append(FeatureNames.categorynames[catindex - 1] + " : " +
                        value.toString(catindex) + "\n");
        }
        text.append("]\n");
        return new String(text);
    }
}
```

StringFeatureMap.java

```
package yall;

import java.util.LinkedList;
import java.util.Iterator;

/**
 * Combines an analysis from the endings trie with its remaining stem.
 */
class StringFeatureMap {
    String stem;
    FeatureMap analysis;

    public LinkedList lexiconMatch(Lexicon matchlexicon) throws NoMatch {
        LinkedList resultlist = new LinkedList();
        LinkedList stemmatches =
            matchlexicon.stemtrie.findEntries(stem,(byte)0);
        //         LinkedList lemmamatches = new LinkedList();
        Iterator stemmatchiter = stemmatches.iterator();
        while(stemmatchiter.hasNext()) {
            LexEntry currentstemmatch = (LexEntry) stemmatchiter.next();
            //      lemmamatches.add(new ExtLexEntry(currentstemmatch,
            //      matchlexicon.lemmatrie));
            ExtLexEntry currentlemmamatch =
                new ExtLexEntry(currentstemmatch,
                                matchlexicon.lemmatrie.entryfile);
            //  for(Iterator lemmamatchiter = lemmamatches.iterator(),
            //          ExtLexEntry currentlemmamatch;
            //      lemmamatchiter.hasNext();
            //      currentlemmamatch =
            //          (ExtLexEntry) lemmamatchiter.next()) {

            FeatureMap unified = null;
            //assignment just for stupid Java compiler...

            try{
                unified = analysis.matchmap(currentlemmamatch.features);
            }
            catch (NotUnifiable e) {
                continue;
            }
            LinkedList simplemaps = unified.splitmap();
            Iterator smapsiter = simplemaps.iterator();
            while(smapsiter.hasNext()) {
                SimpleFeatureMap currentsmap =
                    (SimpleFeatureMap) smapsiter.next();
                class NotPassingNegMap extends Exception {};
                try {
                    Iterator negmapiter =
                        currentlemmamatch.nofeatures.iterator();
                    while(negmapiter.hasNext()) {
                        FeatureMap currentnegmap =
                            (FeatureMap) negmapiter.next();
                        if(!currentsmap.matchnegmap(currentnegmap)) {
                            throw new NotPassingNegMap();
                        }

                    }
                    AnalysisResult validanalysis = new AnalysisResult();
                    validanalysis.lemmaname =
                        currentlemmamatch.lemmagrapheme;
                    validanalysis.stemname = currentlemmamatch.name;
                    validanalysis.features = currentsmap;
                    resultlist.add(validanalysis);
                }
                catch(NotPassingNegMap e) {
                    // do nothing
                }
            }
        }
        if(resultlist.size() != 0) {
            return resultlist;
        }
        else {
            throw new NoMatch();
        }
    }
}
```

TempEntry.java

```java
package yall;

import java.util.LinkedList;

/** Objects of this class are temporary and needed during the build-up
 * of the endings trie when the application is started.
 */
class TempEntry {
    String identifier;
    ExtAffix node;
    LinkedList nextentries;

    /*
      public TempEntry() {
      }
    */

    /*
      public TempEntry(String id, ExtAffix nd, LinkedList ne) {
      this();
      identifier = id;
      node = nd;
      nextentries = ne;
      }
    */

}
```

Value.java

```
package yall;

/**
   Used to specify feature values.
 */
final class Value {
    private byte index;

    private static Value [] object = {new Value((byte)0),
                                      new Value((byte)1),
                                      new Value((byte)2),
                                      new Value((byte)3),
                                      new Value((byte)4),
                                      new Value((byte)5),
                                      new Value((byte)6),
                                      new Value((byte)7),
                                      new Value((byte)8)};

    private Value (byte index) {
        this.index = index;
    }

    public byte index () {
        return index;
    }

    public static Value object (byte index) {
        return object[index];
    }

    public String toString() {
        return String.valueOf(index);
    }

    public String toString(byte catindex) {
        return FeatureNames.valuenames[catindex - 1][index - 1];
    }
}
```

ValueSet.java

```java
package yall;

import java.util.HashSet;
import java.util.Iterator;

/**
   Specifies disjunctions of feature values.
 */
class ValueSet extends HashSet {

    public ValueSet() {
        super();
    }

    ValueSet match(final ValueSet othervalueset) throws NotUnifiable {
        ValueSet newvalueset = new ValueSet();

        Iterator thisrunner = iterator();

    thisloop:
        while(thisrunner.hasNext()) {
            Value thisvalue = (Value) thisrunner.next();
            if(othervalueset.contains(thisvalue)) {
                newvalueset.add(thisvalue);
            }

        }
        if(newvalueset.size() != 0) {
            return newvalueset;
        }
        else {
            throw new NotUnifiable();
        }
    }

    public String toString() {
        StringBuffer text = new StringBuffer("{");
        Iterator runner = iterator();
        while(runner.hasNext()) {
            text.append(((Value) runner.next()).toString());
        }
        text.append("}");
        return new String(text);
    }
    public String toString(byte catindex) {
        StringBuffer text = new StringBuffer("{ ");
        Iterator runner = iterator();
        while(runner.hasNext()) {
            Value val = (Value) runner.next();
            text.append(val.toString(catindex) + " ");
        }
        text.append("}");
        return new String(text);
    }
}
```

Yall.java

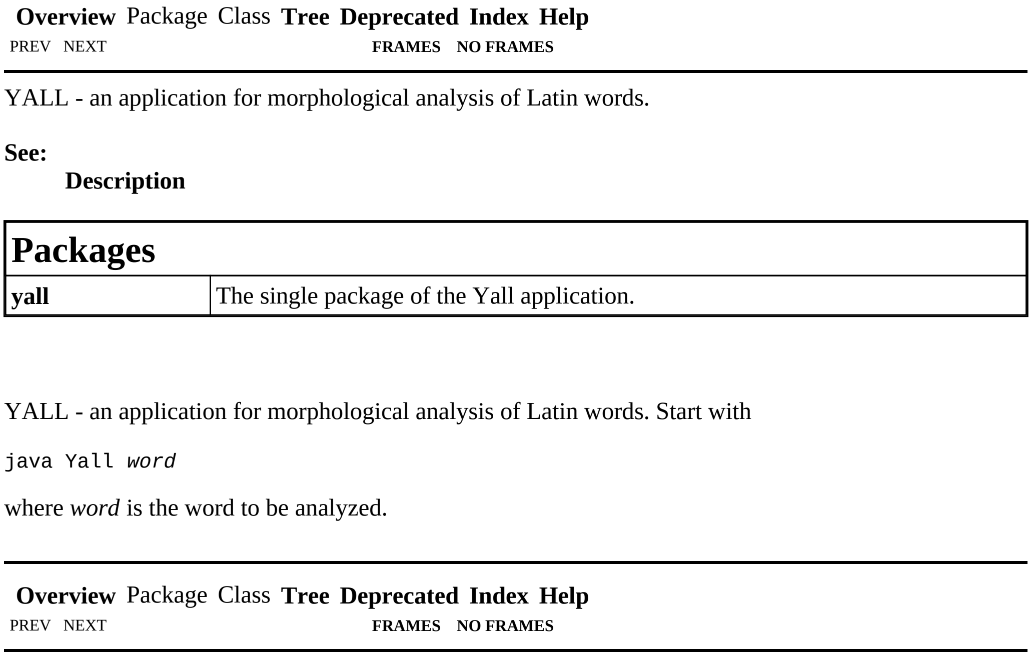
```
package yall;

import java.util.LinkedList;
import java.util.Iterator;

/**
 * The main class. Contains "global variables", initialization
 * methods, and the main method to perform a morphological word
 * analysis.
 */
class Yall {

    /**
     * The endings trie.
     */
    Automat latinendings;
    /**
     * The lexicon.
     */
    Lexicon yalllexicon;

    /**
     * The prefix of the file names for the lexicon files.
     */
    public static String lexiconBaseName = "yalllex";

    /**
     * Builds up the endings trie and initializes the lexicon database.
     */
    public Yall() {
        latinendings = new Automat();
        yalllexicon = new Lexicon(lexiconBaseName);
    }

    /**
     * Performs a morphological analysis.
     *
     * @param word the word to be analyzed
     * @return A {@link java.util.LinkedList} of
     * {@link StringFeatureMap} objects, specifying the results.
     * @throws NoMatch if no valid analysis can be found
     */
    public LinkedList analyze(final String word) throws NoMatch {
        LinkedList endingsanalyses = latinendings.analyze(word);
        LinkedList completeanalyses = new LinkedList();
        Iterator endingsrunner = endingsanalyses.iterator();
        while(endingsrunner.hasNext()) {
            try {
                completeanalyses.
                    addAll(((StringFeatureMap)endingsrunner.next()).
                           lexiconMatch(yalllexicon));
            }
            catch(NoMatch e) {
                //one mismatch is ok
            }
        }
        if(completeanalyses.size() == 0) {
            throw(new NoMatch());
            // only mismatches means complete failure
        }
        return completeanalyses;
    }

    /**
     * <code>main</code> method of the YALL application. Prints out
     * the found analysis to the console window. For instructions how
     * to use YALL see the <a
     * href="../overview-summary.html#overview_description">overview
     * description</a>.
     *
     * @param args a single Latin word to be analyzed
     */
    public static void main(final String [] args) {
        Yall yallinstance = new Yall();
        try {
            LinkedList analysis = yallinstance.analyze(args[0]);
            System.out.println("Found " + analysis.size() +
                               " analyses for " + args[0] + ".\n\n");
            Iterator runner = analysis.iterator();
            while(runner.hasNext()) {
                System.out.println(((AnalysisResult) runner.next()).
                                   toString() + "\n");
            }
        }
        catch(NoMatch e) {
            System.out.println("No valid analysis for " + args[0] +
                               " could be found.");
        }
    }
}
```

Appendix G

YALL Class Documentation

Below the class interface documentation files for YALL are shown, which were generated automatically from the class files with the `javadoc` command. The documentation files are originally in `HTML` hypertext format, hence this appendix can only give an approximation.

G.1 Overview



Overview Package Class **Tree** **Deprecated** **Index** **Help**
PREV NEXT **FRAMES** **NO FRAMES**

YALL - an application for morphological analysis of Latin words.

See:
Description

Packages	
yall	The single package of the Yall application.

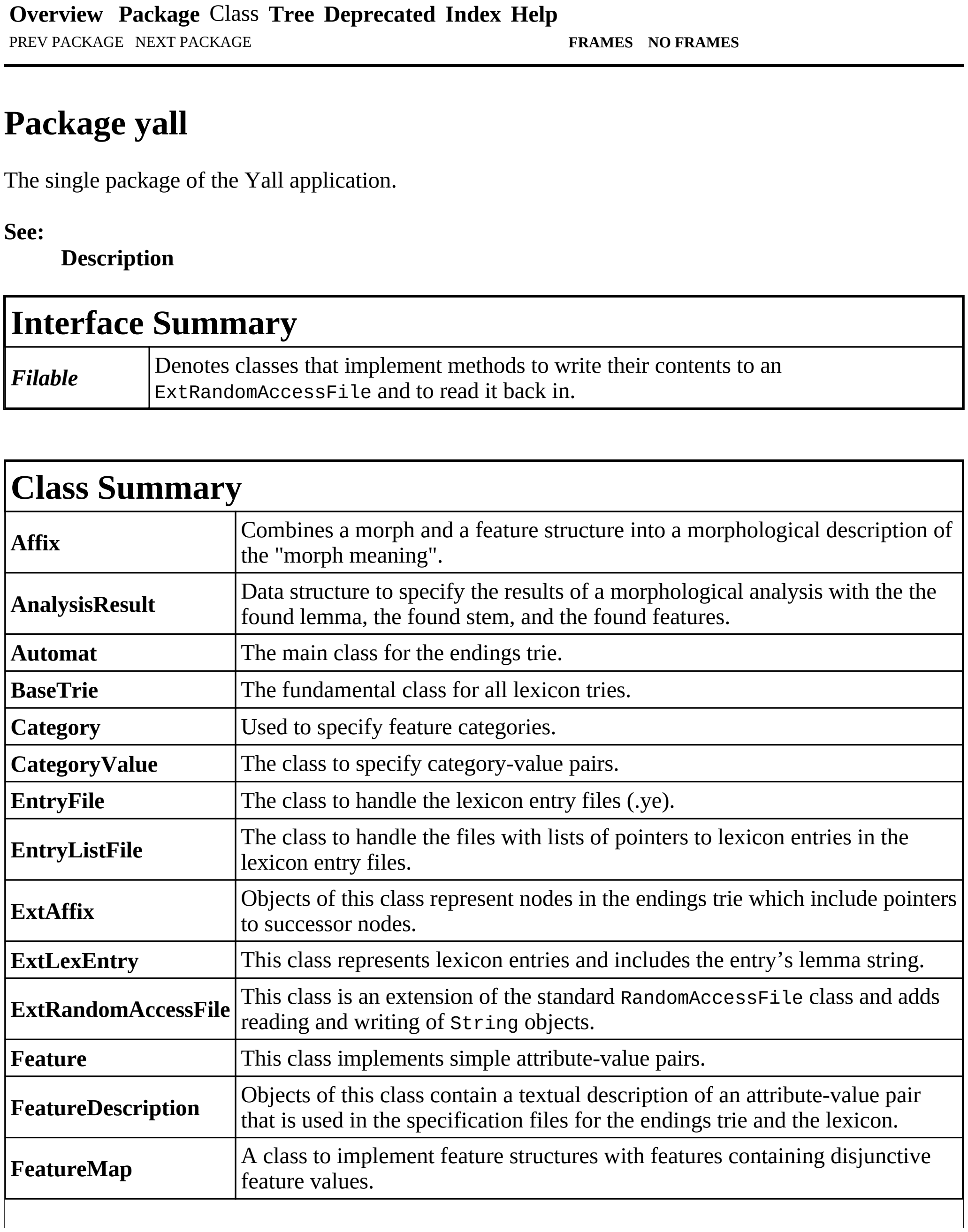
YALL - an application for morphological analysis of Latin words. Start with

`java Yall` *`word`*

where *word* is the word to be analyzed.

Overview Package Class **Tree** **Deprecated** **Index** **Help**
PREV NEXT **FRAMES** **NO FRAMES**

G.2 Package yall



Overview **Package** Class **Tree** **Deprecated** **Index** **Help**
PREV PACKAGE NEXT PACKAGE **FRAMES** **NO FRAMES**

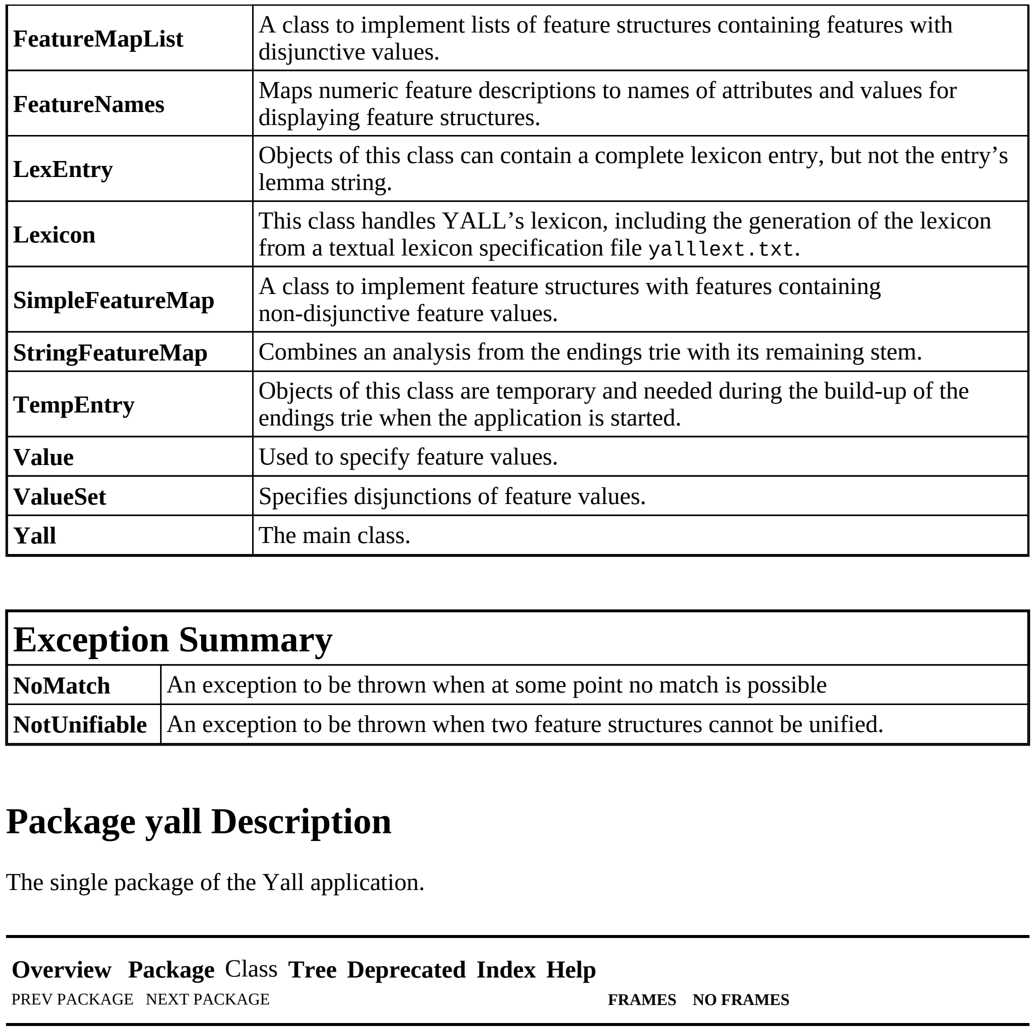
Package yall

The single package of the Yall application.

See:
Description

Interface Summary	
Filable	Denotes classes that implement methods to write their contents to an `ExtRandomAccessFile` and to read it back in.

Class Summary	
Affix	Combines a morph and a feature structure into a morphological description of the "morph meaning".
AnalysisResult	Data structure to specify the results of a morphological analysis with the the found lemma, the found stem, and the found features.
Automat	The main class for the endings trie.
BaseTrie	The fundamental class for all lexicon tries.
Category	Used to specify feature categories.
CategoryValue	The class to specify category-value pairs.
EntryFile	The class to handle the lexicon entry files (.ye).
EntryListFile	The class to handle the files with lists of pointers to lexicon entries in the lexicon entry files.
ExtAffix	Objects of this class represent nodes in the endings trie which include pointers to successor nodes.
ExtLexEntry	This class represents lexicon entries and includes the entry’s lemma string.
ExtRandomAccessFile	This class is an extension of the standard `RandomAccessFile` class and adds reading and writing of `String` objects.
Feature	This class implements simple attribute-value pairs.
FeatureDescription	Objects of this class contain a textual description of an attribute-value pair that is used in the specification files for the endings trie and the lexicon.
FeatureMap	A class to implement feature structures with features containing disjunctive feature values.



FeatureMapList	A class to implement lists of feature structures containing features with disjunctive values.
FeatureNames	Maps numeric feature descriptions to names of attributes and values for displaying feature structures.
LexEntry	Objects of this class can contain a complete lexicon entry, but not the entry's lemma string.
Lexicon	This class handles YALL's lexicon, including the generation of the lexicon from a textual lexicon specification file `yalllext.txt`.
SimpleFeatureMap	A class to implement feature structures with features containing non-disjunctive feature values.
StringFeatureMap	Combines an analysis from the endings trie with its remaining stem.
TempEntry	Objects of this class are temporary and needed during the build-up of the endings trie when the application is started.
Value	Used to specify feature values.
ValueSet	Specifies disjunctions of feature values.
Yall	The main class.

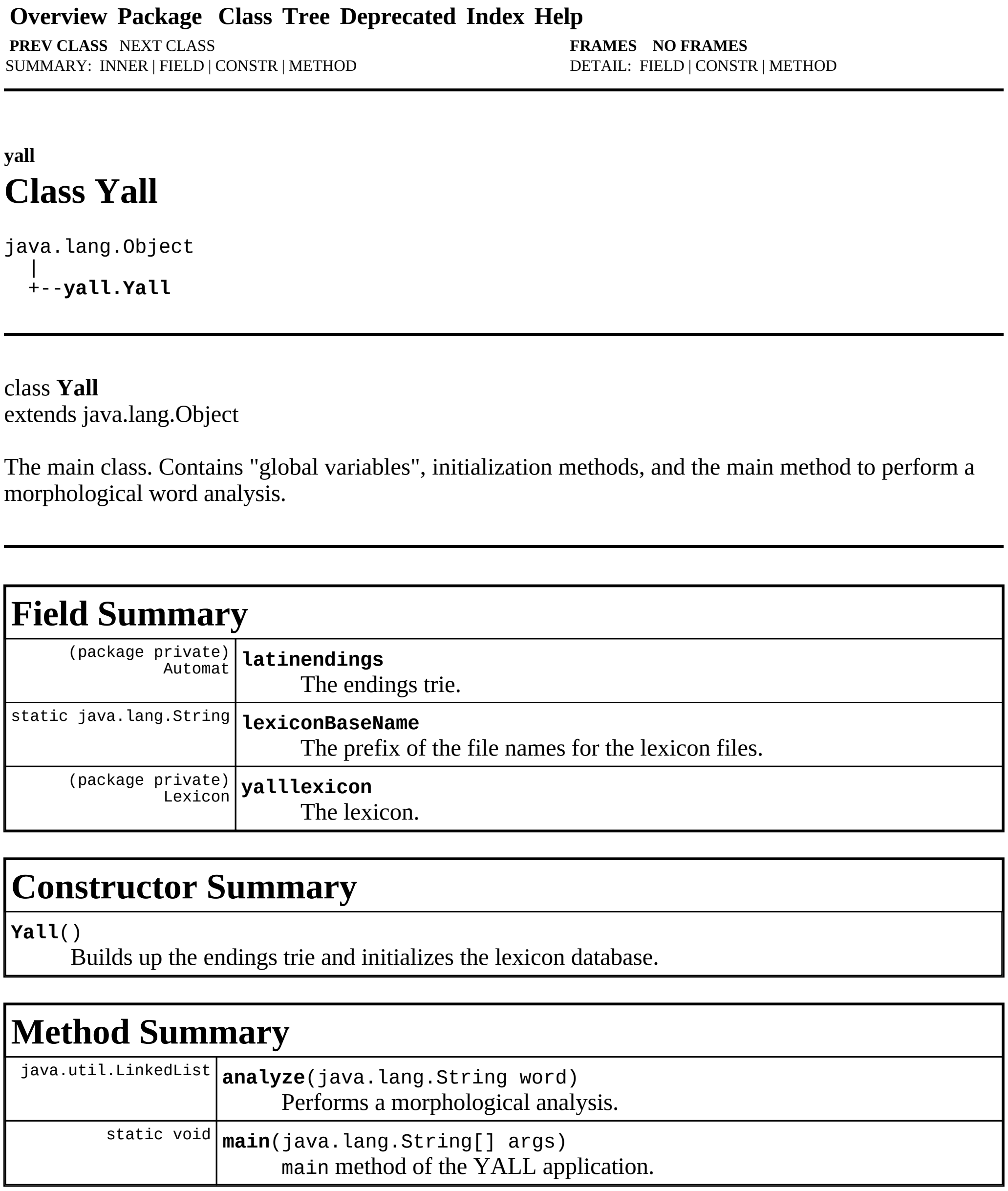
Exception Summary	
NoMatch	An exception to be thrown when at some point no match is possible
NotUnifiable	An exception to be thrown when two feature structures cannot be unified.

Package yall Description

The single package of the Yall application.

Overview **Package** Class **Tree** **Deprecated** **Index** **Help**
PREV PACKAGE NEXT PACKAGE **FRAMES** **NO FRAMES**

G.3 Class Yall



Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES**

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

yall

Class Yall

```
java.lang.Object
  |
  +--yall.Yall
```

class **Yall**
extends java.lang.Object

The main class. Contains "global variables", initialization methods, and the main method to perform a morphological word analysis.

Field Summary	
(package private) Automat	**latinendings** The endings trie.
static java.lang.String	**lexiconBaseName** The prefix of the file names for the lexicon files.
(package private) Lexicon	**yalllexicon** The lexicon.

Constructor Summary
Yall() Builds up the endings trie and initializes the lexicon database.

Method Summary	
java.util.LinkedList	**analyze**(java.lang.String word) Performs a morphological analysis.
static void	**main**(java.lang.String[] args) main method of the YALL application.

Methods inherited from class java.lang.Object
`, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait`

Field Detail

latinendings

```
Automat latinendings
```

The endings trie.

yalllexicon

```
Lexicon yalllexicon
```

The lexicon.

lexiconBaseName

```
public static java.lang.String lexiconBaseName
```

The prefix of the file names for the lexicon files.

Constructor Detail

Yall

```
public Yall()
```

Builds up the endings trie and initializes the lexicon database.

Method Detail

analyze

```
public java.util.LinkedList analyze(java.lang.String word)
                             throws NoMatch
```

Performs a morphological analysis.
Parameters:

`word` - the word to be analyzed

Returns:

A `LinkedList` of `StringFeatureMap` objects, specifying the results.

Throws:

NoMatch - if no valid analysis can be found

---

main

```
public static void main(java.lang.String[] args)
```

`main` method of the YALL application. Prints out the found analysis to the console window. For instructions how to use YALL see the overview description.

Parameters:

`args` - a single Latin word to be analyzed

Overview Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES** **NO FRAMES**

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD