Java-basierter Lexicon Editor

Dominik Kaspar Matthias Wille

WS 2002/2003

Inhaltsverzeichnis

1 Einleitung – Der Lexicon Editor **1**

1.1 Ziele, Designziele 1

1.2 Ausgangslage 1

1.3 Vorgehensweise 2

2 Einführung in SVOX **3**

2.1 Teile der Sprachsynthese 3

2.1.1 Transkriptionsstufe von SVOX 4

2.1.2 Phono-akustische Stufe 6

2.2 Lexika in SVOX 7

2.2.1 Masterlexika 8

3 Die Benutzeroberfläche (GUI) **11**

3.1 Tabelle 11

3.2 History 13

3.3 Flektionen 13

3.4 Toolbar 14

3.5 Menüs 14

3.6 Statusleiste 16

3.7 Login-Fenster 16

4 Editierhilfen im Überblick **17**

4.1 Such-Filter 17

4.2 Hervorhebung ungültiger Werte 17

4.3 Auto-Completion 18

4.4 Automatische phonetische Transkription 18

4.5 Anzeige möglicher Featurewerte und deren Bedeutung 18

4.6 Generierung der Flektionen 19

5 Externe Beschreibungsdateien **21**

5.1 Die Datei *master_description.txt* 21

5.1.1 Allgemeines Format 21

5.1.2 Feature-Definitionen 22

5.1.3 Kategorie-Definitionen 23

5.2 Die Datei *preferences.txt* 24

5.2.1 Allgemeines Format 24

5.2.2 Bedeutung der einzelnen Einstellungen 25

6 Die Flektionsgenerierung **29**

6.1 Kurzer Überblick 29

6.2 Integration des Flektionstools 29

6.3 Ablauf einer Flektion 30

6.4 Formatierte Anzeige der Flektionen 32

7 Usability **35**

7.1 Limitationen des Java Lexicon Editors 35

7.2 Usability-Tests 36

7.3 Bekannte Bugs 37

8 Fazit **39**

8.1 Aufgetretene Probleme 39

8.2 Schlussfolgerungen 39

8.3 Verbesserungsmöglichkeiten und weitere denkbare Funktionen 40

A Aufgabenstellung **43**

B Klassendokumentation **47**

B.1 Paket ethz.svoxlexedit 48

B.2 Abstrakte Klasse AbstractLexTableModel 50

B.3 Klasse Category 52

B.4 Klasse ConsElement 54

B.5 Klasse Feature 55

B.6 Klasse FeatureValue 57

B.7 Klasse HistoryEntry 58
B.8 Klasse HistoryTableModel 60
B.9 Klasse LexFilterField 62
B.10 Klasse LexFormat 64
B.11 Klasse LexHTMLViewer 66
B.12 Klasse Lexicon 68
B.13 Klasse LexiconEditor 70
B.14 Klasse LexiconEntry 71
B.15 Klasse LexTable 73
B.16 Klasse LexTableCellEditor 75
B.17 Klasse LexTableCellRenderer 76
B.18 Klasse LexTableFilter 77
B.19 Klasse LexTableMap 79
B.20 Klasse LexTableModel 81
B.21 Klasse LexTableSorter 83
B.22 Klasse LexTool 85
B.23 Klasse LoginWindow 87
B.24 Klasse MainWindow 88
B.25 Klasse Preferences 90
B.26 Klasse RE 92

C Description-Datei **99**

D Preference-Datei **103**

E CD-ROM **105**

Kapitel 1

Einleitung – Der Lexicon Editor

1.1 Ziele, Designziele

Das Ziel dieser Semesterarbeit ist die Implementierung eines Java-Programmes zur benutzerfreundlichen Editierung von Lexika des SVOX-Systems.

Der Java Lexicon Editor soll neue Lexika anlegen können und die Manipulation bereits bestehender Lexika erlauben. Ausserdem muss es möglich sein, mit dem Java Lexicon Editor syntaktische aber auch semantische Fehler in bestehenden Lexika auf effiziente Weise zu lokalisieren und zu beheben.

Alle zur Verfügung stehenden Funktionen sollen über ein graphisches Benutzer-Interface ausgewählt und ausgeführt werden können. Eine primäre Aufgabe wird also sein, eine ansprechende und einfach bedienbare Oberfläche zu gestalten.

Schliesslich soll es dem Java Lexicon Editor möglich sein, mit bereits existierenden Tools zu kommunizieren. Eine Zusammenarbeit mit dem Programm *lexedit* ermöglicht es beispielsweise, Verben zu konjugieren oder die phonetische Schreibweise eines Wortes zu bestimmen.

1.2 Ausgangslage

Da die bis anhin in elektronischer Form verfügbaren Lexika nicht die für das Sprachsynthese-System SVOX notwendige grammatikalische Information und zum Teil nicht einmal die phonetische Transkription enthalten, ist es unausweichlich, die benötigten Lexika in aufwändiger Kleinarbeit von Hand zu erstellen. Das bereits existierende Tool für die Lexioneditierung kann lediglich über eingetippte Kommandos bedient werden, was die Erschaffung grosser Lexikon-Datenbanken sehr ineffizient macht. Auch war es bisher umständlich, Lexika auf Fehlerstellen zu untersuchen oder auch nur übersichtlich darzustellen.

In dieser Semesterarbeit gilt es nun, die angesprochenen Nachteile zu beseitigen.

1.3 Vorgehensweise

Unser Programmierprojekt haben wir grob in folgende Schritte aufgeteilt:

1. Einarbeitung in die Themen der Sprachverarbeitung und das SVOX Sprachsynthese-System mittles Literatur.
2. Zusammenstellen eines Anforderungsprofils für den Editor zusammen mit unserem Betreuer.
3. Design der graphischen Oberfläche und der grundlegenden Programmstruktur. Anstatt auf vorgefertigte Komponenten zurückzugreifen, haben wir uns entschieden, das Programm von Grund auf selbst zu implementieren.
4. Implementationsphase. Schreiben der Java-Klassen, Debugging und Testen der Funktionstüchtigkeit.
5. Verfassen dieses Berichts.

Kapitel 2

Einführung in SVOX

Da unsere Arbeit Teil des Sprachsynthesesystems SVOX sein wird, soll dieses Kapitel einen kurzen Überblick über die Sprachsynthese im Allgemeinen und SVOX im Speziellen enthalten. Auch wird kurz auf die Verwendung von Lexika in SVOX eingegangen und das Masterlexikon-Format wird vorgestellt, denn dieses Format soll unser Lexicon Editor schlussendlich verarbeiten können.

Da die Sprachsynthese ein höchst kompliziertes und anspruchsvolles Gebiet der Sprachverarbeitung darstellt und eine vollständige Diskussion ganze Bücher füllt, kann im Rahmen dieser Arbeit nur sehr oberflächlich auf das Thema eingegangen werden. Für weitere Informationen wird auf die beiden Skripts der Vorlesungen *Sprachverarbeitung I & II* [1][2] und die Dissertation von C. Traber [3] verwiesen.

SVOX ist ein an der ETH Zürich entwickeltes Sprachsynthesesystem (engl. Text-To-Speech System), das nach dem Prinzip der Lautverkettung funktioniert. Das Sprachsignal wird durch Verkettung von aufgenommenen Sprachsegmenten (Lauten) eines Sprechers erzeugt.

2.1 Teile der Sprachsynthese

Unter Sprachsynthese versteht man im Allgemeinen die Umsetzung orthographischer Texte in Lautsprache, also in ein Sprachsignal. Was für uns Menschen eine Selbstverständlichkeit ist, ist für den Computer mit erheblichen Schwierigkeiten verbunden, die bis heute nicht ganz eliminiert werden konnten.

Da geschriebener Text im Gegensatz zu gesprochener Sprache gänzlich sprecherunabhängig ist, muss auch jedes Sprachsynthesesystem diesen Übergang von sprecherunabhänig zu sprecherabhänig vollziehen und entsprechende Komponenten enthalten. In Abbildung 2.1 ist ein derart aufgegliedertes Sprachsynthesesystem dargestellt. Der sprecherunabhängige Teil wird als *Transkription* bezeichnet und hat die Aufgabe, eine abstrakte Beschreibung (die *phonologische Darstellung*) des zu erzeugenden Sprachsignals zu erzeugen, die selbst noch sprecherunabhängig ist. Der sprecherabhängige Teil wird *phonoakustische Stufe* genannt und hat die Aufgabe, aus der phonologischen Darstellung das konkrete Sprachsignal zu erzeugen.

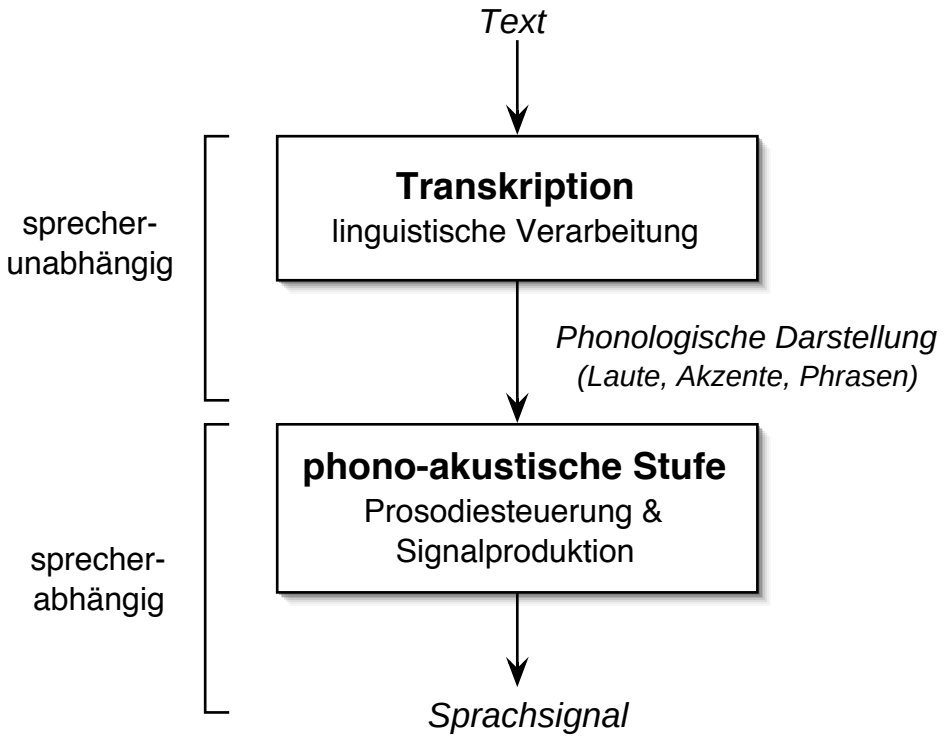


Abbildung 2.1: Der sprecherabhängige und sprecherunabhängige Teil eines Sprachsynthesesystems

2.1.1 Transkriptionsstufe von SVOX

Die Transkriptionsstufe ermittelt die Aussprache der Wörter, legt also fest, welche Laute nacheinander erzeugt werden müssen, welche Sprechsilben wie stark akzentuiert sein sollen, und wo wie starke Sprechgruppengrenzen zu setzen sind.

Die Transkriptionsstufe des SVOX-Systems ist schematisch in Abbildung 2.2 zu sehen. Sie arbeitet satzweise, wobei jedes Wort zuerst morphologisch analysiert wird (Bestimmen von Wortart, Wortform und Aussprache), und anschliessend ermittelt die Syntaxanalyse aufgrund der Wortinformation und der Satzgrammatik den Satzaufbau (Syntaxbaum), aus dem dann die Akzentuierung und die Phrasierung abgeleitet werden.

Die morphologische Analyse in SVOX basiert auf einer Wortgrammatik in DCG[1]-Form und auf einem Lexikon mit Einträgen in orthographischer und phonetischer Form. Die Wortgrammatik enthält eine Menge von Regeln, die festlegen wie die verschiedenen Wortarten aufgebaut sind, z.B dass ein deutsches Verb aus einem Verbstamm und einer Verbendung besteht. Die Einträge im Lexikon sind entweder ganze Morpheme[2] (Stämme, Endungen) oder Morphe[3] (Stammvarianten). Mit Hilfe des Lexikons werden die Wörter der Eingabe geparst und zerlegt, und mit der Wortgrammatik wird versucht das Wort zu analysieren

[1] Definite Clause Grammar.
[2] Ein Morphem ist die kleinste bedeutungstragende Einheit einer Sprache.
[3] Ein Morph ist eine noch nicht weiter klassifizierte Lautfolge.

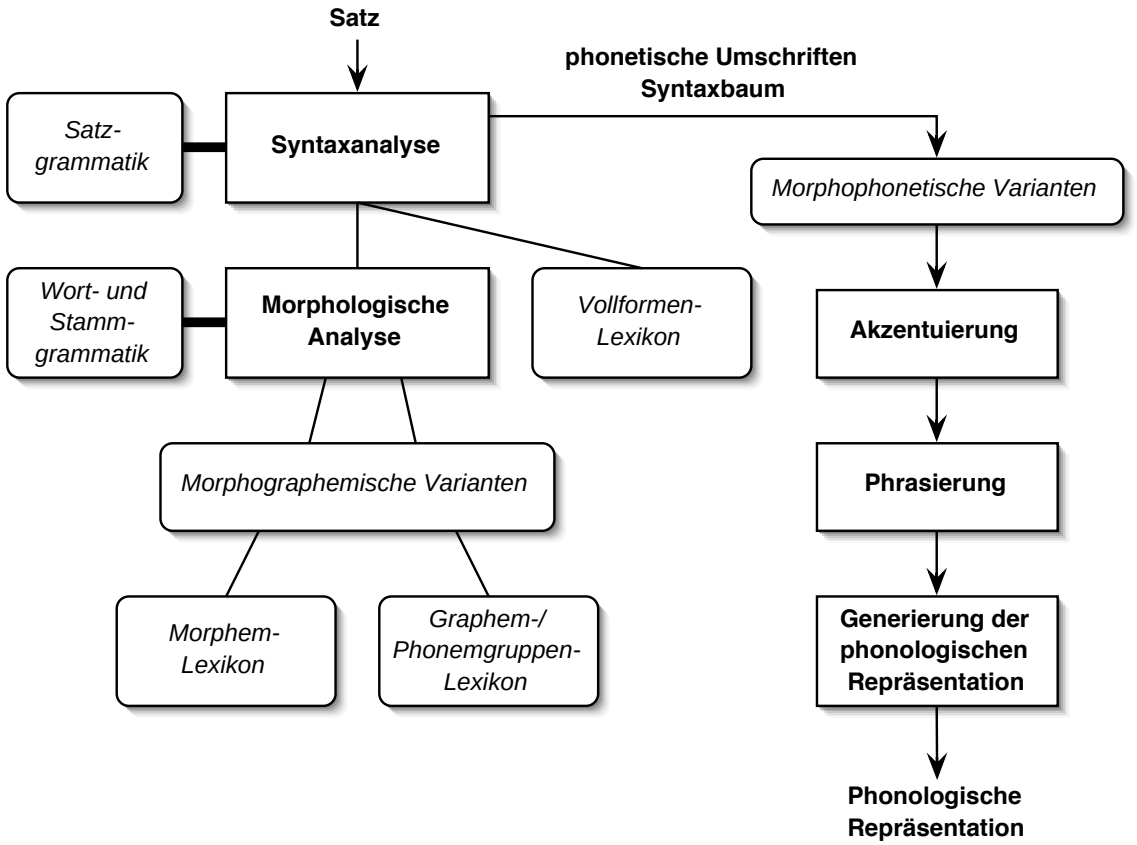


Abbildung 2.2: Die Transkriptionsstufe von SVOX

und die Wortart davon zu bestimmen. Die Aussprache jedes Morphs oder Morphems ergibt sich durch einen einfachen Verweis auf den entsprechenden Lexikoneintrag.

Die Syntaxanalyse in SVOX basiert auf den Resultaten der morphologischen Analyse für flektierbare Wörter, auf einem Vollformlexikon und einer DCG-Satzgrammatik für deutsche Sätze. Das Vollformlexikon enthält Funktionswörter (nichtzerlegbare Wörter wie Artikel, Präpositionen etc.) sowie Ausnahmen, die von der Wortanalyse nicht richtig behandelt würden. Zur Satzanalyse wird ein Bottom-up-Parsing-Verfahren verwendet, das versucht, aus den einzelnen Wörtern mit den bestimmten Wortarten einen syntaktisch korrekten Satz zu ermitteln. Konkret wird versucht, aus den Regeln der Satzgrammatik einen Syntaxbaum des Satzes aufzubauen.

Als abschliessendes Beispiel zur Transkription betrachten wir folgenden Satz:

> "Heinrich besuchte gestern die Ausstellung im Kunstmuseum."

Nach der morphologischen und syntaktischen Analyse liefert SVOX den Syntaxbaum aus Abbildung 2.3.[4]Aus dem Syntaxbaum ist ersichtlich, dass "Heinrich" als Nominalgruppe

[4]In Wirklichkeit resultieren aus obigem Satz 56 verschiedene Lösung, hauptsächlich wegen Unzulänglichkeiten der Syntaxanalyse. Durch geschicktes Gewichten der Lösungen können unsinnige Lösungen wieder ausgeschieden werden.

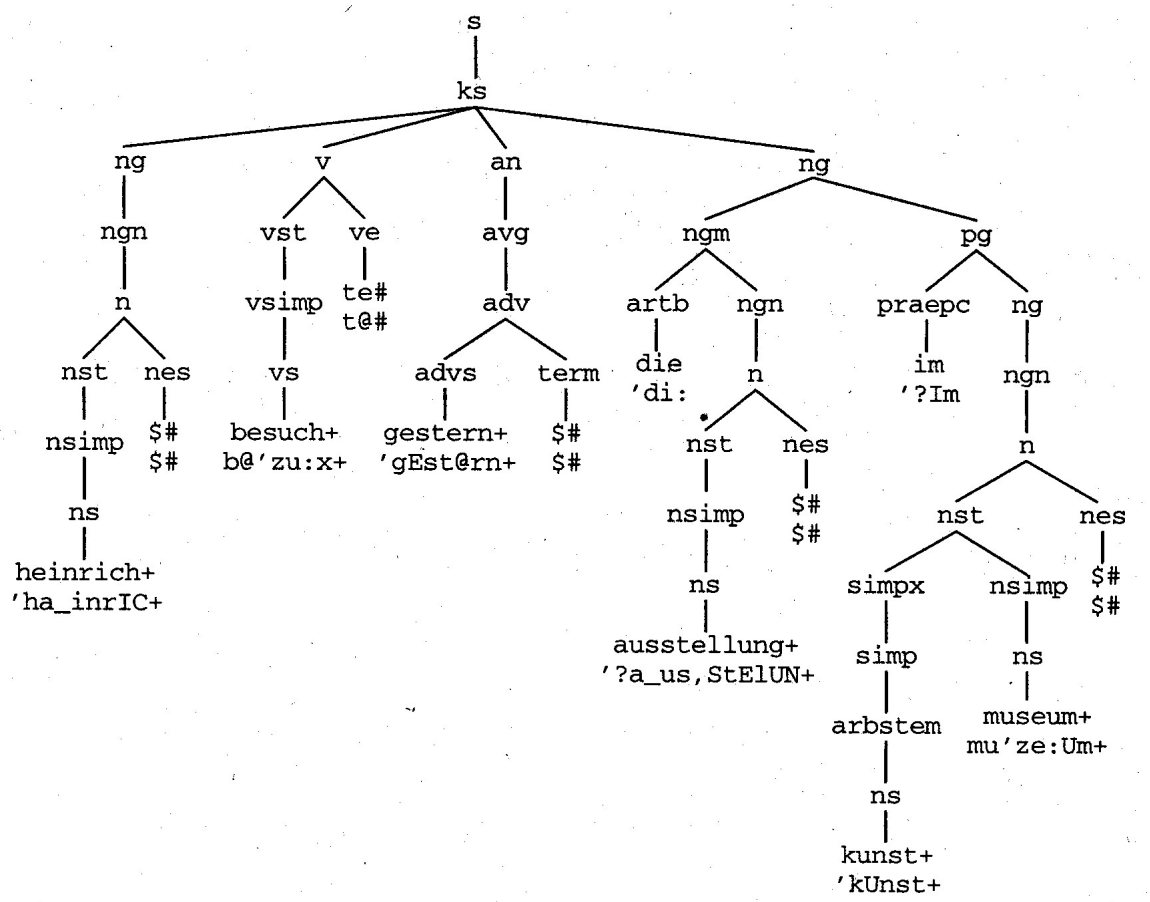


Abbildung 2.3: Die syntaktische Struktur des Satzes "Heinrich besuchte gestern die Ausstellung im Kunstmuseum"

(*ng*), "besuchte" als Vollverb (*v*), "gestern" als ergänzende Angabe (*an*) und "die Ausstellung im Kunstmuseum" wiederum als *ng* klassiert worden ist.

Als Endprodukt der Transkriptionsstufe resultiert nach Akzentuierung und Phrasierung die folgende phonologische Repräsentation:

(P) [1]haɪn-rɪç #{2} (P) bə-[2]zuːx-tə [1]gɛs-tərn #{4} (T) diː [2]|aʊs-[4]ʃtɛ-lʊŋ |ɪm [1]kʊnst-mu-[4]zeː-ʊm.

Diese enthält neben der Aussprache der Wörter in phonetischer Schrift auch Informationen zu Silbengrenzen (-), Betonungsgrad von Silben in eckigen Klammern, Phrasengrenzen in geschweiften Klammern und Phrasentypen in runden Klammern.

2.1.2 Phono-akustische Stufe

Die phono-akustische Stufe erzeugt aus der abstrakten (sprecherunabhängigen) phonologischen Darstellung des Textes das konkrete Sprachsignal. Sie umfasst zwei Teilfunktio-

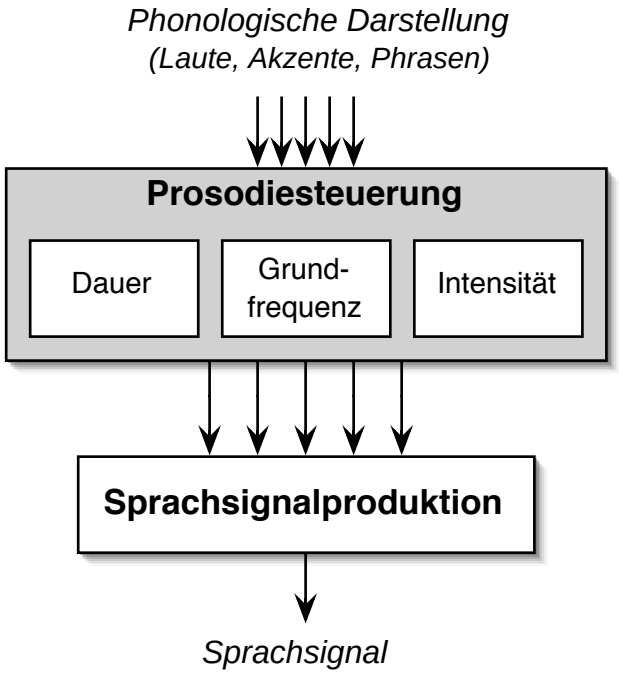


Abbildung 2.4: Die phonoakustische Stufe

nen, die *Prosodiesteuerung* und die *Sprachsignalproduktion* (Abbildung 2.4). Die Prosodiesteuerung leitet aus den Akzenten und den Phrasengrenzen die folgenden Parameter ab: den Tonhöhenverlauf, den Sprechrhythmus und den Intensitätsverlauf. Aus diesen Grössen können sodann die Tonhöhe, die Dauer und die Intensität der einzelnen Laute bestimmt werden. Die Sprachsignalproduktion kann nun die Laute mit der vorgeschriebenen Tonhöhe, Dauer und Intensität erzeugen: es resultiert das synthetische Sprachsignal.

2.2 Lexika in SVOX

Wie wir im obigen Abschnitt gesehen haben, verwendet SVOX für die Analyse von Wörtern und Sätzen verschiedene Lexika, im Einzelnen ein Vollformenlexikon für die Syntaxanalyse, ein Morphemlexikon für die morphologische Analyse und ein Graphem- bzw. Phonemgruppenlexikon für die phonetische Umschrift von nicht analysierbaren Wörtern.

Abbildung 2.5 zeigt einen kleinen Ausschnitt aus dem SVOX-Morphemlexikon. Jeder Eintrag (am Beispiel des ersten Eintrages) enthält folgende Information:

- `NS_D`: *Konstituent*. Ein deutscher Nomenstamm.
- `(SK3,PK7,M,N)`: *Features/Merkmale*. Für Nomenstämme sind die Features *Singularklasse* (`SK3`), *Pluralklasse* (`PK7`), *Geschlecht* (`M` für männlich) und *Oberflächenklasse* (`N` für Nomen) vorgesehen.
- `"sog+"`: *Graphemische Form des Eintrages*. Das `+` markiert dabei das Ende eines

Stammes.

- `"'zo:g+"`: *Phonetische Form des Eintrages.* Dafür wird eine ASCII-taugliche Version der IPA-Lautschrift verwendet (ETHPA).

Für das SVOX-System genügen diese Informationen vollständig, um die entsprechenden Analysen durchführen zu können. Für die Sprachwissenschaftler, die die Lexika erstellen und bearbeiten, wären aber zusätzliche Informationen von Nutzen. Darum gibt es in SVOX noch sogenannte *Masterlexika*, die die normalen SVOX-Lexika um zusätzliche Infos erweitern und ein etwas anderes Format haben.

```
NS_D (SK3,PK7,M,N)  "sog+"  "'zo:g+"
NS_D (SK1,PK3,F,N)  "sohle+"  "'zo:l@+"
NS_D (SK3,PK0,M,N)  "sohn+"  "'zo:n+"
NS_D (SK9,PK4,M,N)  "soldat+"  "zOl'da:t+"
NS_D (SK4,PK1,N,N)  "sommer+"  "'zOm@r+"
NS_D (SK4,PK7,N,N)  "sommerfest+"  "'zOm@rfEst+"
NS_D (SK2,PK7,M,N)  "sonnabend+"  "'zOn?a:b@nd+"
NS_D (SK1,PK3,F,N)  "sonne+"  "'zOn@+"

ADVS_D (M)  "also+"  "'?alzo+"
ADVS_D (M)  "anders+"  "'?and@rz+"
ADVS_D (L)  "anderswo+"  "'?and@rsvo:+"
ADVS_D (L)  "anderswohin+"  "'?and@rsvo,hIn+"
ADVS_D (M)  "anderthalb+"  ",?and@rt'halb+"
ADVS_D (T)  "anfangs+"  "'?anfaNs+"

DIG2_D (O,U)  "40"  ",fi:r-t_sICst+"
DIG2_D (O,B)  "40"  "'fi:r-t_sICst+"
```

Abbildung 2.5: Ausschnitt aus dem SVOX-Morphemlexikon

2.2.1 Masterlexika

Abbildung 2.6 zeigt einen kleine Ausschnitt aus einem Masterlexikon für Nomenstämme. Jeder Lexikoneintrag belegt eine Zeile in der Datei und die einzelnen Merkmale sind durch ein Semikolon (`;`) voneinander getrennt. In diesem Lexikon werden für jeden Eintrag folgende Informationen gespeichert (Als Beispiel die entsprechenden Werte des ersten Eintrages):

- Graphemische Form: `Konten`
- Phonetische Form: `'kOnt@n`
- Konstituententyp: `NS` (Nomenstamm)
- Geschlecht: `N` (Neutrum)

```
Konten;'kOnt@n;NS;N;SK0;PK1;D;D;NI;LE;NI;NI
Konti;'kOnti;NS;N;SK0;PK1;D;D;NI;LE;NI;NI
Kontinent;kOnti'nEnt;NS;M;SK4;PK7;D;D;NI;LE;NI;NI
Konto;'kOnto;NS;N;SK2;PK2;D;D;NI;LE;NI;NI
//Konto;'kOnto;NS;N;SK1;PK1;D;D;NI;LE;NI;NI
Kontroll;kOn'trOl;NS;F;SK10;PK4;D;D;NI;LE;NI;NI
Konzert;kOn't_sErt;NS;N;SK3;PK7;D;D;NI;LE;NI;NI
Kopf;'kOp_f;NS;M;SK3;PK0;D;D;NI;LE;NI;NI
Kopfschmerz;'kOp_f,SmErt_s;NS;M;SK3;PK4;D;D;NI;LE;NI;NI
Kopie;ko'pi:;NS;F;SK1;PK0;D;D;NI;LE;NI;NI
Kopie;ko'pi:@;NS;F;SK0;PK3;D;D;NI;LE;NI;NI
```

Abbildung 2.6: Ausschnitt aus dem Masterlexikon für Nomenstämme

- Singularklasse: `SK0` (Klasse 0)
- Pluralklasse: `PK1` (Klasse 1)
- Herkunftssprache: `D` (Deutsch)
- Transkriptionssprache: `D` (Deutsch)
- Häufigkeit des Eintrages: `NI` (Keine Infos)
- Informationsquelle: `LE` (Lexikon)
- Benutzer: `NI` (Keine Infos)
- Status des Eintrages: `NI` (Keine Infos)

Zusätzlich zu den für das SVOX-System benötigten linguistischen Merkmalen werden im Masterlexikon auch noch allgemeine Informationen zu einem Eintrag abgespeichert. Eine separate Beschreibungsdatei legt dabei fest, welche Merkmale und Informationen für jeden Konstituententyp vorhanden sind und in welcher Reihenfolge diese in der Datei stehen. Aus den Masterlexika lassen sich nach Bedarf automatisch die SVOX-Lexika generieren, die je nach Anwendungsbereich nicht alle Einträge der Masterlexika enthalten müssen (z.B. nur Aussprechen von Zahlen bei einer automatischen Zeitansage).

Unser Java Lexicon Editor operiert auch auf den Masterlexika, liest und schreibt also dieses Format. Er verwendet für die Beschreibung des Formats eine eigene Beschreibungsdatei, die im Kapitel 5 genauer erläutert wird.

Da wir in unserem Editor noch eine sogenannte History eingebaut haben, die Veränderungen an den Einträgen festhalten soll, haben wir das Masterlexikon-Format noch ein wenig erweitert, um History-Einträge direkt im Masterlexikon abspeichern zu können. Alle Einträge im Masterlexikon, die mit `//` beginnen sind History-Einträge und gehören zum nächstoberen Nicht-History-Eintrag. Sie sind nur für den Lexicon Editor von Belang und lassen sich durch die vorangestellten `//` einfach in der Datei identifizieren.

Kapitel 3

Die Benutzeroberfläche (GUI)

Eines unserer Hauptziele beim Design des Editor war, dass das Bearbeiten von Lexika, seien sie noch so umfangreich, schnell, unkompliziert und ohne grossen Aufwand möglich sein soll. Da für diesen Zweck ein gut strukturiertes graphisches Benutzerinterface (GUI) unabdingbar ist, haben wir uns vor allem am Anfang der Arbeit längere Zeit mit dem Design eines für unsere Zwecke geeigneten GUIs befasst.

Die wichtigen Editierfunktionen sollten schnell verfügbar sein, ohne Umwege über irgendwelche Menüs oder Dialoge. Wir haben uns für eine simple Tabelle als Haupteditierhilfsmittel entschieden, weil darin viele Einträge übersichtlich und gleichzeitig dargestellt werden können und die Einträge direkt in der Tabelle verändert werden können. In dieser Hinsicht wären spezielle Eingabemasken viel weniger effizient gewesen. Zusammen mit den Möglichkeiten zur Sortierung und Filterung der Tabelle ist dies bereits ein mächtiges Werkzeug, welches durch weitere Funktionen und Hilfsmittel unterstützt wird.

Es folgt ein kurzer Überblick über die Elemente des GUIs des Lexikon Editors. Das gesamte Fenster ist in Abbildung 3.1 abgebildet.

3.1 Tabelle

Der Hauptbestanteil des Java Lexicon Editors ist die Tabelle, in welcher die Lexikoneinträge dargestellt werden (Abbildung 3.2). Diese Tabelle erlaubt das übersichtliche Darstellen der Lexikoneinträge sowie das einfache Bearbeiten, denn alle Zellen sind editierbar und allfällige Änderungen lassen sich direkt in der Tabelle vornehmen.

Dabei beschreibt jede Zeile ein Wort des Lexikons, wobei die Spalten Attribute bzw. Features des Wortes darstellen. Die Beschreibung der Tabelle ist bewusst allgemein gehalten, es kann aber davon ausgegangen werden, dass jeder Eintrag stets eine graphemische und eine phonetische Form hat und in eine bestimmte Kategorie gehört. Beliebig viele weitere Features wie Verbstämme oder Verbklasse werden auf der gleichen Zeile angezeigt, wobei die Reihenfolge und Breite der Spalten nach Belieben geändert werden kann.

Die Tabelle kann Lexikoneinträge mit unterschiedlichen Kategorien zusammen anzeigen, indem gewisse Spalten für mehrere Features gleichzeitig genutzt werden. Die Spaltenüberschriften passen sich dabei immer der Kategorie das ausgewählten Eintrages an, und falls

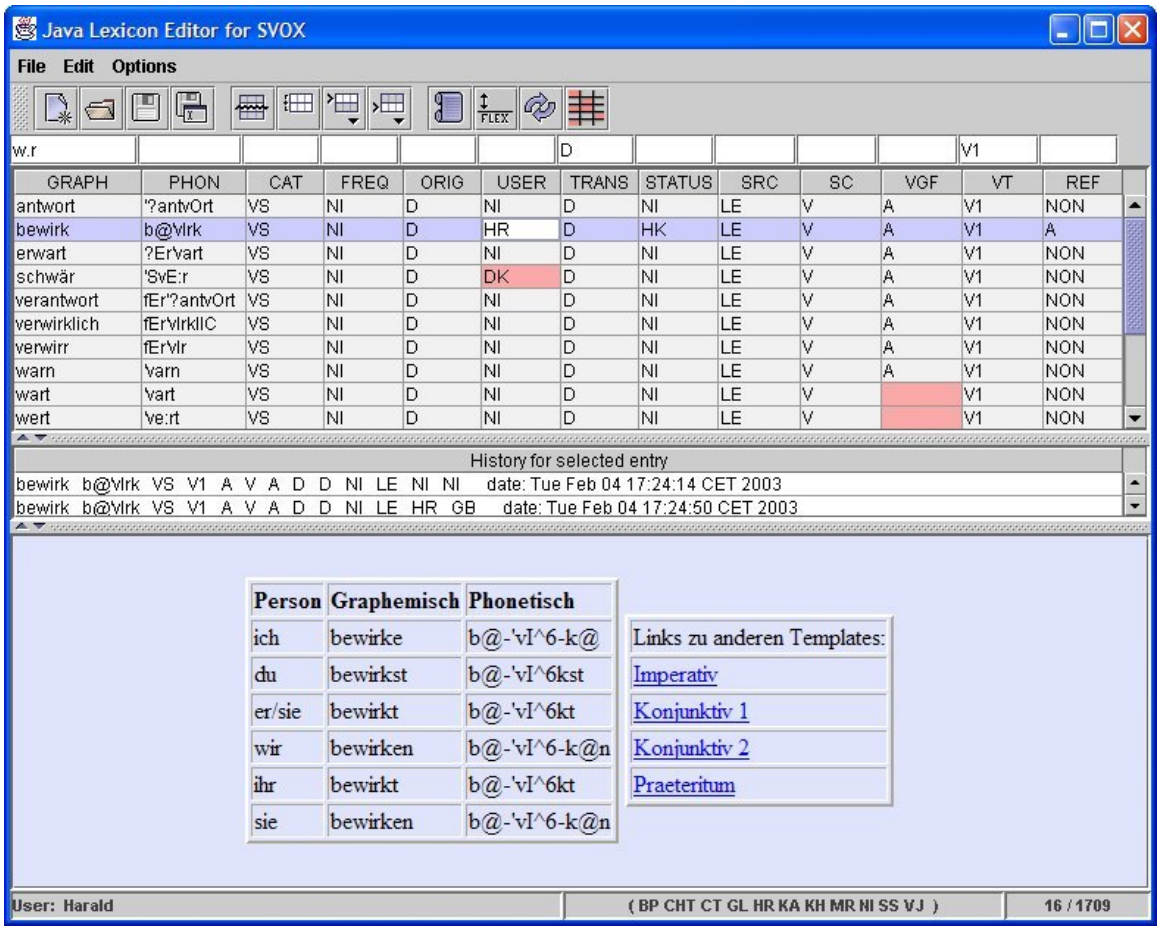


Abbildung 3.1: Das Hauptfenster des Lexikon Editors

GRAPH	PHON	CAT	FREQ	ORIG
abbau	?apba_u	VS	NI	D
abdräng	?apdrEN	VS	NI	D
abfall	?apfal	VS	NI	D
abfiel	?apfi:l	VS	NI	D
abflau	?apfla_u	VS	NI	D

Abbildung 3.2: Ein Ausschnitt aus der Tabelle

eine Kategorie nicht die maximale Anzahl Features besitzt, bleiben die entsprechenden Zellen uneditierbar. So geht zwar eine gewisse Übersichtlichkeit verloren, aber gleichzeitg gewinnen wir Platz, da weniger Spalten benötigt werden, und die meisten Features sind sowieso bei allen Kategorien enthalten.

Die Einträge in der Tabelle lassen sich aufsteigend oder absteigend nach einer beliebigen Spalte sortieren, indem auf die entsprechende Spaltenüberschrift geklickt wird.[1] Durch Anwenden von Filtern können auch nur bestimmte Einträge des Lexikons in der Tabelle angezeigt werden.

3.2 History

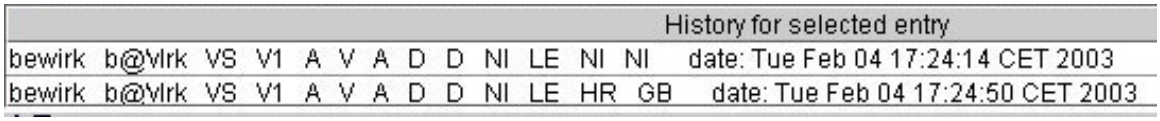
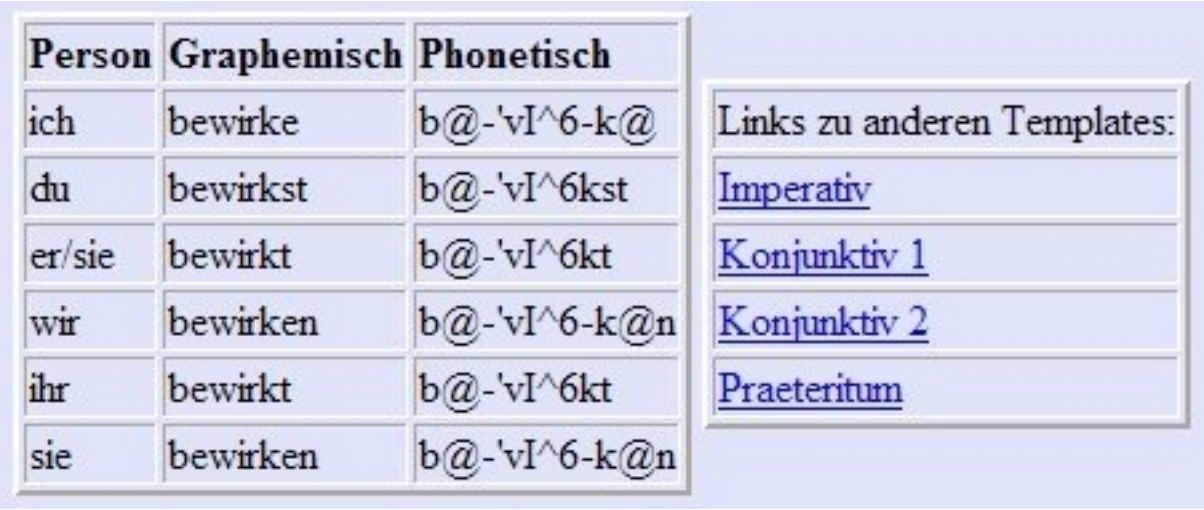


Abbildung 3.3: History

In diesem ausblendbaren Teilfenster werden alle Änderungen aufgelistet, die jemals an einem Lexikoneintrag vorgenommen wurden (Abbildung 3.3). Das ist wichtig, wenn mehrere Leute an den Lexika arbeiten und jeder eigene Änderungen vornimmt. Die Personen können dann sehen, wann und von wem Änderungen gemacht wurden und können nötigenfalls mit dem Betreffenden Rücksprache halten.

3.3 Flektionen



Person	Graphemisch	Phonetisch
ich	bewirke	b@-'vI^6-k@
du	bewirkst	b@-'vI^6kst
er/sie	bewirkt	b@-'vI^6kt
wir	bewirken	b@-'vI^6-k@n
ihr	bewirkt	b@-'vI^6kt
sie	bewirken	b@-'vI^6-k@n

Links zu anderen Templates:
Imperativ
Konjunktiv 1
Konjunktiv 2
Praeteritum

Abbildung 3.4: Ein Beispiel HTML-Template für die Flektion

[1] Um absteigend zu sortieren, während dem Klicken die Shift-Taste gedrückt halten.

Das unterste ausblendbare Teilfenster ist zur Anzeige von HTML-Dateien konzipiert, die zum Beispiel die Konjugationen eines angewählten Verbs enthält (Abbildung 3.4). Zusätzlich zu normalen HTML-Tags können die Templates noch spezielle Tags enthalten, die flektierte Formen von Einträgen darstellen können. Wie genau die Flektionsgenerierung funktioniert, wird im Kapitel 6 erläutert.

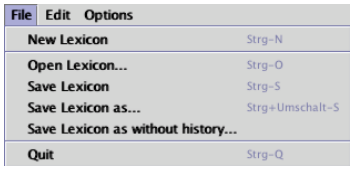
3.4 Toolbar

Die Toolbar bietet einen schnellen Zugriff auf fast alle Editierfunktionen des Lexikons (Abbildung 3.5). Alle Befehle der Toolbar sind auch über die Menüs zugänglich und sind in der Tabelle 3.1 aufgelistet.

Abbildung 3.5: Die Toolbar

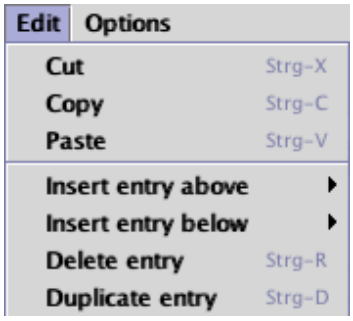
3.5 Menüs

Die Menubar enthält alles, was schon in der Toolbar vohanden ist, erweitert diese aber noch um ein paar weitere Funktionen (siehe folgende Auflistung). Fast alle Menübefehle lassen sich für einen schnellen Zugriff über ein Tastaturkürzel aufrufen.

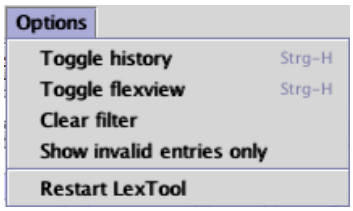


Save Lexikon as without history Speichert das geöffnete Lexikon unter einem neuen Namen und ohne die History.

Quit Beendet den Lexikon Editor.



Cut, Copy, Paste Ausschneiden, Kopieren und Einsetzen von Text in und aus Textfelder und Tabellenzellen.



Restart Lextool Beendet das Flektionstool und startet es neu. Nützlich wenn Dateien des Flektionstool während dem Betrieb geändert werden müssen.

	Erstellt ein neues, leeres Lexikon.
	Öffnet bereits existierende Lexika.
	Speichert die Änderungen im zur Zeit geöffneten Lexikon.
	Speichert eine Kopie des aktuellen Lexikons unter einem neuen Namen.
	Löscht die aktuelle Zeile bzw. alle markierten Zeilen aus der Tabelle. Die Auswahl mehrerer aufeinander folgender Zeilen kann durch Halten der Shift-Taste erzielt werden. Um einzelne Zeilen gezielt auszuwählen kann die Control-Taste verwendet werden.
	Erstellt eine Kopie aller angewählten Zeilen. Diese Funktion ist praktisch für die Eingabe vieler Einträge, bei welchen ein Grossteil der Werte übereinstimmt.
	Fügt eine neue Zeile unterhalb des gerade angewählten Eintrags in die Tabelle ein. Beim Drücken auf den Button erscheint ein Menü, aus welchem die Kategorie der neuen Zeile bestimmt werden muss.
	Fügt eine leere Zeile oberhalb des aktuellen Eintrags in die Tabelle ein.
	Stellt die Originalsicht des Lexikons wieder her, indem sämtliche Filter zurückgesetzt werden.
	Blendet das Teilfenster mit den History-Einträgen ein und aus.
	Blendet das Teilfenster zur Anzeige der Flektionen ein und aus.
	Filtert alle korrekten Einträge aus dem Lexikon, so dass nur noch solche Einträge sichtbar sind, die ungültige Werte enthalten.

Tabelle 3.1: Übersicht über die Toolbar-Icons

3.6 Statusleiste

Abbildung 3.6: Die Statusleiste

Am unteren Fensterrand befindet sich die Statusleiste (Abbildung 3.6). Sie ist ein Hilfesystem, das Informationen und Erklärungen zur Tabelle, insbesondere zur aktuell ausgewählten Zelle in der Tabelle anzeigt. Wir haben die Statusleiste ToolTips[2] vorgezogen, denn die Erklärungen sind so immer vorhanden, auch wenn man mit der Tastatur durch die Tabelle navigiert, und sie stören nicht wenn man sie nicht benötigt. Sie soll zum einen bei der Eingabe von Werten in die Tabelle helfen, indem sie mögliche Werte angibt, zum anderen auch als Beschreibung der vielen kurzen und kryptischen Kürzel der Featurenamen und -werten.

Die Statusleiste ist dreigeteilt und bietet in den drei Anzeigefelder von links nach rechts folgende Information:

- Eine Beschreibung des Inhaltes der ausgewählten Zelle in der Tabelle. Die Beschreibung enthält eine Beschreibung des aktuellen Features (Spalte) und eine Beschreibung zum aktuellen Featurewert in der Zelle.
- Alle möglichen Featurewerte, die in der aktuellen Zelle erlaubt sind.
- Anzahl angezeigte Einträge in der Tabelle / Anzahl Einträge im Lexikon.

3.7 Login-Fenster

Das Login-Fenster dient zur Identifikation des aktuellen Benutzers und wird bei jedem Start des Editors eingeblendet. Das Benutzerkürzel wird beim Erstellen von neuen Einträgen dann automatisch in die Tabelle eingefügt, und beim Ändern eines Eintrages wird das Kürzel entsprechend dem aktuellen Benutzer angepasst.

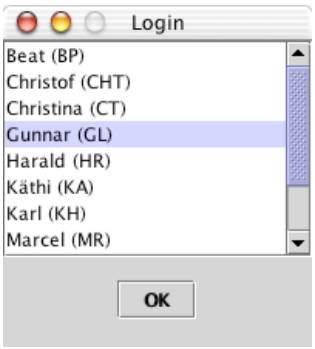


Abbildung 3.7: Das Login-Fenster

[2]ToolTips sind Erklärungen die in einer kleinen Box erscheinen, wenn man mit der Maus einige Zeit über dem fraglichen Objekt verweilt.

Kapitel 4

Editierhilfen im Überblick

In diesem Kapitel wollen wir auf alle implementierten Funktionen, welche die Arbeit mit dem Java Lexicon Editor benutzerfreundlicher machen, kurz zu sprechen kommen.

4.1 Such-Filter

Über jeder Tabellenspalte befindet sich eine Textbox, in die man Suchbegriffe eingeben kann. Nach Drücken der Enter-Taste wird sich die aktuelle Sicht des Lexikons auf jene Zeilen beschränken, welche den Suchbegriff im entsprechenden Spaltenwert enthalten. Es ist möglich gleichzeitig über mehreren Spalten einen Filter anzulegen, wobei als Suchbegriffe sogar reguläre Ausdrücke zulässig sind (Siehe Abbildung 4.1).

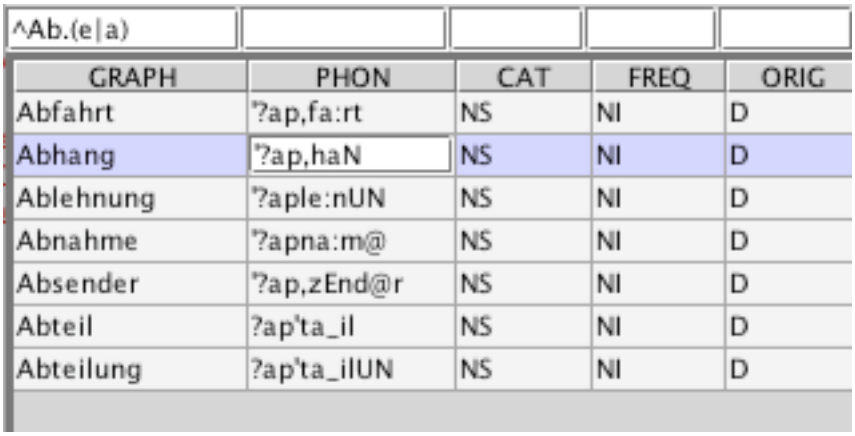


Abbildung 4.1: Der Filter in Aktion

4.2 Hervorhebung ungültiger Werte

Enthält ein Lexikon ungültige Werte oder noch leere Einträge, so werden diese durch eine rote Markierung hervorgehoben (Abbildung 4.2). Es kann somit äusserst effizient festgestellt

werden, welche Einträge noch zu korrigieren sind. Für den Fall, dass man nur die Zeilen angezeigt haben möchte, die noch fehlerhafte Werte enthalten, ist dafür eigens ein Button in der Toolbar vorgesehen.

Welche Werte gültig und welche ungültig sind entscheidet der Editor anhand den Feature-Definitionen in der Beschreibungsdatei *master_description* (Siehe Kapitel 5.1 und Anhang C).

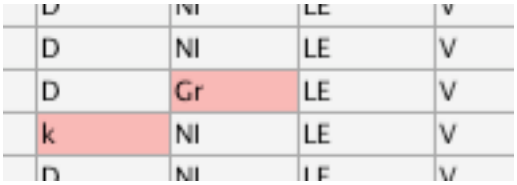


Abbildung 4.2: Ungültige Werte werden rot markiert

4.3 Auto-Completion

Mit Auto-Completion ist gemeint, dass beim Editieren der Tabelle automatisch Werte vervollständigt werden, sobald der Anfang des eingegebenen Wertes mit einem möglichen Wert übereinstimmt. Es wird nach jedem Tastendruck ein Wert angezeigt, der aufgrund der bisherigen Eingabe möglich wäre. Gleichzeitig wird der automatisch vervollständigte Teil markiert, so dass ein weiterer Tastendruck diesen gleich wieder überschreibt. Das Ziel war, ein ähnliches Verhalten wie bei den Eingabefeldern für Internetadressen in Web-Browsern zu erhalten.

4.4 Automatische phonetische Transkription

Wird von einem Eintrag die graphemische Form eingegeben oder verändert, bei dem noch keine phonetische Form eingetragen ist, schlägt der Editor eine mögliche phonetische Schreibweise vor und trägt sie gleich in die Tabelle ein. Die vorgeschlagene phonetische Schreibweise wird in den meisten Fällen nicht perfekt richtig sein, aber zumindest so gut, dass sie mit wenig Aufwand nachträglich von Hand angepasst und korrigiert werden kann. Das spart viel Zeit, denn eine komplette manuelle Umschrift braucht einiges an Erfahrung in Linguistik und Übung mit der phonetischen Schrift.

Die Generierung der phonetischen Umschrift geschieht mit dem gleichen externen Tool, das auch für die Flektierung der Lexikoneinträge verantwortlich ist.

4.5 Anzeige möglicher Featurewerte und deren Bedeutung

Sobald eine Zelle in der Tabelle den Fokus erhält, werden alle erlaubten Werte dieser Zelle in der Statusbar aufgelistet. Die Bedeutung des Wertes in der angewählten Zelle wird ebenfalls

in der Statusbar angezeigt. Dies erleichtert einerseits die Eingabe von Werten und hilft auch, wenn man von einem eingegebenen Wert die Beschreibung schnell haben möchte.

4.6 Generierung der Flektionen

Die Anzeige der Flektionen soll die Erkennung von semantisch fehlerhaften Featurewerten erleichtern. Beispielsweise kann ein Verbstamm einen syntaktisch korrekten Wert als Verbklasse enthalten, der aber dazu führt, dass das Verb völlig falsch konjugiert wird. Da aufgrund der automatischen Flektionsgenerierung sofort alle Konjugationen angezeigt werden können, ist nun schnell entscheidbar, ob die eingegebene Verbklasse Sinn ergibt oder nicht.

Desweiteren lassen sich auf diese Weise auch Fehler in den der Flektion zu Grunde liegenden Grammatikregeln aufdecken, wenn trotz richtigen Featurewerten falsche oder unsinnige Formen generiert werden.

Kapitel 5

Externe Beschreibungsdateien

In diesem Kapitel werden die beiden externen Textdateien behandelt, durch die der Lexikon Editor konfiguriert werden kann.

Da sich das Format der SVOX-Masterlexika im Laufe der Zeit ändern kann, sollte der Lexikon Editor sich daran einfach anpassen können und sollte darum das Format nicht fest eingebrannt bekommen. Diesen Dienst tut uns die Datei *master_descripton.txt*, die das Lexikon-Format in textueller Form beschreibt. Auf sie wird im nächsten Abschnitt genauer eingegangen.

Daneben sollen auch Eigenschaften des Lexikon Editors selbst konfigurierbar sein und Benutzereinstellungen sollen gespeichert werden, damit sie nicht bei jedem neuen Start wieder manuell hergestellt werden müsen. Diesen Zweck erfüllt die Datei *preferences.txt*, die im übernächsten Abschnitt behandelt wird.

5.1 Die Datei *master_description.txt*

In der Datei *master_description.txt* wird das Format der Masterlexika beschrieben. Insbesondere werden die folgenden Eigenschaften definiert:

- Der Name und die Beschreibung aller möglichen Wortkategorien.
- Der Name und die Beschreibung aller möglichen Features.
- Die erlaubten Feature-Werte aller Features inkl. Bschreibung.
- Die Features und deren Reihenfolge im Lexikon aller Wortkategorien.

Die gesamte Datei, wie wir sie im Rahmen dieser Arbeit erstellt haben, ist im Anhang C auf Seite 99 zu finden.

5.1.1 Allgemeines Format

Die Datei ist eine ASCII Text-Datei mit einer *ISO Latin 1* Codierung für die Sonderzeichen und Umlaute und sollte als Zeilenbegrenzungszeichen LF (Linefeed) verwenden. Dies sind die Standardeinstellungen unter UNIX.

In der Datei treten zwei Arten von Definitionen auf: zum einen können Wortkategorien, zum anderen Features definiert werden. Die Reihenfolge der Definitionen in der Datei spielt dabei keine Rolle, da die Datei vom Editor zweimal geparst wird, um beim ersten Durchgang alle Feature-Definitionen einzulesen und beim zweiten Durchgang alle Kategorien aufzubauen. Feature- und Kategorie-Definitionen dürfen also auch gemischt werden, das sollte aber der Übersicht wegen nach Möglichkeit vermieden werden.

Zeilen, die mit `//` beginnen, sind Kommentare und werden vom Parser ignoriert. Kommentare müssen immer auf einer eigenen Zeile stehen, d.h. Kommentare dürfen nicht auf Zeilen beginnen, die bereits eine Definition enthalten. Bereichskommentare mit `/*` und `*/` wie in C und Java sind nicht erlaubt.

5.1.2 Feature-Definitionen

Feature-Definitionen können folgende zwei Formen haben:

```
[Feature] Name "Description"
   [ Value "ValueDescr" Value "ValueDescr" ... ]
[End Feature]
```

```
[Feature] Name "Description"
   { Token Token ... }
[End Feature]
```

Name Der Name des Features. Da dieser Name verwendet wird, um Flektionen zu generieren, sollte er konsistent sein mit demjenigen des Flektionstools. Dieser Name wird auch für die Spaltenüberschriften in der Tabelle des Editors verwendet.

Description Eine Beschreibung des Features. Diese wird im Editor in der Statusleiste als Hilfe bei der Eingabe eingeblendet. Sie darf ausser `"` alle Zeichen enthalten und darf auch ganz fehlen.

Value Ein gültiger Wert für das Feature. Bei der Eingabe prüft der Editor, ob die Eingabe mit einem der hier definierten Werte übereinstimmt. Da diese Werte für die Flektionsgenerierung verwendet werden, sollten sie konsistent mit denjenigen des Flektionstools sein.

ValueDescr Eine Beschreibung des Feature-Wertes. Diese wird als Hilfe in der Statusleiste des Editors eingeblendet. Sie darf ausser `"` alle Zeichen enthalten und darf auch ganz fehlen.

Token Ein Token, das im Wert für das Feature vorkommen darf. Der Wert eines solchen Features ist nur gültig, wenn er aus einer beliebigen Folge von definierten Tokens besteht.

Die Werteliste darf beliebig viele Werte zusammen mit Beschreibungen definieren und darf auch durch einen Linefeed getrennt über mehrere Zeilen gehen.

Die Features `GRAPH`, `PHON` und `CAT` spielen für den Editor eine zentrale Rolle und dürfen deshalb nicht umbenannt werden! Die Werteliste darf hingegen beliebig verändert werden.

Als Beispiel hier die Definition der Features `SC` und `GRAPH`:

```
[Feature] SC "Surface Class"
  [ V "Vollverb" AUXH "Hilfsverb haben" AUXS "Hilfsverb sein"
    AUXW "Hilfsverb werden" MVERB "Modalverb" NI "No Info" ]
[End Feature]
```

```
[Feature] GRAPH "Graphemic Form"
  { a b c d e f g h i j k l m n o p q r s t u v w x y z
    ä ö ü ë ï â ê î ô û á é í ó ú à è ì ò ù ç ñ Â Ê Î Ô
    Û Ã Ñ À Ì Á Í Ä Ö Ü ø $ + # }
[End Feature]
```

5.1.3 Kategorie-Definitionen

Kategorie-Definitionen haben folgende Form:

```
[Category] Name "Description"
   [ FeatureName FeatureName ... ]
[End Category]
```

Name Der Name der Kategorie. Da dieser Name für die Flektionsgenerierung verwendet wird, sollte er konsistent mit demjenigen des Flektionstools sein.

Description Eine Beschreibung der Kategorie. Sie darf ausser " alle Zeichen enthalten und darf auch ganz fehlen.

FeatureName Ein Name eines definierten Features. Er muss mit einem Namen einer Feature-Definition übereinstimmen.

Die Reihenfolge der Features in der Featureliste bestimmt, in welcher Reihenfolge die Feature-Werte eines Eintrages in die Lexikon-Datei geschrieben werden und umgekehrt welchen Features die Werte in der Lexikon-Datei zugeordnet werden. Die Reihenfolge der Werte in den Masterlexika muss also konsistent sein mit der Reihenfolge der Features in der Kategorie-Definition.

Die Features `GRAPH`, `PHON` und `CAT` spielen für den Editor eine zentrale Rolle und müssen deshalb im Masterlexikon immer als erste drei Features auftauchen.[1] Dies sind auch die Features, die jeder Lexikoneintrag sicher haben muss.

Als Beispiel hier die Definition der Kategorie `VS` (Verbstamm):

```
[Category] VS
  [ GRAPH PHON CAT VT VGF SC REF ORIG TRANS FREQ SRC USER STATUS ]
[End Category]
```

[1] Da die Position der Features durch die Kategorie-Definition festgelegt wird und die Kategorie selbst auch ein Feature ist, muss zumindest die Position von `CAT` fest bekannt sein.

5.2 Die Datei *preferences.txt*

In der Datei `preferences.txt` werden Einstellungen, die nur den Lexikon Editor und dessen Benutzung betreffen, gespeichert. Insbesondere werden folgende Eigenschaften festgelegt:

- Anordnung und Breite der Spalten in der Tabelle
- Aktueller Benutzer
- Fenstergrösse und Fensterposition
- Aktuelles Arbeitsverzeichnis
- Grösse der History- und Flektionsansicht
- Pfad zum Flektionstools
- Befehle zur Steuerung des Flektionstools
- HTML Template-Dateien für die Flektionsansicht

Einige der gespeicherten Eigenschaften sind nur für interne Zwecke gedacht und müssen nie manuell geändert werden, andere Eigenschaften bedürfen je nach Arbeitsumgebung zuerst einer manuellen Konfiguration, bevor der Lexikon Editor problemlos arbeitet.

Die gesamte Datei, wie wir sie im Rahmen dieser Arbeit erstellt haben, ist im Anhang D auf Seite 103 zu finden.

5.2.1 Allgemeines Format

Die Datei ist eine ASCII Text-Datei mit einer *ISO Latin 1* Codierung für die Sonderzeichen und Umlaute und sollte als Zeilenbegrenzungszeichen LF (Linefeed) verwenden. Dies sind die Standardeinstellungen unter UNIX.

Wir haben bewusst ein einfaches und für den Menschen gut lesbares Format gewählt, da wir im Lexikon Editor selber keine Möglichkeit anbieten, die Einstellungen über ein Dialogfeld zu ändern. Da aber die Einstellungen nach erstmaliger Konfiguration nur selten geändert werden müssen, stellt dies kein gewichtiger Nachteil dar.

Die Einstellungen werden zeilenweise gespeichert und jede Zeile hat folgendes Format:

Key = *Value*

Key Ein String, der eine Einstellung bezeichnet. Der String darf kein Gleichheitsszeichen (=) enthalten. Der Key darf nur in speziellen Fällen verändert werden (siehe nächster Abschnitt), da der Editor sich auf den Key abstützt.

Value Ein String, der dem Wert der entsprechenden Einstellung entspricht. Der String darf kein Gleichheitsszeichen (=) enthalten.

Die Datei *preferences.txt* wird beim Start des Editors geparst und die Key-Value Paare werden intern zur Verwaltung in eine Hash-Tabelle gespeichert. Von jeder Zeile, die ein = enthält, wird angenommen, dass sie ein gültiges Key-Value Paar darstellt. Zeilen ohne = werden als Kommentarzeilen angesehen und einfach ignoriert. Die Key-Value Paare werden beim Beenden des Editors in sortierter Form wieder in die Datei zurückgeschrieben.

5.2.2 Bedeutung der einzelnen Einstellungen

Hier sollen nun die einzelnen Zeilen der Datei aufgelistet und dessen Bedeutung und Benutzung erklärt werden.

```
COLUMN_POSITION_0 = 0
COLUMN_POSITION_1 = 1
        ⋮
COLUMN_POSITION_12 = 12
```

Diese Zeilen speichern die **Anordnung der Spalten** der Lexikontabelle. Die Spalten können im Editor direkt umgeordnet werden, diese Einstellungen müssen also nie direkt geändert werden.

```
COLUMN_WIDTH_0 = 118
COLUMN_WIDTH_1 = 118
        ⋮
COLUMN_WIDTH_12 = 69
```

Diese Zeilen speichern die **Breite der Spalten** der Lexikontabelle. Da die Spalten im Editor mit der Maus auf die gewünschte Breite gezogen werden können, müssen diese Einstellungen nie direkt geändert werden.

```
DIVIDER_FLEX = 598
DIVIDER_HISTORY = 382
HIDE_FLEXVIEW = TRUE
HIDE_HISTORY = TRUE
```

Diese Zeilen speichern Informationen zur **Grösse und Sichtbarkeit der History- und Flektionsansicht**. Diese Werte werden vom Editor selbst bestimmt und müssen nie manuell geändert werden.

```
HTML_TEMPLATE_DEFAULT = default.html
```

Diese Zeile legt die **Standard HTML-Datei** fest, die in der Flektionsansicht angezeigt werden soll, wenn sonst kein passendes Template gefunden wurde, z.B. weil eine Flektion für die angewählte Kategorie keinen Sinn macht. Die Datei muss sich im Ordner *html* befinden, der sich im selben Verzeichnis befindet wie der Editor.

```
HTML_TEMPLATE_NS = ns.html
HTML_TEMPLATE_VS = vs_ggw.html
          ⋮
```

Diese Zeilen legen die **Standard HTML-Templates für die einzelnen Wortkategorien** fest, die in der Flektionsansicht angezeigt werden sollen. Der letzte Teil des Keys (`NS` bzw. `VS`) muss mit einem Kategorienamen übereinstimmen, der in der Datei *master_description.txt* definiert wurde, denn anhand der Kategorie eines Eintrages wird nach einem entsprechenden Template gesucht. Die Templates müssen sich im Ordner *html* befinden, der sich im selben Verzeichnis befindet wie der Editor.

```
LEXTOOL_BASE_CONS_COMMAND = \cons $CONS
LEXTOOL_BASE_FEAT_COMMAND = \feat $FEAT_NAME $FEAT_VALUE
```

Diese Zeilen definieren die Kommandos, die an das Flektionstool gesendet werden, wenn die **Basis-Konstituente und Basis-Features gesetzt werden** müssen. Die Tokens `$CONS`, `$FEAT_NAME`, `$FEAT_VALUE` sind dabei Platzhalter, die vor dem Versenden des Kommandos durch die Werte der Wortkategorie, des Featurenamens und des Featurewerts des aktuellen Lexikoneintrages ersetzt werden.

```
LEXTOOL_TARGET_CONS_COMMAND = \flectCons $CONS
LEXTOOL_TARGET_FEAT_COMMAND = \flectFeat $FEAT_NAME $FEAT_VALUE
```

Diese Zeilen definieren die Kommandos, die an das Flektionstool gesendet werden, wenn die **Ziel-Konstituente und Ziel-Features gesetzt werden** müssen. Die Tokens `$CONS`, `$FEAT_NAME`, `$FEAT_VALUE` sind dabei Platzhalter, die vor dem Versenden des Kommandos durch die Werte des entsprechenden Tags im HTML-Template ersetzt werden.

```
LEXTOOL_CLEAR_COMMAND = \clear
```

Diese Zeile definiert das Kommando, das ans Flektionstool gesendet wird, wenn der **aktuelle Eintrag im Flektionstool gelöscht** und alle Features zurückgesetzt werden soll. Dieses Kommando sollte vor jeder Flektion als erstes gesendet werden.

```
LEXTOOL_FLECT_COMMAND = \flect
```

Diese Zeile definiert das Kommando, das ans Flektionstool gesendet wird, wenn die Flektion des Eintrags basierend auf den aktuell gesetzten Features beginnen soll. Das Resultat dieses Kommandos wird vom Editor ausgewertet und in die HTML-Templates eingesetzt.

```
LEXTOOL_GRAPH_COMMAND = \graph $GRAPH
```

Diese Zeile definiert das Kommando, das ans Flektionstool gesendet wird, wenn die **graphemische Form des aktuellen Eintrages gesetzt** werden soll. Der Platzhalter `$GRAPH` wird dabei vor dem Versenden durch die entsprechende graphemische Form des Eintrages ersetzt.

```
LEXTOOL_GRAPH_TO_PHON_COMMAND = \phon
```

Diese Zeile definiert das Kommando, das ans Flektionstool gesendet wird, wenn vom aktuellen Eintrages eins phonetische Umschrift basierend auf der graphemischen Form gemacht werden soll.

```
LEXTOOL_PHON_COMMAND = \phon $PHON
```

Diese Zeile definiert das Kommando, das ans Flektionstool gesendet wird, wenn die **phonetische Form des aktuellen Eintrages gesetzt** werden soll. Der Platzhalter `$PHON` wird dabei vor dem Versenden durch die entsprechende phonetische Form des Eintrages ersetzt.

```
LEXTOOL_OPTIONS = -c lexedit.cmd
LEXTOOL_PATH = /home/sprstud1/lexedit_v1.1/lexedit
```

Diese Zeilen definieren den **Pfad, der zum Flektionstool führt** und die **Commandline-Argumente**, mit denen das Tool gestartet werden soll. Der Pfad sollte nach Möglichkeit absolut sein.

```
LEXTOOL_PROMPT = SVOX LexEdit>
```

Diese Zeile definiert den **Prompt**, der das Flektionstool für die Kommunikation mit dem Benutzer verwendet. Der Prompt wird vom Editor verwendet, um das Ende einer Meldung des Flektionstools zu finden.

```
LEXTOOL_QUIT_COMMAND = \quit
```

Diese Zeile definiert das Kommando, das an das Flektionstool gesendet wird, wenn das **Tool beendet** werden soll.

```
USER = HR
```

Diese Zeile speichert das **Benutzerkürzel** des Benutzers, der den Editor zuletzt verwendet hat. Diese Einstellung muss nie manuell geändert werden.

```
WINDOW_HEIGHT = 766
WINDOW_POS_X = 168
WINDOW_POS_Y = 58
WINDOW_WIDTH = 994
```

Diese Zeilen speichern die **Position und die Grösse des Hauptfensters** des Lexikon Editors. Diese Einstellungen dienen internen Zwecken und müssen nicht manuell verändert werden.

```
WORKING_DIR = /home/sprstud1/LexikonEditor/Masterlexica Files
```

Diese Zeilen speichern den absoluten Pfad des **aktuellen Arbeitsverzeichnis**, das bei allen Öffnen- und Speicher-Dialogen direkt angesprungen wird. Beim Öffnen und Speichern wird das Arbeitsverzeichnis vom Editor neu gesetzt und muss daher hier nicht manuell verändert werden.

Kapitel 6

Die Flektionsgenerierung

6.1 Kurzer Überblick

Zur zusätzlichen Information, aber vor allem um fehlerhafte Werte im Lexikon schneller zu lokalisieren und Mängel in der Grammatik aufzudecken, ist es möglich, sich vom Java Lexicon Editor die Lexikoneinträge flektieren zu lassen. Dies geschieht im Moment mittels Kommunikation mit dem bereits existierenden Tool `lexedit`, das die Flektionen der aktuellen Tabellenzeile generiert, um sie anschliessend dem Lexicon Editor zu übermitteln. Die Aufgabe des Lexicon Editors besteht dann noch darin, die erhaltenen Informationen übersichtlich in einem Teilfenster darzustellen.

Der Flektionsmechanismus im Editor ist so allgemein wie möglich entworfen worden, sodass prinzipiell auch ein anderes Tool für die Flektionsgenerierung verwendet werden kann, sofern dieses auch über die Kommandozeile gesteuert werden kann.

6.2 Integration des Flektionstools

Da wir für die Flektionsgenerierung auf ein bereits vorhandenes Tool zurückgreifen konnten — die Implementation eines eigenen Flektionsalgorithmus hätte den Rahmen dieser Arbeit sowieso bei weitem gesprengt — mussten wir uns zuerst um die Integration dieses Tools in unseren Editor und die Kommunikation der beiden Programme kümmern.

Abbildung 6.1 zeigt schematisch, wie die Integration funktioniert. Wir haben eine Klasse *LexTool* geschrieben, die als Schnittstelle zum eigentlichen Flektionstool für andere Klassen agiert, indem sie statische Methoden für das Versenden von Befehlen und Empfangen der Resultate zur Verfügung stellt. Beim Start des Editors startet die Klasse *LexTool* das Flektionstool als separaten Prozess ausserhalb der Java Virtual Machine und klinkt sich in den standard In- und Output des Flektionstools ein. Dabei entnimmt der Editor der Preference-Datei, wo sich das Flektionstool befindet (`LEXTOOL_PATH`) und mit welchen Argumenten es gestartet werden soll (`LEXTOOL_OPTIONS`). Falls das Tool gestartet werden konnte, kann die Klasse *LexTool* über einen Outputstream und einen Inputstream Kommandos direkt an das Flektionstool senden respektive Antworten vom Tool wieder empfangen und auswerten.

Wenn eine Methode des Editors nun ein Kommando an das Flektionstool senden will,

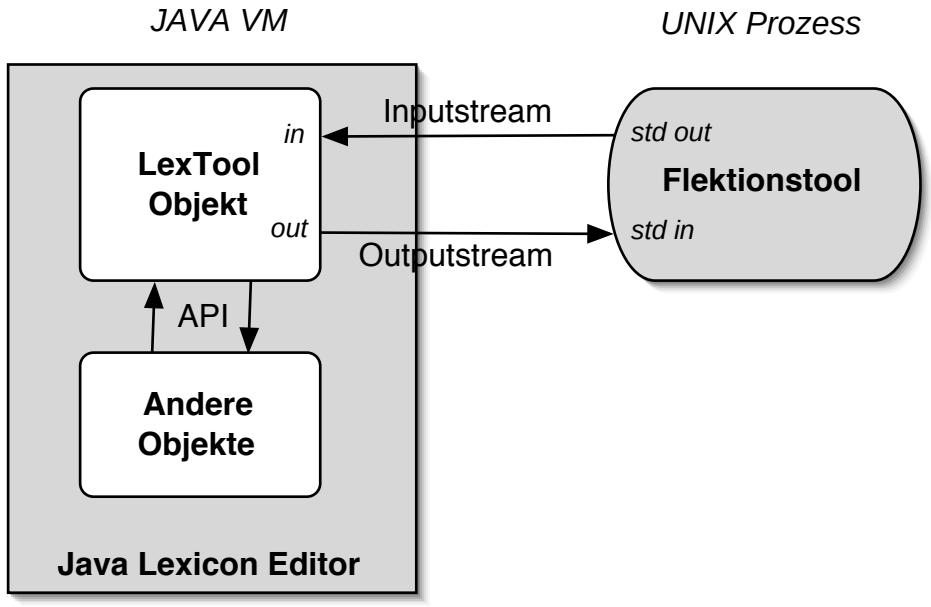


Abbildung 6.1: Integration des Flektionstool in den Editor

verwendet es die statische Methode *sendCommand* der Klasse *LexTool*, welche dann zuerst den aktuellen Inhalt des Inputstreams löscht[1] und dann das gewünschte Kommando auf den Outputsream schreibt. Eine allfällige Antwort des Flektionstools kann man mit der Methode *getResult* der Klasse *LexTool* empfangen.

Da die Streams immer offen sind und man nicht im Voraus weiss, wie lange es dauert, bis das Tool seine Antwort vollständig auf den Stream geschrieben hat, ergibt sich das Problem, dass man erkennen muss, wann die Antwort vollständig angekommen ist und man mit dem Lesen aus dem Stream aufhören kann. Da das Flektionstool (jedenfalls das jetztige Tool *lexedit*) nach jeder Antwort einen Prompt ausgibt, orientiert sich *LexTool* an diesem Prompt und hört auf vom Stream zu lesen, wenn der Prompt im Stream auftaucht. Wie der Prompt aussieht, ist in der Preference-Datei unter `LEXTOOL_PROMPT` anzugeben.

6.3 Ablauf einer Flektion

Abbildung 6.2 zeigt schematisch, wie die Generierung einer flektierten Form abläuft.

Die Generierung beginnt, sobald im HTML-Template, das in der Flektionsansicht gerendert werden soll, ein `<CONS>`-Tag auftaucht (Siehe dazu nächster Abschnitt). Dieses wird geparst und für spätere Zwecke gespeichert. An das Flektionstool wird zunächst das Kommando `LEXTOOL_CLEAR_COMMAND` gesendet, welches wie alle folgenden Kommandos in der Preference-Datei definiert ist. Dieses Kommando dient dazu, den gerade in Bearbeitung befindlichen Eintrag zu löschen und allfällige Parameter zurückzusetzen. Danach werden nacheinander die graphemische und phonetische Form und die Wortkategorie (Konstituententyp)

[1] Dies wird gemacht, um sicherzustellen dass nach dem Senden des Kommandos nur noch Antworten auf das zuletzt gesandte Kommando im Stream auftauchen.

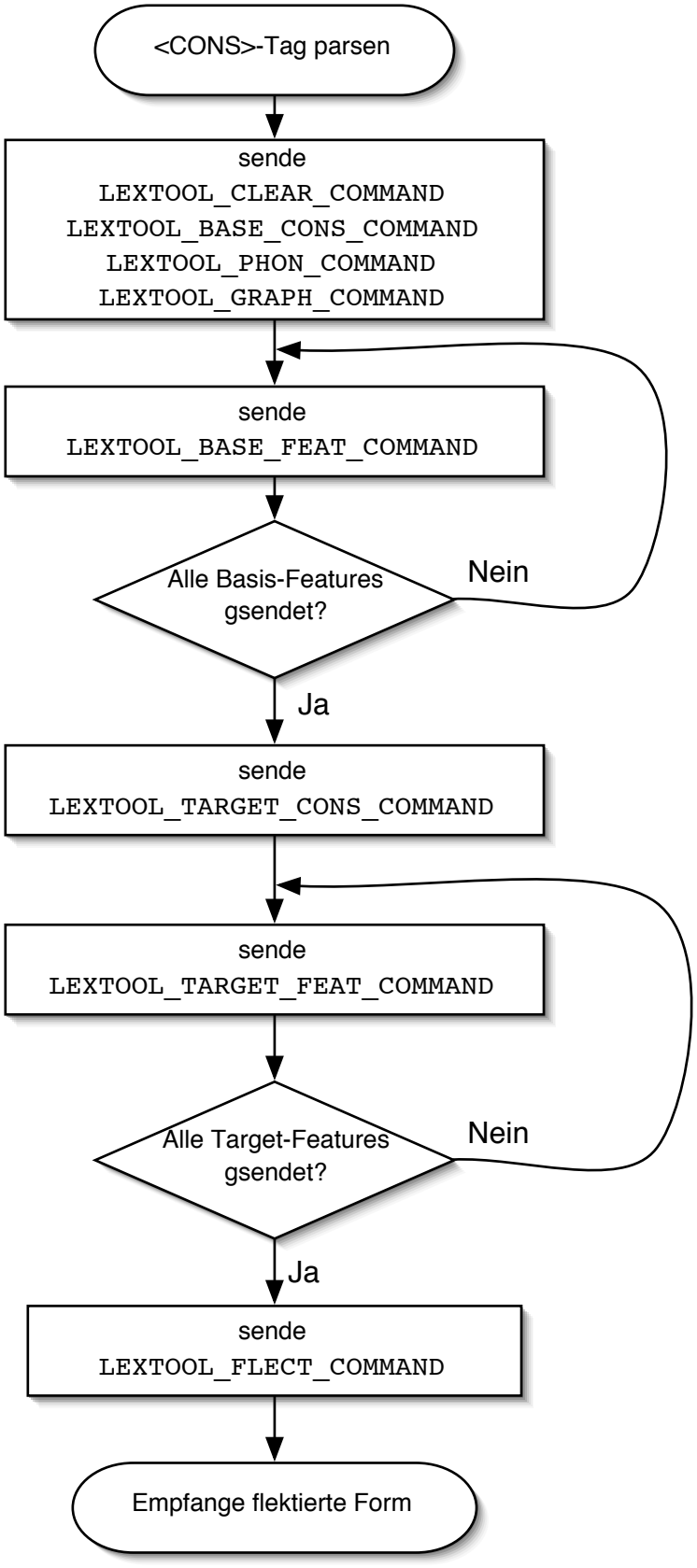


Abbildung 6.2: Schematischer Ablauf einer Flektion

des aktuell in der Tabelle ausgewählten Eintrages an das Flektionstool übermittelt (Kommandos LEXTOOL_BASE_CONS_COMMAND, LEXTOOL_GRAPH_COMMAND, LEXTOOL_PHON_COMMAND). Danach wird wiederholt LEXTOOL_BASE_FEAT_COMMAND gesendet, bis vom aktuellen Eintrag alle Features gesendet wurden.

Die aus dem <CONS>-Tag extrahierten Target-Features und -Konstituenten werden jetzt mittels LEXTOOL_TARGET_CONS_COMMAND und LEXTOOL_TARGET_FEAT_COMMAND gesendet. Damit hat das Flektionstool genug Informationen, um sich auf die Suche nach einer passenden Konstituente zu machen, welche die verlangten Features aufweist. Die Flektion wird mit LEXTOOL_FLECT_COMMAND initiiert und falls alles gepasst hat, kann die flektierte Form danach vom Inputstream gelesen werden.

6.4 Formatierte Anzeige der Flektionen

Um die Darstellung der Flektionen im Lexicon Editor flexibel zu gestalten, haben wir uns entschieden, HTML-Templates zu verwenden. Das Teilfenster zur Anzeige der Flektionen ist in der Lage einfache HTML-Seiten anzuzeigen. Damit es aber auch möglich wird, dynamisch, je nach ausgewähltem Lexikoneintrag, die richtige Flektion im HTML-Code einzubauen, haben wir einen eigenen HTML-Tag namens <CONS> kreiert. Die Attribute dieses Tags spezifizieren, welche Flektion vorgenommen werden soll, und welche Resultate schlussendlich auf dem Bildschirm angezeigt werden. Wenn also eine Zeile des Lexikons markiert wird, kommuniziert der Lexikon Editor mit dem Flektionstool und erhält die anhand des <CONS>-Tags generierten Flektionen. Dabei wird der <CONS>-Tag in der HTML-Datei gerade durch die resultierende Flektion ersetzt. Dies wird für alle im HTML-Template vorhandenen <CONS>-Tags wiederholt, bis ein reines HTML-File entsteht, das zur Darstellung geladen werden kann. Der <CONS>-Tag hat folgendes Format:

<CONS *Parameter*>
Konstituent (*Feat_Name1* = *Feat_Value1* , *Feat_Name2* = *Feat_Value2* , ...)
</CONS>

Parameter Der <CONS>-Tag kann um die vier Ausdrücke GRAPH, PHON, FEAT und TYPE ergänzt werden. Diese Parameter bestimmen, welche der vom Flektionstool zurückgelieferten Werte ins HTML-Template eingefügt werden sollen. Dabei bedeutet GRAPH, dass die graphemische Form angezeigt werden soll. Analog wird mit PHON die phonetische Form sichtbar. FEAT und TYPE haben zur Folge, dass die übermittelten Features bzw. der Konstituent ebenfalls dargestellt werden. Es sind mehrere Ausdrücke gleichzeitig als Parameter zulässig, wobei die Reihenfolge egal ist.

Konstituent Der Target-Konstituententyp, den die flektierte Form haben soll. Da Lexikoneinträge flektiert werden, beginnt dieser in den meisten Fällen mit LEX_.

Featureliste Die zum Konstituenten passenden Features, die gesetzt werden sollen, um die Flektion in die gewünschte Richtung einzuschränken. Es müssen nicht alle Features des Konstituenten spezifiziert werden. Unspezifizierte Features resultieren darin, dass das Tool für alle möglichen Werte des Features eine flektierte Form sucht. Es wird jedoch nur die erste Form berücksichtigt bei der Anzeige.

Der Konstituent und die Featureliste werden eins zu eins aus dem Template genommen und an das Flektionstool gesendet. Es sollten also Namen und Werte sein, mit denen das Tool etwas anfangen kann.

Als Beispiel hätte

```
<CONS GRAPH PHON> LEX_V_G (PERS = S1, TEMP = PRAET, MOD = IND) </CONS>
```

den Effekt, dass anhand eines deutschen Verbstammes die Präteritumform der ersten Person Singular im Indikativ generiert wird. Der `<CONS>`-Tag wird dabei in der HTML-Datei mit den graphemischen und phonetischen Formen des so entstandenen Ausdrucks ersetzt.

Kapitel 7

Usability

7.1 Limitationen des Java Lexicon Editors

Natürlich war es nicht möglich, in der vorgegebenen Zeit, ein komplettes und fehlerfreies Programm zu entwickeln, das auf dem Weltmarkt sofort reissenden Absatz findet. Zugegebenermassen hat der Java Lexicon Editor einige Schwachstellen, auf die wir zwar nicht stolz sind, die aber dennoch nicht verborgen werden sollten. Folgende Liste soll kurz auf die Limitationen zu sprechen kommen:

- *Plattformunabhängigkeit* – Wie der Name schon sagt, wurde der Java Lexicon Editor in der Programmiersprache Java geschrieben. Die fertige Applikation funktioniert demnach auf allen Betriebsystemen, für welche es eine Java Virtual Machine gibt. So läuft alles was mit der Manipulation der Lexika zu tun hat einwandfrei auf Betriebssystemen wie z.B. UNIX, MacOS und Windows. Die Flektionsgenerierung hingegen ist abhängig vom Flektionstool, das gegenwärtig nur unter UNIX funktioniert. Dies hat zur Folge, dass die Funktionen des Lexicon Editors nur auf einem einzigen System — UNIX — zu hundert Prozent verfügbar sind.

- *Gültigkeit von Tabellenwerten* – Die Überprüfung von Werten in der Tabelle funktioniert nur für Featurewerte, die als solche vorgegeben sind, also nur für Features die ihre möglichen Werte in folgender Form definieren:

  ```
  [Feature] Name "Description"
     [ Value "ValueDescr" Value "ValueDescr" ... ]
  [End Feature]
  ```

  Bei Features, welchen in der Description-Datei eine festcodierte Menge von möglichen Werten zugeordnet werden, kann ausnahmslos korrekt entschieden werden, ob der eingegebene Wert gültig ist (sprich: in dieser Menge enthalten ist).
  Jedoch stellen solche Features ein besonderes Problem dar, welche ihre möglichen Werte als beliebige Folge von atomaren Zeichenketten definieren. Features der Form

  ```
  [Feature] Name "Description"
     { Token Token ... }
  [End Feature]
  ```

sind wesentlich schwieriger nach deren Gültigkeit zu testen, weil nicht mehr einfach eine Menge von Werten durchsucht werden kann. Es ist ein komplizierter Algorithmus notwendig um dieses Problem zu lösen, der bei einer grossen Menge atomarer Zeichenketten beliebig lange rechnen müsste um ein Urteil zu fällen. Da wir vor allem auf Flexibilität achten wollten, und der Nachteil keinesfalls ins Gewicht fällt, eine derartige Überprüfung nicht anbieten zu können, haben wir uns entschieden, das Problem zu ignorieren.

- *Flektionsrückgabe* – Obwohl das Flektionstool unter Umständen mehr als nur eine Form an der Lexicon Editor retourniert, wird nur der erste erhaltene Wert ins HTML-Template übernommen. Natürlich kann dieser Nachteil durch genauere Spezifikation der Features und mehreren `<CONS>`-Tags umgangen werden, die gleichzeitige Anzeige einer Menge von Rückgabewerten wäre aber bestimmt um einiges angenehmer.

- *Lexikonformate* – Im derzeitigen Zustand ist es dem Lexicon Editor nur möglich, Dateien mit dem Format der Masterlexika zu bearbeiten. Weil nur eine Description-Datei geladen wird, die wiederum nur ein einziges Dateiformat beschreibt, widerspricht dies ein wenig unserem Streben nach Flexibilität und Allgemeinheit.

- *Sprachausgabe* – Da der Java Lexicon Editor für das SVOX Sprachsynthese-System konzipiert ist, wäre eine Funktion naheliegend, die es erlaubt einzelne Lexikoneinträge und Flektionen über ein Lautsprecher aussprechen zu lassen. Da dies jedoch kaum praktischen Nutzen hat, musste diese Funktion leider vergebens auf ihre Implementierung warten.

- *Phonetische Transkription* – Die automatische Umschrift von graphemischer nach phonetischer Form funktioniert nur zusammen mit dem `lexedit` Tool, da beim Empfang der phonetischer Form auf einen speziellen Prompt gewartet wird, bis aufgehört wird vom Stream zu lesen. Änderungen am Sourcecode können diese Einschränkung leicht umgehen.

7.2 Usability-Tests

Die Brauchbarkeit des Lexikon Editors zum Erstellen eines neuen Lexikons, Ändern und Erweitern bestehender Lexika ist gewährleistet. Die absolute Korrektheit ist zwar nicht bewiesen, da während der Lexikonverwaltung jedoch nie Fehler auftraten, gehen wir davon aus, dass das Programm diesbezüglich korrekt ist. Während der Entwicklung des Programmes haben wir zu Test- und Debbugingzwecken unzählige Male Tabellenwerte manipuliert, Zeilen gelöscht, eingefügt und dupliziert, Lexika geöffnet und abgespeichert, ohne dass dabei jemals unerwartete Seiteneffekte aufgetaucht sind.
Die Brauchbarkeit der Flektionsgenerierung ist für uns schwieriger zu beurteilen, da wir uns mit der deutschen Grammatik zu wenig auskennen. Es wären genauere Kenntnisse der Bedeutungen der einzelnen Featurewerte nötig, um die generierten Flektionen als Hilfe zur Korrektur falscher Einträge verwenden zu können.

7.3 Bekannte Bugs

- Zufällige unvollständige oder leere Rückgabe der Flektionen. Ursache konnte nicht genau eruiert werden.
- Unschönheiten beim Ein/Ausblenden der unteren Teilfenster.
- Auto-Complete funktioniert noch nicht 100-prozentig zuverlässig und funktioniert gar nicht, wenn sich der Eingabecursor nicht in der Zelle befindet.
- Einige Exceptions in Zusammenhang mit Auto-Complete, die aber nicht weiter stören.

Kapitel 8

Fazit

8.1 Aufgetretene Probleme

- Es ist zwar schwer einzusehen, aber der mit Abstand arbeitsintensivste Teil der Arbeit war die Aufgabe, zu erreichen, dass sich die Filterfelder in Grösse und Position korrekt anordnen, wenn Tabellenspalten umgeordnet werden. Das Problem konnte mit Hilfe von Antworten aus Newsgroups gelöst werden.

- Bei der Kommunikation mit dem LexTool tauchte die Frage auf, wie man erkennen kann, ob alle Rückgabewerte aus dem Inputstream vollständig gelesen worden sind. Wie schon erwähnt, verlässt sich der Lexicon Editor nun auf das Erscheinen eines Prompts, der am Ende jeder Ausgabe des LexTools erscheint, um das "Ende" des Inputstreams zu erkennen.

- Ein weiteres Problem in Zusammenhang mit der Flektionsgenerierung war, dass trotz der obigen Lösung mit dem Prompt manchmal merkwürdige oder leere Antworten vom Flektionstool eintrafen. Wir konnten diese Effekte verringern, indem wir nach jedem Senden eines Kommandos und vor dem Empfangen eine gewisse Zeit lang gewartet haben, dadurch dauerte aber die ganze Flektion ziemlich lange. Dieses Timing-Problem haben wir bis am Schluss nicht ganz in den Griff bekommen, aber die Flektion funktioniert soweit recht zuverlässig und in vertretbarer Zeit.

8.2 Schlussfolgerungen

Der Java Lexicon Editor ist ein einsetzbares und funktionstüchtiges Werkzeug geworden. Die Verwaltung der Lexika ist bis auf die Behandlung mehrerer Lexikonformate komplett. Die Möglichkeit zur effizienten Überprüfung der Lexika nach Fehlerstellen ist uns genauso gelungen wie die Bereitstellung einer Fülle von Editierhilfsmitteln, welche den Lexicon Editor zu einem wirksamen und einfach bedienbaren Programm machen. Natürlich war es uns aber dennoch nicht möglich all unsere Ideen zu verwirklichen und im Programm einzubauen. Gedanken und Vorschläge für allfällige Folgearbeiten werden im nächsten Abschnitt kurz erläutert.

8.3 Verbesserungsmöglichkeiten und weitere denkbare Funktionen

Sicherlich wären als Erweiterungen des Java Lexicon Editors die bereits im Kapitel 7.1 diskutierten Punkte denkbar:

- Erreichen einer besseren Plattformunabhängigkeit.
- Überprüfung der Gültigkeit aller Features.
- Berücksichtigung aller retournierten Flektionen in den HTML-Templates.
- Erkennen von mehr als nur einem einzigen Lexikonformat.
- Fähigkeit zur Sprachausgabe angewählter Lexikoneinträge und Flektionen.
- Verbesserung der Zuverlässigkeit und Geschwindigkeit der Flektionsgenerierung.

Ausserdem wären auch folgende Zusatzfunktionen eine Implementation wert:

- Möglichkeit zur Editierung der Preference-Datei im Lexicon Editor selbst.
- Kontextsensitives Ersetzen/Umbenennen von Featurewerten.
- Funktionen zum Kopieren, Ausschneiden und Einfügen ganzer Tabellenzeilen, nicht bloss der Werte in einzelnen Zellen.

Literaturverzeichnis

[1] B. Pfister, H.-P. Hutter. *Sprachverarbeitung I.* Vorlesungsskript für das Wintersemester 2001/2002, Departement ITET, ETH Zürich, 2001.

[2] B. Pfister, H.-P. Hutter und C. Traber. *Sprachverarbeitung II.* Vorlesungsskript für das Sommersemester 2002, Departement ITET, ETH Zürich, 2002.

[3] H. Traber. *SVOX: The Implementation of a Text-to-Speech System for German.* PhD thesis, No. 11064, Computer Engineering and Networks Laboratory, ETH Zurich (TIK-Schriftenreihe Nr. 7, ISBN 3-7281-2239-4), March 1995.

[4] David Flanagan. *Java in a nutshell*, Deutsche Ausgabe für Java 1.2 und 1.3, O'Reilly Verlag 2000, ISBN 3-89721-190-4.

[5] David Flanagan. *Java Foundation Classes in a nutshell*, Deutsche Ausgabe für Java 1.2, O'Reilly Verlag 2000, ISBN 3-89721-191-2.

Anhang A

Aufgabenstellung

Institut für Technische Informatik und Kommunikationsnetze
ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich Ecole polytechnique fédérale de Zurich Politecnico federale di Zurigo
Gruppe für Sprachverarbeitung

Sommersemester 2002/03

SEMESTERARBEIT

für

Herrn Dominik Kaspar
Herrn Matthias Wille

Betreuer: H. Romsdorfer ETZ D97.5
Stellvertreter: R. Beutler ETZ D97.7

Ausgabe: 22. Oktober 2002
Abgabe: 7. Februar 2003

Java-basierter Lexikon Editor

Einleitung

Das in der Gruppe für Sprachverarbeitung entwickelte Sprachsynthesesystem SVOX erlaubt es, geschriebene Sätze vom Computer aussprechen zu lassen. In einem ersten Verarbeitungsschritt werden diese Sätze vom Sprachsynthesesystem morphologisch und syntaktisch analysiert. Für diese Analysen werden sowohl ein Vollformenlexikon als auch ein Morphemlexikon[1] verwendet (siehe [1], Kap. 3). Beide Lexika enthalten neben Einträgen in graphemischer und phonetischer Form auch die zugehörige grammatikalische Information unter Verwendung des DCG-Formalismus[2] .

Da die zum heutigen Zeitpunkt in elektronischer Form verfügbaren Lexika nicht die für das Sprachsynthesesystem SVOX notwendige grammatikalische Information und zum Teil nicht einmal die phonetische Transkription enthalten, ist es notwendig, die benötigten Lexika in aufwendiger Kleinarbeit per Hand zu erstellen. Der Lexikon-Editor soll dabei helfen, diesen manuellen Editiervorgang erheblich zu beschleunigen und die Editierfehlerhäufigkeit zu senken.

[1] Als Morphem bezeichnet man den kleinsten bedeutungstragenden Teil eines Wortes.

[2] Der Definite Clause Grammar (DCG) - Formalismus dient zur Beschreibung kontextsensitiver Sprachen unter Verwendung eines kontextfreien SSkelettesmit zusätzlichen einschränkenden Bedingungen (vergl. [2], Kap. 7.4).

Problemstellung

Die Aufgaben eines solchen Lexikon-Editors umfassen vielfältige Bereiche. Diese beinhalten:

- die Verwaltung der Lexika (Einlesen und Speichern unterschiedlicher Lexikon-Formate, Versionskontrolle, Benutzer-Identifizierung, ...),
- das Testen der Lexika (Verwaltung der Testkorpora, Identifizierung und Anzeige der Fehlerstellen, ...),
- Editierhilfsmitteln (automatisierte Ermittlung der phonetischen Transkription, Sprachausgabe, Flexionsgenerierung, kontextabhängige Eingabefelder, ...).

In dieser Studien-Arbeit geht es nun darum, die genauen Anforderungen an einen Lexikon-Editor im Sinne eines Pflichtenheftes in enger Zusammenarbeit mit den Betreuern zu ermitteln, ausgehend von diesen Anforderungen einen entsprechenden Design-Vorschlag (GUI-Design und Software-Design) zu entwerfen, und abschliessend diesen Entwurf in der Programmiersprache Java zu implementieren. Bei der Implementation kann auf bereits vorhandene Funktionen zur phonetischen Transkription, Sprachausgabe und Flexionsgeneration zurückgegriffen werden.

Aufgaben

In Rahmen dieser Studien-Arbeit sind die folgenden Aufgaben zu lösen:

1. Einarbeitung in den DCG-Formalismus und das Lexikon-Format von SVOX anhand folgender Literatur ([1], [2]).
2. Zusammenstellen eines Anforderungsprofils (Pflichtenheft) für einen Lexikon-Editor. Dieses Anforderungsprofil muss vor der Implementierung mit den Betreuern abgesprochen werden.
3. Erstellung eines Software- und GUI-Designs basierend auf diesem Anforderungsprofil.
4. Implementation dieses Designs mittels der Programmiersprache Java. Eine entsprechende Kenntnis von Java wird vorausgesetzt.
5. Durchführen von Usability-Tests, um die Brauchbarkeit des Lexikon-Editors einerseits zum Erstellen eines neuen Lexikons, anderseits zum Erweitern bzw. Ändern eines bestehenden Lexikons zu testen.
6. Die ausgeführten Arbeiten und die erhaltenen Resultate sind in einem Bericht zu dokumentieren, der in zwei Exemplaren abzugeben ist, wovon eines Eigentum des Instituts bleibt.

Literaturverzeichnis

[1] C. Traber. *SVOX: The Implementation of a Text-to-Speech System for German.* PhD thesis, No. 11064, Computer Engineering and Networks Laboratory, ETH Zurich (TIK-Schriftenreihe Nr. 7, ISBN 3 7281 2239 4), March 1995.

[2] B. Pfister, H.-P. Hutter und C. Traber. *Sprachverarbeitung II.* Vorlesungsskript für das Sommersemester 2002, Departement ITET, ETH Zürich, 2002.

Zürich, den 15. Oktober 2002 Prof. Dr. L. Thiele

Anhang B

Klassendokumentation

Auf den folgenden Seiten ist ein Überblick über die API der Java-Klassen des Lexicon Editors zu finden, die wir aus den Source-Files automatisch mit Hilfe des `javadoc` Tools erstellt haben.

Die vollständige javadoc Klassendokumentation in HTML ist auf der beiliegenden CD-ROM zu finden.

B.1 Paket ethz.svoxlexedit

Package ethz.svoxlexedit

Class Summary	
AbstractLexTableModel	Provides an abstract common superclass for all table models which are responsible for manipulating and storing lexicon data and displaying them in a `LexTable`, like filters and sorters.
Category	`Category` class objects represent the category of a lexicon entry and store all the links to the features belonging to the category.
ConsElement	`ConsElement` class object represent an entire CONS tag by analysing a CONS String and storing all information into better accessible data structures.
Feature	`Feature` class objects represent the features and its valid values. **Copyright:** Copyright (c) 2002 **Organisation:** ETH Zurich
FeatureValue	A helper class which represents a valid feature value represented by its name and a description. **Copyright:** Copyright (c) 2002 **Organisation:** ETH Zurich
HistoryEntry	`HistoryEntry` class objects are a special case of normal lexicon entries.
HistoryTableModel	This class holds the data for the history table.
LexFilterField	This class represents a filter textfield which is used together with a `LexTableFilter` and enables users to filter a table with respect to some regular expressions over columns of the table.
LexFormat	`LexFormat` is the main class of handling the lexicon format.
LexHTMLViewer	`LexHTMLViewer` is responsible for displaying HTML files in the flection subwindow.
Lexicon	The class `Lexicon` covers all basic functions dealing with the manipulation of lexicons and its entries. **Copyright:** Copyright (c) 2002 **Organisation**: ETH Zurich
LexiconEditor	This is the main class of the Java Lexicon Editor for the SVOX speech synthesis system.
LexiconEntry	`LexiconEntry` represents a row of the lexicon table and implements all the basic functionality of handling direct table data manipulation.
LexTable	LexTable is a subclass of JTable which renders a Lexicon Model on the screen in a table.
LexTableCellEditor	This is a custom table cell editor used by `LexTable` to provide auto-completion for feature values.
LexTableCellRenderer	This is a custom table cell renderer used by `LexTable` to provide highlighting of invalid features in the table cells.
LexTableFilter	This class implements filter functionality for any other `AbstractLexTableModel` which is used as the underlying data source.
LexTableMap	This class is a basic implemenation of a `AbstractLexTableModel` and the superclass for both `LexTableFilter` and `LexTableSorter`.
LexTableModel	This Class provides the `LexTable` and all other AbsractLexTableModels with the actual entries of the table.

LexTableSorter	This class implements sorter functionality for any other `AbstractLexTableModel` which is used as the underlying data source.
LexTool	This static class represents the extern *Flection Tool which is used for flection generation.*
LoginWindow	This class represents the Login window which is shown before the main window opens.
MainWindow	This is the main window of the Lexicon Editor and a subclass of JFrame.
Preferences	`Preferences` is the class for storing all the user and environment settings.

B.2 Abstrakte Klasse AbstractLexTableModel

ethz.svoxlexedit

Class AbstractLexTableModel

```
java.lang.Object
  |
  +--javax.swing.table.AbstractTableModel
        |
        +--ethz.svoxlexedit.AbstractLexTableModel
```

All Implemented Interfaces:
java.io.Serializable, javax.swing.table.TableModel

Direct Known Subclasses:
LexTableMap, LexTableModel

---

abstract class **AbstractLexTableModel**
extends javax.swing.table.AbstractTableModel

Provides an abstract common superclass for all table models which are responsible for manipulating and storing lexicon data and displaying them in a `LexTable`, like filters and sorters.

Provides some additional methods to the ones of `AbstractTableModel`.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`LexTable`, `LexTableFilter`, `LexTableSorter`, `LexTableModel`, `LexTableMap`, `JTable`, `AbstractTableModel`, Serialized Form

Constructor Summary

(package private)	**AbstractLexTableModel**()

Method Summary	
abstract LexiconEntry	**getEntryAt**(int row) Returns the lexicon entry specified by `row` in the underlying `Lexicon` object.
abstract LexTableFilter	**getFilter**() Returns the `LexTableFilter` in the chain of table models.
abstract Lexicon	**getLexicon**() Returns the `Lexicon` which stores the actual entries.
int	**getRealIndex**(int index) Returns the possible mapped index of an lexicon entry based on the underlying tablemodel.

B.3 Klasse Category

ethz.svoxlexedit

Class Category

```
java.lang.Object
  |
  +--ethz.svoxlexedit.Category
```

All Implemented Interfaces:
java.lang.Comparable

---

public class **Category**
extends java.lang.Object
implements java.lang.Comparable

`Category` class objects represent the category of a lexicon entry and store all the links to the features belonging to the category.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary	
java.lang.String	**desc** Description of the category name.
java.util.Vector	**featurelist** List of `Feature` objects for this `Category` (in the same order as in the lexicon file).
int[]	**featuretableorder** Mapping of the table model columns to the feature list.
java.lang.String	**name** Name of the word category.

Constructor Summary
Category() Default constructor of a `Category` object.
Category(java.lang.String n, java.lang.String d) Constructor specifying name of `Category` and description of `Category` name.

Method Summary	
int	**compareTo**(java.lang.Object o) Implementation of comparable interface.
Feature	**getFeatureAt**(int index) Fetches the `Feature` at a given index in the feature list.
java.lang.String	**getFeatureNameAt**(int index) Fetches the name of a feature at a given index in the feature list.
int	**getIndexOfFeature**(java.lang.String featname) Returns the index in the `featurelist` of this `category` for a feature name.
int	**getNumberOfFeatures**() Counts the number of features in this `Category`.
boolean	**hasFeature**(java.lang.String feature) Checks if the category contains the specified feature.
void	**initFeatureList**(java.lang.String list) Initialization of the `featurelist` with `Feature` objects.
void	**initFeatureTableOrderArray**(java.lang.Object[] common, int size) Sets the order of the feature columns in the table: common features first, then the special features and possibly some empty columns.

B.4 Klasse ConsElement

ethz.svoxlexedit

Class ConsElement

```
java.lang.Object
  |
  +--ethz.svoxlexedit.ConsElement
```

---

public class **ConsElement**
extends java.lang.Object

`ConsElement` class object represent an entire CONS tag by analysing a CONS String and storing all information into better accessible data structures.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary

java.lang.String	**constituent** Specifies the constituent of the CONS tag.
java.util.HashMap	**featurelist** Stores all the features with its values of the CONS tag.
boolean	**param_feat** Indicates whether there is a FEAT parameter in the CONS tag.
boolean	**param_graph** Indicates whether there is a GRAPH parameter in the CONS tag.
boolean	**param_phon** Indicates whether there is a PHON parameter in the CONS tag.
boolean	**param_type** Indicates whether there is a TYPE parameter in the CONS tag.

Constructor Summary

ConsElement(java.lang.String element) The constructor parses a CONS element and initializes the public fields with the information given in the CONS element.

Method Summary

java.lang.String[]	**getCommand**() Creates the flection tool specific commands by combining the information given from the parsed CONS element.

B.5 Klasse Feature

ethz.svoxlexedit

Class Feature

```
java.lang.Object
  |
  +--ethz.svoxlexedit.Feature
```

---

public class **Feature**
extends java.lang.Object

`Feature` class objects represent the features and its valid values.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary

java.lang.String	**desc** Description of the meaning of the feature name.
char	**listtype** Type of value list parsed, either '[' or '{'.
java.lang.String	**name** Name of the `Feature`.
java.util.HashMap	**validValues** All the allowed values for this `Feature`.

Constructor Summary

Feature() Default constructor of a `Feature` object.
Feature(java.lang.String n, java.lang.String d) Creates a `Feature` object from a feature name and its description.

Method Summary	
java.lang.String	**autoComplete**(java.lang.String value) Creates an auto-complete suggestion for part of a feature.
java.lang.Object[]	**getValidValues**() Returns an array of valid value strings for this `Feature`.
java.lang.String	**getValueDescription**(java.lang.String value) Reads the description of a `Feature`.
void	**initValueList**(java.lang.String line) Builds the list of `validValues`.
boolean	**isValueValid**(java.lang.String value) Checks if a given string is a valid value for this `Feature`.

B.6 Klasse FeatureValue

ethz.svoxlexedit

Class FeatureValue

```
java.lang.Object
  |
  +--ethz.svoxlexedit.FeatureValue
```

---

public class **FeatureValue**
extends java.lang.Object

A helper class which represents a valid feature value represented by its name and a description.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`Feature`

Field Summary	
`java.lang.String`	**`desc`** Descriptive comment about the `Feature` name.
`java.lang.String`	**`name`** Name of the `Feature`

Constructor Summary
`FeatureValue``()` Default constructor of a `FeatureValue` object.

B.7 Klasse HistoryEntry

ethz.svoxlexedit

Class HistoryEntry

```
java.lang.Object
  |
  +--ethz.svoxlexedit.LexiconEntry
        |
        +--ethz.svoxlexedit.HistoryEntry
```

All Implemented Interfaces:
java.lang.Comparable

---

public class **HistoryEntry**
extends LexiconEntry

`HistoryEntry` class objects are a special case of normal lexicon entries. As changes occur to lexicon entries they are tracked by creating a history entry, which carries some additional information such as the date when the changes occured.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary

`java.util.Date`	**`creationDate`** Time when changes to this entry occured.

Fields inherited from class ethz.svoxlexedit.LexiconEntry
`commandsArray, features, history, isModified, isValid`

Constructor Summary

`HistoryEntry``()` Default constructor of an empty `HistoryEntry` object.
`HistoryEntry``(LexiconEntry source)` Creates a history entry from a regular entry by copying its features.
`HistoryEntry``(java.lang.String line)` Creates a history entry from a simple `String`.

Method Summary	
void	**initWithString**(java.lang.String line) Initializes a history entry from a `String`.
java.lang.String	**toHistoryString**() Converts a `HistoryEntry` to a `String` representation for writing in to the lexicon files by adding additional history information.
java.lang.String	**toString**() Converts an entry to a `String` representation for writing in to the lexicon files.

Methods inherited from class ethz.svoxlexedit.LexiconEntry
buildCommands, changeCategory, checkValidity, clone, compareTo, getCategory, getCategoryName, getComm setFeatureValueAt, toStringWithoutHistory

B.8 Klasse HistoryTableModel

ethz.svoxlexedit

Class HistoryTableModel

```
java.lang.Object
  |
  +--javax.swing.table.AbstractTableModel
        |
        +--ethz.svoxlexedit.HistoryTableModel
```

All Implemented Interfaces:
java.util.EventListener, javax.swing.event.ListSelectionListener, java.io.Serializable, javax.swing.table.TableModel

---

public class **HistoryTableModel**
extends javax.swing.table.AbstractTableModel
implements javax.swing.event.ListSelectionListener

This class holds the data for the history table. It holds all history entries of the currently selected entry in the lexicon table and a JTable can display them.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`AbstractTableModel`, Serialized Form

Constructor Summary

`HistoryTableModel``(LexTable table, Lexicon lex)` Constructs a new `HistoryTableModel` which observes `table` and `lex`.

Method Summary	
int	**getColumnCount**() Returns the number of columns to display in the table.
java.lang.String	**getColumnName**(int columnIndex) Returns the header for the column index `columnIndex` to be displayed in the table.
int	**getRowCount**() Returns the number of rows to display in the table.
java.lang.Object	**getValueAt**(int rowIndex, int columnIndex) Returns the value which should be displayed in the table at row `rowIndex` and column `columnIndex`.
void	**setHistory**(java.util.Vector newHistory) Sets the history to be displayed.
void	**valueChanged**(javax.swing.event.ListSelectionEvent e) Invoked when the row selection of the observed table changes.

B.9 Klasse LexFilterField

ethz.svoxlexedit

Class LexFilterField

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.text.JTextComponent
                          |
                          +--javax.swing.JTextField
                                |
                                +--ethz.svoxlexedit.LexFilterField
```

All Implemented Interfaces:
javax.accessibility.Accessible, java.awt.event.ActionListener, java.util.EventListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.beans.PropertyChangeListener, javax.swing.Scrollable, java.io.Serializable, javax.swing.SwingConstants, javax.swing.event.TableColumnModelListener

---

public class **LexFilterField**
extends javax.swing.JTextField
implements java.awt.event.ActionListener, java.beans.PropertyChangeListener, javax.swing.event.TableColumnModelListener

This class represents a filter textfield which is used together with a `LexTableFilter` and enables users to filter a table with respect to some regular expressions over columns of the table.

A `LexFilterField` is connected to a `LexTableFilter` to pass the user input to the filter, which then does the filtering. It is also connected to a specific column of the `LexTable` to keep track of width changes and to tell the filter which column should be filtered.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`LexTableFilter`, `JTextField`, `RE`, Serialized Form

Field Summary

int	**colmodelindex** The column index in the model of the column associated with the field.
int	**colviewindex** The view column index of the column associated with the field.
protected LexTableFilter	**filter** The `LexTableFilter` which the field sends its filter string to.
protected LexTable	**table** The `LexTable` which the fields belong to.

Constructor Summary

LexFilterField(LexTable table, int col)
Constructs a new `LexFilterField` with `table` as associated `LexTable` for filtering over column index `col`.

Method Summary

void	**actionPerformed**(java.awt.event.ActionEvent e) Invoked when the user presses return in the field.
void	**clear**() Clears the content of the field.
void	**columnAdded**(javax.swing.event.TableColumnModelEvent e) Invoked when a column is added in the associated table.
void	**columnMarginChanged**(javax.swing.event.ChangeEvent e) Invoked when the margins of the columns in the associated table changed.
void	**columnMoved**(javax.swing.event.TableColumnModelEvent e) Invoked when columns are move in the associated table.
void	**columnRemoved**(javax.swing.event.TableColumnModelEvent e) Invoked when columns are removed in the associated table.
void	**columnSelectionChanged**(javax.swing.event.ListSelectionEvent e) Invoked when column selection changed in the associated table.
void	**propertyChange**(java.beans.PropertyChangeEvent e) Invoked when a property of the associated column has changed.

B.10 Klasse LexFormat

ethz.svoxlexedit

Class LexFormat

```
java.lang.Object
  |
  +--ethz.svoxlexedit.LexFormat
```

---

public class **LexFormat**
extends java.lang.Object

`LexFormat` is the main class of handling the lexicon format. It it responsible for reading the description file and building up the datastructure for the categories and features.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary	
static java.util.HashMap	**categories** Directory holding all the `Category` objects.
static java.lang.Object[]	**common** Array holding the features that exist in all different categories.
static java.util.HashMap	**features** Directory of all `Feature` objects.
static int	**maxColumns** Maximum number of features in all categories.

Constructor Summary
LexFormat() Default constructor for a `LexFormat` object.

Method Summary	
static Category	**getCategory**(java.lang.String name) Get a `Category` by its name.
static Feature	**getFeature**(java.lang.String name) Get a `Feature` by its name.
static boolean	**isCommonFeature**(java.lang.String name) Find out whether a feature exists in all categories or not.
static void	**readFromFile**(java.io.File formatfile) Reads in a description file.

B.11 Klasse LexHTMLViewer

ethz.svoxlexedit

Class LexHTMLViewer

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.text.JTextComponent
                          |
                          +--javax.swing.JEditorPane
                                |
                                +--ethz.svoxlexedit.LexHTMLViewer
```

All Implemented Interfaces:
javax.accessibility.Accessible, java.util.EventListener, javax.swing.event.HyperlinkListener, java.awt.image.ImageObserver, javax.swing.event.ListSelectionListener, java.awt.MenuContainer, javax.swing.Scrollable, java.io.Serializable

---

public class **LexHTMLViewer**
extends javax.swing.JEditorPane
implements javax.swing.event.ListSelectionListener, javax.swing.event.HyperlinkListener

`LexHTMLViewer` is responsible for displaying HTML files in the flection subwindow. It also initiates the communication to the flection tool and handles the replacement of the CONS Tags.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
Serialized Form

Field Summary	
boolean	**enabled** Whether the communication with the FlexTool should take place or not.

Constructor Summary

LexHTMLViewer() Default LexHTMLViewer constructor.
LexHTMLViewer(LexTable table) Constructor specifying the table containing the entries to flect.

Method Summary

void	**display**(java.lang.String filename) Displays a filename as an HTML document.
void	**hyperlinkUpdate**(javax.swing.event.HyperlinkEvent e) Follows an HTML link, when it is clicked on.
java.lang.String	**process**(java.lang.String oldcode) Parses HTML code and replaces the special CONS tags by the corresponding flected expressions.
void	**updateDisplay**() Refreshes the HTML file display.
void	**valueChanged**(javax.swing.event.ListSelectionEvent e) Display the correct HTML file once new lexicon entry gets focussed.

B.12 Klasse Lexicon

ethz.svoxlexedit

Class Lexicon

```
java.lang.Object
  |
  +--ethz.svoxlexedit.Lexicon
```

---

public class **Lexicon**
extends java.lang.Object

The class `Lexicon` covers all basic functions dealing with the manipulation of lexicons and its entries.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Constructor Summary
Lexicon() Default constructor creating an empty lexicon.
Lexicon(java.io.File lexfile) Constructor that initializes a lexicon from a file.
Lexicon(java.lang.String filename) Constructor that initializes a lexicon from a file.

Method Summary	
void	**addEntry**(LexiconEntry entry) Adds a new entry to the lexicon.
void	**addEntryAt**(LexiconEntry entry, int index) Adds a new entry at a given position to the lexicon.
int	**countEntries**() Determines the number of entries in the lexicon.
void	**deleteAllEntries**() Delete all entries from the lexicon.
void	**deleteEntryAt**(int index) Removes an entry from the lexicon.
void	**fireLexiconChange**() Inform the table model that data changes have occured.
LexiconEntry	**getEntryAt**(int index) Fetches a lexicon entry by a given index.
boolean	**isDirty**() Find out whether the lexicon has ever been edited since it was loaded.
void	**newEntryAt**(int insertposition, java.lang.String catname) Inserts an empty entry into the lexicon.
void	**readFile**(java.io.File lexfile) Reads a lexicon from a file.
void	**readFile**(java.lang.String fileName) Reads a lexicon from a file.
void	**setDirty**(boolean val) Set the dirty flag (a change in the lexicon has occurred since it was loaded).
void	**setParent**(LexTableModel p) Set the parent table model.
void	**sort**() Sort the vector containing all the lexicon entries.
void	**writeToFile**(java.io.File lexfile) Stores the vector containing the lexicon entries including all the history items to a text file.
void	**writeToFileWithoutHistory**(java.io.File lexfile) Stores the vector containing the lexicon entries to a text file, ignoring all the history items.

B.13 Klasse LexiconEditor

ethz.svoxlexedit

Class LexiconEditor

```
java.lang.Object
  |
  +--ethz.svoxlexedit.LexiconEditor
```

---

public class **LexiconEditor**
extends java.lang.Object

This is the main class of the Java Lexicon Editor for the SVOX speech synthesis system.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary

static java.lang.String	**EOL** Constant specifying the line seperator to use.
static java.lang.String	**MASTER_DESC_FILENAME** Constant specifying the name of the descrption file.

Constructor Summary

LexiconEditor() Default `LexiconEditor` constructor.

Method Summary

static void	**main**(java.lang.String[] args) The Main Method of the Java Lexicon Editor.

B.14 Klasse LexiconEntry

ethz.svoxlexedit

Class LexiconEntry

```
java.lang.Object
  |
  +--ethz.svoxlexedit.LexiconEntry
```

All Implemented Interfaces:
java.lang.Comparable

Direct Known Subclasses:
HistoryEntry

---

public class **LexiconEntry**
extends java.lang.Object
implements java.lang.Comparable

`LexiconEntry` represents a row of the lexicon table and implements all the basic functionality of handling direct table data manipulation.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary

java.lang.String[]	**commandsArray** Stores the commands for the flection tool (LexTool).
java.util.Vector	**features** Vector containing all features of an entry as strings.
java.util.Vector	**history** Vector containing the history as other lexicon entries.
boolean	**isModified** `True` if the entry has been edited, else `False`.
boolean	**isValid** `True` if all its values are valid, else `False`.

Constructor Summary

LexiconEntry() Default `LexiconEntry` constructor.
LexiconEntry(java.lang.String line) Constructor that initializes a `LexiconEntry` from a line out of a lexicon text file.

Method Summary	
void	**buildCommands**() Builds the command strings which are handed to the LexTool for setting it according to this entry.
void	**changeCategory**(java.lang.String catname) Change the category of a `LexiconEntry`.
void	**checkValidity**() Set the `isValid` flag if the value of all features is valid.
java.lang.Object	**clone**() Duplicate this `LexiconEntry` with all its feature values.
int	**compareTo**(java.lang.Object o) Implementation of comparable interface.
Category	**getCategory**() Fetches the category of a lexicon entry.
java.lang.String	**getCategoryName**() Fetches the category of a lexicon entry.
java.lang.String[]	**getCommands**() Get the commands that are to be handed to the LexTool.
java.lang.String	**getFeatureValue**(java.lang.String feature) Fetches the value of a feature by its name.
java.lang.String	**getFeatureValueAt**(int index) Fetches the value of a feature by its index.
java.lang.String	**getGraphemicForm**() Fetches the graphemic form of a lexicon entry.
int	**getNumberOfValues**() Determines the number of values in the lexicon entry.
java.lang.String	**getPhoneticForm**() Fetches the phonetic form of a lexicon entry.
void	**initWithString**(java.lang.String line) Initializes an empty `LexiconEntry` by parsing a `String` for features.
void	**setCategory**(java.lang.String catname) Set the category feature to a specific value.
void	**setFeatureValue**(java.lang.String feature, java.lang.String value) Writes the value ot a feature given its name.
void	**setFeatureValueAt**(int index, java.lang.String value) Writes the value of a feature given its index.
java.lang.String	**toString**() Converts the `LexiconEntry` to a `String` form.
java.lang.String	**toStringWithoutHistory**() Converts the `LexiconEntry` to a `String` form, ignoring all history entries belonging to it.

B.15 Klasse LexTable

ethz.svoxlexedit

Class LexTable

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JTable
                          |
                          +--ethz.svoxlexedit.LexTable
```

All Implemented Interfaces:
javax.accessibility.Accessible, javax.swing.event.CellEditorListener, java.util.EventListener, java.awt.image.ImageObserver, javax.swing.event.ListSelectionListener, java.awt.MenuContainer, javax.swing.Scrollable, java.io.Serializable, javax.swing.event.TableColumnModelListener, javax.swing.event.TableModelListener

---

public class **LexTable**
extends javax.swing.JTable
implements javax.swing.event.ListSelectionListener

LexTable is a subclass of JTable which renders a Lexicon Model on the screen in a table.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`JTable`, Serialized Form

Constructor Summary

`LexTable``()`
Constructs a new LexTable.

Method Summary	
void	**adjustColumns**() Sets the order and width of the columns according to the preferences.
boolean	**cellIsValid**(int row, int viewcol) Check if the table cell given by `row` and `viewcol` contains a valid value.
LexiconEntry	**getEntryAt**(int row) Returns the lexicon entry at the specifig table row index.
Feature	**getFeatureAtCell**(int row, int viewcol) Returns the Feature which is displayed in the cell given by `row` and `viewcol`.
LexTableFilter	**getFilter**() Returns the filter Object used by the LexTable.
int	**getIndexInLexicon**(int tablerow) Returns the real index of a table row in the lexicon.
void	**updateColumnHeaders**() Changes the table headers to match with the category of the selected entry in the table.
void	**valueChanged**(javax.swing.event.ListSelectionEvent e) Invoked when the row selection changes.

B.16 Klasse LexTableCellEditor

ethz.svoxlexedit

Class LexTableCellEditor

```
java.lang.Object
  |
  +--javax.swing.AbstractCellEditor
        |
        +--javax.swing.DefaultCellEditor
              |
              +--ethz.svoxlexedit.LexTableCellEditor
```

All Implemented Interfaces:
javax.swing.CellEditor, java.io.Serializable, javax.swing.table.TableCellEditor, javax.swing.tree.TreeCellEditor

---

public class **LexTableCellEditor**
extends javax.swing.DefaultCellEditor

This is a custom table cell editor used by `LexTable` to provide auto-completion for feature values.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`DefaultCellEditor`, Serialized Form

Constructor Summary

`LexTableCellEditor``()` Constructs a new `LexTableCellEditor` with a `JComboBox` as editor component as a default.
`LexTableCellEditor``(javax.swing.JCheckBox checkBox)` Constructs a new `LexTableCellEditor` with a `JCheckBox` as editor component.
`LexTableCellEditor``(javax.swing.JComboBox comboBox)` Constructs a new `LexTableCellEditor` with a `JComboBox` as editor component.
`LexTableCellEditor``(javax.swing.JTextField textField)` Constructs a new `LexTableCellEditor` with a `JTextField` as editor component.

Method Summary

`java.awt.Component`	**`getTableCellEditorComponent`**`(javax.swing.JTable table, java.lang.Object value, boolean isSelected, int row, int column)` Returns the `Component` which is used by the table to edit `value` in row `row` and column `column`.

B.17 Klasse LexTableCellRenderer

ethz.svoxlexedit

Class LexTableCellRenderer

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
                    |
                    +--javax.swing.JLabel
                          |
                          +--javax.swing.table.DefaultTableCellRenderer
                                |
                                +--ethz.svoxlexedit.LexTableCellRenderer
```

All Implemented Interfaces:
javax.accessibility.Accessible, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, javax.swing.SwingConstants, javax.swing.table.TableCellRenderer

---

public class **LexTableCellRenderer**
extends javax.swing.table.DefaultTableCellRenderer

This is a custom table cell renderer used by `LexTable` to provide highlighting of invalid features in the table cells.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`DefaultTableCellRenderer`, Serialized Form

Constructor Summary

`LexTableCellRenderer``()`

Method Summary

`java.awt.Component`	**`getTableCellRendererComponent`**`(javax.swing.JTable table, java.lang.Object value, boolean isSelected, boolean hasFocus, int row, int column)` Returns the `Component` which is be used by a table to display `value` in row `row` and column `column`.

B.18 Klasse LexTableFilter

ethz.svoxlexedit

Class LexTableFilter

```
java.lang.Object
  |
  +--javax.swing.table.AbstractTableModel
        |
        +--ethz.svoxlexedit.AbstractLexTableModel
              |
              +--ethz.svoxlexedit.LexTableMap
                    |
                    +--ethz.svoxlexedit.LexTableFilter
```

All Implemented Interfaces:
java.util.EventListener, java.io.Serializable, javax.swing.table.TableModel, javax.swing.event.TableModelListener

---

public class **LexTableFilter**
extends LexTableMap

This class implements filter functionality for any other `AbstractLexTableModel` which is used as the underlying data source.

This table model doesn't store any data, it only stores a mapping of indices to an underlying table model. A reference to the underlying model is inherited from its superclass `LexTableMap`. Any requests to this model are redirected via the mapping array to the underlying model.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`AbstractLexTableModel`, `LexTableMap`, `LexTableFilter`, `LexTableModel`, Serialized Form

Fields inherited from class ethz.svoxlexedit.LexTableMap
`model`

Constructor Summary

`LexTableFilter``()` Constructs a new `LexTableFilter` object without any underlying datmodel.
`LexTableFilter``(AbstractLexTableModel model)` Constructs a new `LexTableFilter` object with `model` as its underlying data model which will be filtered.

Method Summary	
void	**addExpression**(java.lang.String pattern, int colindex) Adds a Filter expression `pattern` for column `colindex` to the current filter settings.
void	**addTextFieldforColumn**(javax.swing.JTextField textfield, int col) Registers a JTextField `textfield` for column `col` to accept filter strings from the field.
LexiconEntry	**getEntryAt**(int row) Returns the `LexiconEntry` at index `row`.
LexTableFilter	**getFilter**() Overriding of Method getFilter() from LexTableMap which just return a reference to self.
int	**getRealIndex**(int index) Returns the real index of row `index`, that is the index in the underlying Lexicon.
int	**getRowCount**() Returns the number of rows to be displayed.
java.lang.Object	**getValueAt**(int aRow, int aColumn) Returns the object value at row `aRow` and column `aColumn` of the underlying model.
void	**initFilter**() Inits the filter with a null filter: no filtering is performed and all entries of the underlying data model are visible.
boolean	**isCellEditable**(int row, int column) Returns true if the cell specified by `row` and `column` should be editable to the user.
void	**performFiltering**() Filters the underlying model with the current filter strings and update the internal index mapping accordingly.
void	**setValueAt**(java.lang.Object value, int aRow, int aColumn) Sets the value at row `aRow` and column `aColumn` of the underlying model to `value`.
void	**showInvalidEntries**() Filters out the valid entries in the underlying model and displays only the invalid entries of the underlying data model.
void	**tableChanged**(javax.swing.event.TableModelEvent e) Invoked when the underlying table model has changed somehow.

Methods inherited from class ethz.svoxlexedit.LexTableMap
getColumnClass, getColumnCount, getColumnName, getLexicon, getModel, setModel

B.19 Klasse LexTableMap

ethz.svoxlexedit

Class LexTableMap

```
java.lang.Object
  |
  +--javax.swing.table.AbstractTableModel
        |
        +--ethz.svoxlexedit.AbstractLexTableModel
              |
              +--ethz.svoxlexedit.LexTableMap
```

All Implemented Interfaces:
java.util.EventListener, java.io.Serializable, javax.swing.table.TableModel, javax.swing.event.TableModelListener

Direct Known Subclasses:
LexTableFilter, LexTableSorter

---

public class **LexTableMap**
extends AbstractLexTableModel
implements javax.swing.event.TableModelListener

This class is a basic implemenation of a `AbstractLexTableModel` and the superclass for both `LexTableFilter` and `LexTableSorter`. It stores a reference to an underlying `AbstractLexTableModel` and implements all its methods by just redirecting all requests to this underlying model. It can be regarded as a null filter or a null sorter and should have no effect if inserted in a chain of such table models.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`AbstractLexTableModel`, `LexTableMap`, `LexTableFilter`, `LexTableModel`, Serialized Form

Field Summary

protected AbstractLexTableModel	**model** The underlying model.

Constructor Summary

LexTableMap()

Method Summary	
java.lang.Class	**getColumnClass**(int aColumn) Returns the class of the values in column with index aColumns.
int	**getColumnCount**() Returns the number of columns in the underlying table model.
java.lang.String	**getColumnName**(int aColumn) Returns the name of the column with index aColumns.
LexiconEntry	**getEntryAt**(int row) Returns the lexicon entry specified by row in the underlying Lexicon object.
LexTableFilter	**getFilter**() Returns the LexTableFilter in the chain of table models.
Lexicon	**getLexicon**() Returns the Lexicon which stores the actual entries.
AbstractLexTableModel	**getModel**() Returns the currently used underlying table model.
int	**getRealIndex**(int index) Returns the index of an lexicon entry based on the underlying tablemodel.
int	**getRowCount**() Returns the number of rows in the underlying table model.
java.lang.Object	**getValueAt**(int aRow, int aColumn) Returns the object value at row aRow and column aColumn of the underlying model.
boolean	**isCellEditable**(int row, int column) Returns true if the cell specified by row and column should be editable to the user.
void	**setModel**(AbstractLexTableModel model) Sets the underlying table model to model.
void	**setValueAt**(java.lang.Object aValue, int aRow, int aColumn) Sets the value at row aRow and column aColumn of the underlying model to value.
void	**tableChanged**(javax.swing.event.TableModelEvent e) Invoked when the underlying table model has changed somehow.

B.20 Klasse LexTableModel

ethz.svoxlexedit

Class LexTableModel

```
java.lang.Object
  |
  +--javax.swing.table.AbstractTableModel
        |
        +--ethz.svoxlexedit.AbstractLexTableModel
              |
              +--ethz.svoxlexedit.LexTableModel
```

All Implemented Interfaces:
java.io.Serializable, javax.swing.table.TableModel

---

public class **LexTableModel**
extends AbstractLexTableModel

This Class provides the `LexTable` and all other AbsractLexTableModels with the actual entries of the table. It stores all entries of the table in a `Lexicon` class and provides methods of access to it for models which use the LexTableModel as their underlying model.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`AbstractLexTableModel`, `LexTableMap`, `LexTableFilter`, `LexTableModel`, `Lexicon`, `LexiconEntry`, Serialized Form

Constructor Summary

`LexTableModel``()` Constructs a new `LexTableModel` with an empty lexicon and no parent LexTable.
`LexTableModel``(Lexicon thatlex, LexTable table)` Constructs a new `LexTableModel` with `thatlex` as the lexicon and `table` as the parent LexTable.

Method Summary	
java.lang.Class	**getColumnClass**(int columnIndex) Returns the class of the values in column columnIndex.
int	**getColumnCount**() Return the number of columns in the table.
java.lang.String	**getColumnName**(int columnIndex) Returns the name for column columnIndex.
LexiconEntry	**getEntryAt**(int row) Returns the lexicon entry specified by row in the underlying Lexicon object.
LexTableFilter	**getFilter**() Returns the LexTableFilter in the chain of table models.
Lexicon	**getLexicon**() Returns the Lexicon which stores the actual entries.
int	**getRealIndex**(int index) Returns the index of an lexicon entry based on the underlying tablemodel.
int	**getRowCount**() Returns the number of rows in the table.
java.lang.Object	**getValueAt**(int rowIndex, int columnIndex) Returns the object value at row aRow and column aColumn of the table.
boolean	**isCellEditable**(int rowIndex, int columnIndex) Returns true if the cell specified by row and column should be editable to the user.
void	**setLexicon**(Lexicon lex) Sets the lexicon to lex and fires a TableDataChanged event to all listeners.
void	**setValueAt**(java.lang.Object aValue, int rowIndex, int columnIndex) Sets the value at row aRow and column aColumn of the underlying model to value.

B.21 Klasse LexTableSorter

ethz.svoxlexedit

Class LexTableSorter

```
java.lang.Object
  |
  +--javax.swing.table.AbstractTableModel
        |
        +--ethz.svoxlexedit.AbstractLexTableModel
              |
              +--ethz.svoxlexedit.LexTableMap
                    |
                    +--ethz.svoxlexedit.LexTableSorter
```

All Implemented Interfaces:
java.util.EventListener, java.io.Serializable, javax.swing.table.TableModel, javax.swing.event.TableModelListener

---

public class **LexTableSorter**
extends LexTableMap

This class implements sorter functionality for any other `AbstractLexTableModel` which is used as the underlying data source.

This table model doesn't store any data, it only stores a mapping of indices to an underlying table model. A reference to the underlying model is inherited from its superclass `LexTableMap`. Any requests to this model are redirected via the mapping array to the underlying model.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`AbstractLexTableModel`, `LexTableMap`, `LexTableFilter`, `LexTableModel`, Serialized Form

Field Summary

(package private) boolean	**ascending** A flag which indicates if sorting is ascending or descending
(package private) int	**compares**
(package private) int[]	**indexes** The index mapping array.
(package private) java.util.Vector	**sortingColumns** The columns used to perform the sorting

Fields inherited from class ethz.svoxlexedit.LexTableMap
model

Constructor Summary

LexTableSorter() Constructs a new `LexTableSorter` with an empty mapping array.
LexTableSorter(AbstractLexTableModel model) Constructs a new `LexTableSorter` object with `model` as its underlying data model which will be filtered.

Method Summary

void	**addMouseListenerToHeaderInTable**(javax.swing.JTable table) Registers the sorter with a JTable to allow the user to invoke the sorter when clicking in a column header.
void	**checkModel**() Checks if the length of the mapping array and the number of rows of the underlying model are equal.
int	**compare**(int row1, int row2) Compares two rows with respect to all current sorting columns and to the sorting order.
int	**compareRowsByColumn**(int row1, int row2, int column) Compares two rows of the model by a common column.
LexiconEntry	**getEntryAt**(int row) Returns the `LexiconEntry` at index `row`.
int	**getRealIndex**(int index) Returns the real index of row `index`, that is the index in the underlying Lexicon.
java.lang.Object	**getValueAt**(int aRow, int aColumn) Returns the object value at row `aRow` and column `aColumn` of the underlying model.
boolean	**isCellEditable**(int row, int column) Returns true if the cell specified by `row` and `column` should be editable to the user.
void	**reallocateIndexes**() Resets the mapping to the indentiy mapping after adjusting the size of the mapping array.
void	**setModel**(AbstractLexTableModel model) Sets the underlying table model to `model` which is used for sorting.
void	**setValueAt**(java.lang.Object aValue, int aRow, int aColumn) Sets the value at row `aRow` and column `aColumn` of the underlying model to `value`.
void	**sort**(java.lang.Object sender)
void	**sortByColumn**(int column) Performs ascending sorting based on column index `column`.
void	**sortByColumn**(int column, boolean ascending) Performs the sorting based on column index `column` and the sorting order indicated by `ascending`.
void	**tableChanged**(javax.swing.event.TableModelEvent e) Invoked when the underlying table model has changed somehow.

Methods inherited from class ethz.svoxlexedit.LexTableMap
`getColumnClass, getColumnCount, getColumnName, getFilter, getLexicon, getModel, getRowCount`

B.22 Klasse LexTool

ethz.svoxlexedit

Class LexTool

```
java.lang.Object
  |
  +--ethz.svoxlexedit.LexTool
```

---

public class **LexTool**
extends java.lang.Object

This static class represents the extern *Flection Tool* which is used for flection generation.

`LexTool` is a static class, so it is never instantiated and only the static methods are called for communication with the Flection Tool.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary

`static boolean`	**`isvalid`** Indicates if the flection tool is valid, i.e. if the flection tool is running and communication is possible.

Constructor Summary

`LexTool``()` This is the default constructor of the class.

Method Summary	
static void	**clear**() Empties the current contents of the flection tool input stream.
static java.lang.String	**getPhoneticForm**(java.lang.String graph) Returns a phonetic transcription of a graphemic String `graph` via the flection tool.
static java.lang.String	**getResponse**() Returns current contents of the flection tool's standard ouput.
static boolean	**quit**() Quits the lexTool and closes all streams properly.
static boolean	**sendCommand**(java.lang.String command) Sends `command` to the flection tool.
static void	**sleep**(long ms) Waits for `ms` milliseconds.
static boolean	**start**() Starts the flection tool in the background and inits Input and Outputstreams for further communication with the process.

B.23 Klasse LoginWindow

ethz.svoxlexedit

Class LoginWindow

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
                          |
                          +--javax.swing.JFrame
                                |
                                +--ethz.svoxlexedit.LoginWindow
```

All Implemented Interfaces:
javax.accessibility.Accessible, java.awt.event.ActionListener, java.util.EventListener, java.awt.image.ImageObserver, java.awt.MenuContainer, javax.swing.RootPaneContainer, java.io.Serializable, javax.swing.WindowConstants

---

public class **LoginWindow**
extends javax.swing.JFrame
implements java.awt.event.ActionListener

This class represents the Login window which is shown before the main window opens.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`JFrame`, Serialized Form

Constructor Summary

`LoginWindow``()` Constructs the window, registers some listeners , builds the user interface and displays the window in the center of the screen.

Method Summary

void	**`actionPerformed`**`(java.awt.event.ActionEvent e)` Invoked when the user clicks on the OK button.

B.24 Klasse MainWindow

ethz.svoxlexedit

Class MainWindow

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Window
                    |
                    +--java.awt.Frame
                          |
                          +--javax.swing.JFrame
                                |
                                +--ethz.svoxlexedit.MainWindow
```

All Implemented Interfaces:
javax.accessibility.Accessible, java.awt.event.ActionListener, java.util.EventListener, java.awt.image.ImageObserver, javax.swing.event.ListSelectionListener, java.awt.MenuContainer, javax.swing.RootPaneContainer, java.io.Serializable, javax.swing.event.TableColumnModelListener, javax.swing.WindowConstants

---

public class **MainWindow**
extends javax.swing.JFrame
implements java.awt.event.ActionListener, javax.swing.event.ListSelectionListener, javax.swing.event.TableColumnModelListener

This is the main window of the Lexicon Editor and a subclass of JFrame. It holds a menu bar, a toolbar, a status bar, the main lexicon table and a history and flection view.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar
See Also:
`JFrame`, Serialized Form

Constructor Summary
MainWindow() Constructs a new instance of the window and inits all contents.

Method Summary	
void	**actionPerformed**(java.awt.event.ActionEvent e) Invoked when the user clicks a button or chooses a menu item in the menu bar.
void	**clearFilter**() Clears all filter fields and resets the filter.
void	**columnAdded**(javax.swing.event.TableColumnModelEvent e) Invoked when a column is added in the lexicon table.
void	**columnMarginChanged**(javax.swing.event.ChangeEvent e) Invoked when a column margin changes in the lexicon table.
void	**columnMoved**(javax.swing.event.TableColumnModelEvent e) Invoked when columns are moved in the lexicon table.
void	**columnRemoved**(javax.swing.event.TableColumnModelEvent e)

	Invoked when a column is removed in the lexicon table.
void	**columnSelectionChanged**(javax.swing.event.ListSelectionEvent e) Invoked when the column selection changes in the lexicon table.
void	**copy**() Copies the contents of the component with the focus to the clipboard.
void	**cut**() Cuts the contents of the component with the focus to the clipboard.
void	**deleteEntry**() Deletes the selected entries in the lexicon table.
void	**duplicateEntry**() Duplicates the selected entries in the lexicon table.
void	**hideInvalidEntries**() Hides all invalid entries in the lexicon table.
void	**insertEntry**(boolean below, java.lang.String catname) Insert a new entry with category `catname` either below or before the selected entry in the lexicon table.
void	**newLexicon**() Creates a new lexicon.
void	**openLexicon**() Opens a saved lexicon.
void	**paste**() Pastes the contents of the clipboard into the component which has the focus.
boolean	**promptSave**(java.lang.String action) Displays a save confirmation dialog if the current lexicon was modified.
void	**quit**() Quits the Lexicon Editor after showing a save confirmation dialg if necessary.
void	**restartLexTool**() Tries to restart the flection tool.
void	**restoreProperties**() Restores the window properties from the preferences.
void	**saveLexicon**() Saves the Lexicon under the same name.
void	**saveLexiconAs**() Saves the current lexicon under a new name.
void	**saveLexiconAsWithoutHistory**() Save the current lexicon with a new name and without the history.
void	**saveProperties**() Saves the current window and table properties to the preferences.
void	**showInvalidEntries**() Shows all entries in the lexicon table.
void	**toggleFlexview**() Hide or show the flection view.
void	**toggleHistory**() Hide or show the history table.
void	**updateStatusBar**() Updates the contents of the statusbar to match the current selection in the lexicon table.
void	**valueChanged**(javax.swing.event.ListSelectionEvent e) Implementation of ListSelectionListener for updating the statusbar.

B.25 Klasse Preferences

ethz.svoxlexedit

Class Preferences

```
java.lang.Object
  |
  +--ethz.svoxlexedit.Preferences
```

---

public class **Preferences**
extends java.lang.Object

`Preferences` is the class for storing all the user and environment settings.

Copyright: (c) 2002-2003

Organisation: ETH Zurich

Version:
1.0
Author:
Matthias Wille, Dominik Kaspar

Field Summary	
static java.lang.String	**openFile** Name of the lexicon file currently open for editing.
static java.util.HashMap	**prefs** Hashmap of all the user and environment settings.

Constructor Summary
Preferences()

Method Summary	
static boolean	**getBoolean**(java.lang.String lhs) Gets the boolean value of a preference key.
static double	**getDouble**(java.lang.String lhs) Gets the double value of a preference key.
static int	**getInt**(java.lang.String lhs) Gets the integer value of a preference key.
static java.lang.String	**getString**(java.lang.String lhs) Gets the value of a preference key in `String` representation.
static void	**init**() Initializes the `Preferences` to a set of start values.
static void	**readPrefsFromFile**() Reads in preference file into the `prefs` hash map.
static void	**savePrefsToFile**() Stores all the preferences into a text file.
static void	**setBoolean**(java.lang.String lhs, boolean rhs) Sets a preference to a boolean value (KEY = VALUE).
static void	**setDouble**(java.lang.String lhs, double rhs) Sets a preference to a double value.
static void	**setInt**(java.lang.String lhs, int rhs) Sets a preference to an integer value.
static void	**setString**(java.lang.String lhs, java.lang.String rhs) Sets a preference to a `String` value.

B.26 Klasse RE

org.apache.regexp
Class RE

```
java.lang.Object
  |
  +--org.apache.regexp.RE
```

---

public class **RE**
extends java.lang.Object

RE is an efficient, lightweight regular expression evaluator/matcher class. Regular expressions are pattern descriptions which enable sophisticated matching of strings. In addition to being able to match a string against a pattern, you can also extract parts of the match. This is especially useful in text parsing! Details on the syntax of regular expression patterns are given below.

To compile a regular expression (RE), you can simply construct an RE matcher object from the string specification of the pattern, like this:

```
RE r = new RE("a*b");
```

Once you have done this, you can call either of the RE.match methods to perform matching on a String. For example:

```
boolean matched = r.match("aaaab");
```

will cause the boolean matched to be set to true because the pattern "a*b" matches the string "aaaab".

If you were interested in the *number* of a's which matched the first part of our example expression, you could change the expression to "(a*)b". Then when you compiled the expression and matched it against something like "xaaaab", you would get results like this:

```
RE r = new RE("(a*)b");                 // Compile expression
boolean matched = r.match("xaaaab");    // Match against "xaaaab"



String wholeExpr = r.getParen(0);       // wholeExpr will be 'aaaab'
String insideParens = r.getParen(1);    // insideParens will be 'aaaa'



int startWholeExpr = getParenStart(0);  // startWholeExpr will be index 1
int endWholeExpr = getParenEnd(0);      // endWholeExpr will be index 6
int lenWholeExpr = getParenLength(0);   // lenWholeExpr will be 5



int startInside = getParenStart(1);     // startInside will be index 1
int endInside = getParenEnd(1);         // endInside will be index 5
int lenInside = getParenLength(1);      // lenInside will be 4
```

You can also refer to the contents of a parenthesized expression within a regular expression itself. This is called a 'backreference'. The first backreference in a regular expression is denoted by \1, the second by \2 and so on. So the expression:

```
([0-9]+)=\1
```

will match any string of the form n=n (like 0=0 or 2=2).

The full regular expression syntax accepted by RE is described here:

Characters

unicodeChar	Matches any identical unicode character
\	Used to quote a meta-character (like '*')
\\	Matches a single '\' character
\0nnn	Matches a given octal character
\xhh	Matches a given 8-bit hexadecimal character
\\uhhhh	Matches a given 16-bit hexadecimal character
\t	Matches an ASCII tab character
\n	Matches an ASCII newline character
\r	Matches an ASCII return character
\f	Matches an ASCII form feed character

Character Classes

[abc]	Simple character class
[a-zA-Z]	Character class with ranges
[^abc]	Negated character class

Standard POSIX Character Classes

[:alnum:]	Alphanumeric characters.
[:alpha:]	Alphabetic characters.
[:blank:]	Space and tab characters.
[:cntrl:]	Control characters.
[:digit:]	Numeric characters.
[:graph:]	Characters that are printable and are also visible. (A space is printable, b
[:lower:]	Lower-case alphabetic characters.
[:print:]	Printable characters (characters that are not control characters.)
[:punct:]	Punctuation characters (characters that are not letter, digits, control char
[:space:]	Space characters (such as space, tab, and formfeed, to name a few).
[:upper:]	Upper-case alphabetic characters.
[:xdigit:]	Characters that are hexadecimal digits.

Non-standard POSIX-style Character Classes

```
[:javastart:]       Start of a Java identifier
[:javapart:]        Part of a Java identifier
```

Predefined Classes

```
.                   Matches any character other than newline
\w                  Matches a "word" character (alphanumeric plus "_")
\W                  Matches a non-word character
\s                  Matches a whitespace character
\S                  Matches a non-whitespace character
\d                  Matches a digit character
\D                  Matches a non-digit character
```

Boundary Matchers

```
^                   Matches only at the beginning of a line
$                   Matches only at the end of a line
\b                  Matches only at a word boundary
\B                  Matches only at a non-word boundary
```

Greedy Closures

```
A*                  Matches A 0 or more times (greedy)
A+                  Matches A 1 or more times (greedy)
A?                  Matches A 1 or 0 times (greedy)
A{n}                Matches A exactly n times (greedy)
A{n,}               Matches A at least n times (greedy)
A{n,m}              Matches A at least n but not more than m times (greedy)
```

Reluctant Closures

```
A*?                 Matches A 0 or more times (reluctant)
A+?                 Matches A 1 or more times (reluctant)
A??                 Matches A 0 or 1 times (reluctant)
```

Logical Operators

```
AB                  Matches A followed by B
A|B                 Matches either A or B
(A)                 Used for subexpression grouping
```

Backreferences

```
\1                  Backreference to 1st parenthesized subexpression
\2                  Backreference to 2nd parenthesized subexpression
\3                  Backreference to 3rd parenthesized subexpression
\4                  Backreference to 4th parenthesized subexpression
\5                  Backreference to 5th parenthesized subexpression
\6                  Backreference to 6th parenthesized subexpression
\7                  Backreference to 7th parenthesized subexpression
\8                  Backreference to 8th parenthesized subexpression
\9                  Backreference to 9th parenthesized subexpression
```

All closure operators (+, *, ?, {m,n}) are greedy by default, meaning that they match as many elements of the string as possible without causing the overall match to fail. If you want a closure to be reluctant (non-greedy), you can simply follow it with a '?'. A reluctant closure will match as few elements of the string as possible when finding matches. {m,n} closures don't currently support reluctancy.

RE runs programs compiled by the RECompiler class. But the RE matcher class does not include the actual regular expression compiler for reasons of efficiency. In fact, if you want to pre-compile one or more regular expressions, the 'recompile' class can be invoked from the command line to produce compiled output like this:

```
// Pre-compiled regular expression "a*b"
char[] re1Instructions =
{
    0x007c, 0x0000, 0x001a, 0x007c, 0x0000, 0x000d, 0x0041,
    0x0001, 0x0004, 0x0061, 0x007c, 0x0000, 0x0003, 0x0047,
    0x0000, 0xfff6, 0x007c, 0x0000, 0x0003, 0x004e, 0x0000,
    0x0003, 0x0041, 0x0001, 0x0004, 0x0062, 0x0045, 0x0000,
    0x0000,
};


REProgram re1 = new REProgram(re1Instructions);
```

You can then construct a regular expression matcher (RE) object from the pre-compiled expression re1 and thus avoid the overhead of compiling the expression at runtime. If you require more dynamic regular expressions, you can construct a single RECompiler object and re-use it to compile each expression. Similarly, you can change the program run by a given matcher object at any time. However, RE and RECompiler are not threadsafe (for efficiency reasons, and because requiring thread safety in this class is deemed to be a rare requirement), so you will need to construct a separate compiler or matcher object for each thread (unless you do thread synchronization yourself).

ISSUES:

- com.weusours.util.re is not currently compatible with all standard POSIX regcomp flags

- com.weusours.util.re does not support POSIX equivalence classes ([=foo=] syntax) (I18N/locale issue)
- com.weusours.util.re does not support nested POSIX character classes (definitely should, but not completely trivial)
- com.weusours.util.re Does not support POSIX character collation concepts ([.foo.] syntax) (I18N/locale issue)
- Should there be different matching styles (simple, POSIX, Perl etc?)
- Should RE support character iterators (for backwards RE matching!)?
- Should RE support reluctant {m,n} closures (does anyone care)?
- Not *all* possibilities are considered for greediness when backreferences are involved (as POSIX suggests should be the case). The POSIX RE "(ac*)c*d[ac]*\1", when matched against "acdacaa" should yield a match of acdacaa where \1 is "a". This is not the case in this RE package, and actually Perl doesn't go to this extent either! Until someone actually complains about this, I'm not sure it's worth "fixing". If it ever is fixed, test #137 in RETest.txt should be updated.

Version:
$Id: RE.java,v 1.6 2000/08/22 17:19:38 jon Exp $
Author:
Jonathan Locke
See Also:
`recompile`, `RECompiler`

Method Summary

int	**getMatchFlags**() Returns the current match behaviour flags.
java.lang.String	**getParen**(int which) Gets the contents of a parenthesized subexpression after a successful match.
int	**getParenCount**() Returns the number of parenthesized subexpressions available after a successful match.
int	**getParenEnd**(int which) Returns the end index of a given paren level.
int	**getParenLength**(int which) Returns the length of a given paren level.
int	**getParenStart**(int which) Returns the start index of a given paren level.
REProgram	**getProgram**() Returns the current regular expression program in use by this matcher object.
java.lang.String[]	**grep**(java.lang.Object[] search) Returns an array of Strings, whose toString representation matches a regular expression.
protected void	**internalError**(java.lang.String s) Throws an Error representing an internal error condition probably resulting from a bug in the regular expression compiler (or possibly data corruption).
boolean	**match**(CharacterIterator search, int i) Matches the current regular expression program against a character array, starting at a given index.
boolean	**match**(java.lang.String search) Matches the current regular expression program against a String.

boolean	**match**(java.lang.String search, int i) Matches the current regular expression program against a character array, starting at a given index.
protected boolean	**matchAt**(int i) Match the current regular expression program against the current input string, starting at index i of the input string.
protected int	**matchNodes**(int firstNode, int lastNode, int idxStart) Try to match a string against a subset of nodes in the program
void	**setMatchFlags**(int matchFlags) Sets match behaviour flags which alter the way RE does matching.
protected void	**setParenEnd**(int which, int i) Sets the end of a paren level
protected void	**setParenStart**(int which, int i) Sets the start of a paren level
void	**setProgram**(REProgram program) Sets the current regular expression program used by this matcher object.
static java.lang.String	**simplePatternToFullRegularExpression**(java.lang.String pattern) Converts a 'simplified' regular expression to a full regular expression
java.lang.String[]	**split**(java.lang.String s) Splits a string into an array of strings on regular expression boundaries.
java.lang.String	**subst**(java.lang.String substituteIn, java.lang.String substitution) Substitutes a string for this regular expression in another string.
java.lang.String	**subst**(java.lang.String substituteIn, java.lang.String substitution, int flags) Substitutes a string for this regular expression in another string.

Anhang C

Description-Datei

```
// Configuration File for the Java Lexicon Editor File Format
// Defines which Categories have which features
// and defines the valid values for each feature;

// FEATURE DEFINITIONS
// Define the features and possible values
// The features GRAPH, PHON and CAT must NOT
// be renamed

[Feature] GRAPH "Graphemic Form"
  { a b c d e f g h i j k l m n o p q r s t u v w x y z

               $ + # }
[End Feature]

[Feature] PHON "Phonetic Form"
  { a a: q A A: ~a V 6 ^6 a_i a_I a_u a_U b b_b B C d D
    d_d d_d_Z d_d_z d_Z d_z 3 3: e e: E E: ~E e_I e_@ @
    @_U c f f_f F g G g_g h i i: I ^i i_@ I_@ j J J_J k
    k_k k_h l =l L L_L l_l m =m m_m n n_n =n N o o: O
    O: ^o ~o O_I O_y Q 2 2: 9 ~9 p p_p p_f p_h r R r_r
    s S S_S s_s t t_t t_t_S t_t_s T t_h t_s t_S u u: U
    ^u  U_@ v v_v w x y y: Y ^y y_@ H z Z ? ' , - ( ) <
    >           $ + | \ # }
[End Feature]

[Feature] CAT "Syntactic Category of Entry"
  [ NS "Nomenstamm" NPRS "Nomen-Proprium-Stamm"
    NPRP "Nomen-Proprium-Partikel" VS "Verbstamm"
    AS "Adjektvstamm" ADVS "Adverbstamm"
    ASER "Abgeleit. Stamm mit ER" NI "No Info" ]
[End Feature]
```

```
[Feature] G "Gender"
  [ M "Maskulin" F "Feminin" N "Neutrum" FM "Fem-und-Mask" NI "No Info" ]
[End Feature]

[Feature] SK "Singular Class"
  [ SK0 SK1 SK2 SK3 SK4 SK5 SK6 SK7 SK8 SK9
    SK10 SK11 NI "No Info" ]
[End Feature]

[Feature] PK "Plural Class"
  [ PK0 PK1 PK2 PK3 PK4 PK5 PK6 PK7 PK8 PK9
    PK10 PK11 PK12 PK13 NI "No Info" ]
[End Feature]

[Feature] VT "Verbclass"
  [ V1 "Regulr" V2 V3 V4 V5 V6 V7 V8 V9 V10 NI "No Info" ]
[End Feature]

[Feature] VGF "Stem Type"
  [ A "Infinitvstamm" B "2. Person Prsens" C "Imperfektstamm"
    D "Konjunktiv2 Stamm" NI "No Info" ]
[End Feature]

[Feature] SC "Surface Class"
  [ V "Vollverb" AUXH "Hilfsverb haben" AUXS "Hilfsverb sein"
    AUXW "Hilfsverb werden" MVERB "Modalverb" NI "No Info" ]
[End Feature]

[Feature] REF "Reflexive"
  [ NON "Nichtreflexiv" A "Akkusativreflexiv" D "Dativreflexiv" NI "No Info" ]
[End Feature]

[Feature] ADPOS "Adjective positive stem"
  [ P "Positivstamm" X "kein Positivstamm" NI "No Info" ]
[End Feature]

[Feature] ADKOMP "Adjective comparative stem"
  [ K "Komparativstamm" X "Kein Komparativstamm" NI "No Info" ]
[End Feature]

[Feature] ADSUP "Adjective superlative stem"
  [ S "Superlativstamm" X "Kein Superlativstamm" NI "No Info" ]
[End Feature]

[Feature] ADVSEM "Semantic adverb category"
  [ M "Modal" T "Temporal" L "Lokal" D "Direktional" C "Kausal" NI "No Info" ]
[End Feature]
```

```
[Feature] ORIG "Language of origin"
  [ D "Deutsch" F "Franzsisch" I "Italienisch" R "Romanisch" E "Englisch"
    S "Spanisch" P "Portugiesisch" T "Trkisch" SL "Slawisch" AR "Arabisch"
    AS "Asiatisch" A "Andere" NI "No Info" ]
[End Feature]

[Feature] TRANS "Language of transcription"
  [ D "Deutsch" F "Franzsisch" E "Englisch" I "Italienisch"
    SD "Schweizerdeutsch" SV "Schweizerdeutsch-Variante" NI "No Info" ]
[End Feature]

[Feature] SUBCAT "Semantic category of proper name"
  [ VN "Vorname" NN "Nachname" FN "Firmenname" ON "Ortsname"
    SN "Strassenname" GN "Geograph. Name" NI "No Info" ]
[End Feature]

[Feature] FREQ "Wie verbreitet ist der Gebrauch dieses Wortes?"
  [ 1 2 3 4 5 6 7 8 9 10 NI "No Info" ]
[End Feature]

[Feature] SRC "Source of Information"
  [ LE "Lexikon" AN "Anfrage" PK "Pers. Kenntnis" KA "Kenntnis andere"
    OF "Offensichtlich" AR "Aussprache-Regeln" AT "Automat-Transkription"
    NB "Namensbuch" CT "Christina" EN "ENST" NI "No Info" ]
[End Feature]

[Feature] USER "User"
  [ CT "Christina" KA "Kthi" BP "Beat" CHT "Christof" GL "Gunnar" KH "Karl"
    MR "Marcel" SS "Schamai" VJ "Volker" HR "Harald" NI "No Info" ]
[End Feature]

[Feature] STATUS "Status of entry"
  [ UB "Unbehandelt" US "Unsicher" PS "Plausibel" SI "Grosse Sicherheit"
    TR "Transkription" PR "Aussprache" UE "Uebersetzung" VL "Vokallnge"
    AK "Betonung" KA "Kategorie" GE "Genus" SG "Singular" PL "Plural"
    HK "Herkunft" SP "Tr-Sprache" SK "Subkategorie" GB "Gebrauch"
    OR "Orthographie" NI "No Info" ]
[End Feature]



// CATEGORY DEFINITIONS
// Define the feature list for the different categories
// in the order they appear in the masterlexicon
// The features GRAPH, PHON and CAT need to be on the
// first three positions!!

[Category] NPRS "Nomen-Proprium-Stamm"
```

```
  [ GRAPH PHON CAT G SK PK ORIG TRANS SUBCAT FREQ SRC USER STATUS ]
[End Category]

[Category] NPRP
  [ GRAPH PHON CAT G SK PK ORIG TRANS SUBCAT FREQ SRC USER STATUS ]
[End Category]

[Category] VS
  [ GRAPH PHON CAT VT VGF SC REF ORIG TRANS FREQ SRC USER STATUS ]
[End Category]

[Category] NS
  [ GRAPH PHON CAT G SK PK ORIG TRANS FREQ SRC USER STATUS ]
[End Category]

[Category] AS
  [ GRAPH PHON CAT ADPOS ADKOMP ADSUP ORIG TRANS FREQ SRC USER STATUS ]
[End Category]

[Category] ADVS
  [ GRAPH PHON CAT ADVSEM ORIG TRANS FREQ SRC USER STATUS ]
[End Category]

[Category] ASER
  [ GRAPH PHON CAT ORIG TRANS FREQ SRC USER STATUS ]
[End Category]

[Category] DEFAULT
  [ GRAPH PHON CAT ORIG TRANS FREQ SRC USER STATUS ]
[End Category]
```

Anhang D

Preference-Datei

```
LEXICON EDITOR PREFERENCES:
---------------------------

COLUMN_POSITION_0 = 0
COLUMN_POSITION_1 = 1
COLUMN_POSITION_10 = 10
COLUMN_POSITION_11 = 9
COLUMN_POSITION_12 = 12
COLUMN_POSITION_2 = 2
COLUMN_POSITION_3 = 7
COLUMN_POSITION_4 = 6
COLUMN_POSITION_5 = 4
COLUMN_POSITION_6 = 8
COLUMN_POSITION_7 = 3
COLUMN_POSITION_8 = 5
COLUMN_POSITION_9 = 11
COLUMN_WIDTH_0 = 88
COLUMN_WIDTH_1 = 78
COLUMN_WIDTH_10 = 52
COLUMN_WIDTH_11 = 48
COLUMN_WIDTH_12 = 48
COLUMN_WIDTH_2 = 51
COLUMN_WIDTH_3 = 50
COLUMN_WIDTH_4 = 51
COLUMN_WIDTH_5 = 50
COLUMN_WIDTH_6 = 51
COLUMN_WIDTH_7 = 49
COLUMN_WIDTH_8 = 52
COLUMN_WIDTH_9 = 49
DIVIDER_FLEX = 548
DIVIDER_HISTORY = 536
HIDE_FLEXVIEW = TRUE
```

```
HIDE_HISTORY = TRUE
HTML_TEMPLATE_DEFAULT = default.html
HTML_TEMPLATE_NS = ns.html
HTML_TEMPLATE_VS = vs_ggw.html
LEXTOOL_BASE_CONS_COMMAND = \cons $CONS
LEXTOOL_BASE_FEAT_COMMAND = \feat $FEAT_NAME $FEAT_VALUE
LEXTOOL_CLEAR_COMMAND = \clear
LEXTOOL_FLECT_COMMAND = \flect
LEXTOOL_GRAPH_COMMAND = \graph $GRAPH
LEXTOOL_GRAPH_TO_PHON_COMMAND = \phon
LEXTOOL_OPTIONS = -c lexedit.cmd
LEXTOOL_PATH = /home/sprstud1/lexedit_v1.1/lexedit
LEXTOOL_PHON_COMMAND = \phon $PHON
LEXTOOL_PROMPT = SVOX LexEdit>
LEXTOOL_QUIT_COMMAND = \quit
LEXTOOL_TARGET_CONS_COMMAND = \flectCons $CONS
LEXTOOL_TARGET_FEAT_COMMAND = \flectFeat $FEAT_NAME $FEAT_VALUE
USER = KH
WINDOW_HEIGHT = 673
WINDOW_POS_X = 110
WINDOW_POS_Y = 48
WINDOW_WIDTH = 724
WORKING_DIR = /Users/matthias/SVOXLexiconEditor/Masterlexica Files
```

Anhang E

CD-ROM

Die beiliegende CD-ROM enthält die Ergebnisse unserer Arbeit, allen voran die kompilierte und ausführbare Version des Java Lexicon Editors zusammen mit der Dokumentation und dem Sourcecode.

Die CD-ROM enthält folgendes:

- Java jar-File des Java Lexicon Editors
- Java Sourcecode des Java Lexicon Editors
- Flektionstool lexedit (ohne Data-Files)
- Durch javadoc generierte Klassen-API
- Beispiellexika
- PDF-Version dieses Berichts