

Prüfer: Prof. Dr. E. Lehmann
Betreuer: Dr. phil. L. Wanner

begonnen am: 1. März 1998
beendet am: 31. August 1998

CR-Klassifikation: I.2.6, I.2.7, I.2.8

Entwicklung und Implementierung
einer intelligenten Schnittstelle
für maschinell-nutzbare Erklärende
Kombinatorische Wörterbücher in Java

Klaus Hildner
Studienarbeit Nr. 1709

Institut für Informatik
Universität Stuttgart
Breitwiesenstraße 20 – 22
D-70565 Stuttgart

Zusammenfassung

Diese Studienarbeit beschäftigt sich mit der Entwicklung und Implementierung einer intelligenten Schnittstelle für maschinell-nutzbare Erklärende Kombinatorische Wörterbücher in Java.

Ausgehend von einer Problembeschreibung wird entlang des Wasserfallmodells der Softwareentwicklung über die Phasen der Analyse und des Entwurfs ein lauffähiger Prototyp in Java implementiert und dessen grundlegende Funktionalität dokumentiert.

Es wird eine Beschreibung der verwendeten wissensbasierten Algorithmen zur Überprüfung der syntaktischen und semantischen Konsistenz von lexikalischen Einträgen gegeben, die einen Benutzer bei der Arbeit mit einem Wörterbuch intelligent unterstützen, mögliche Fehler aufdecken sowie Gemeinsamkeiten zwischen Einträgen erkennen und Generalisierungen (Verallgemeinerungen) ermöglichen sollen.

Abschließend werden die Resultate diskutiert und ein Ausblick gegeben.

Die Begriffe “Java”, “JDK”, “Java CompilerCompiler” und andere sind eingetragene Warenzeichen und als solche geschützt.

Schlüsselwörter:

LEXIKOGRAPHIE, LEXIKON, LEXIKALISCHE FUNKTIONEN, SEMANTISCHE MERKMALE, INFORMATIONSEXTRAKTION, ERKLÄRENDES KOMBINATORISCHES WÖRTERBUCH

key words:

LEXICOGRAPHY, LEXICON, LEXICAL FUNCTIONS, SEMANTIC FEATURES, INFORMATION EXTRACTION, EXPLANATORY COMBINATORIAL DICTIONARY

Inhaltsverzeichnis

1	Einleitung	1
1.1	Problemstellung	1
1.2	Zielsetzung	2
1.3	Übersicht	3
1.4	Struktur der Studienarbeit	3
1.5	Verwendete Terminologie	4
2	Erklärende Kombinatorische Wörterbücher	5
2.1	Übersicht	5
2.2	Der semantische Bereich: die Definition	5
2.3	Der syntaktische Bereich: das Valenzschema	7
2.4	Der Bereich der lexikalischen Funktionen	7
3	Überprüfungen und Generalisierungen	9
3.1	Syntaktische Überprüfungen	9
3.1.1	Motivation	9
3.1.2	Resultate	9
3.2	Semantische Überprüfungen	10
3.2.1	Motivation	10
3.3	Generalisierungen	11
3.3.1	Motivation	11
3.3.2	Vorüberlegungen	11
3.3.3	Das Auffinden einfacher Korrelationen	13
3.3.4	Das Auffinden komplexerer Korrelationen	14
3.3.5	Statistische Auswertungen	16
3.3.6	Ansatz 1: Entscheidungsbäume	20
3.3.7	Ansatz 2: Ausdrucksbäume	21
3.3.8	Ansatz 3	23
4	Spezifikation	24
4.1	Ist-Analyse	24
4.2	Anforderungen	25
4.3	Interaktionsform	26
4.4	Funktionalität des Menüsystems	26
4.5	Syntaktische Überprüfungen	30
4.6	Semantische Überprüfungen	31
4.7	Generalisierungen	32
4.8	Weitere Funktionalitäten	32

5 Entwurf	33
5.1 Abstrakte Datentypen	34
5.2 Datenstrukturen	34
5.3 Zentrale Datenstrukturen	36
5.4 Funktionale Anforderungen	37
5.5 Externe Schnittstellen	37
5.6 Leistungsanforderungen	37
5.7 Entwurfseinschränkungen und Portierbarkeit	38
5.8 Modularisierung	38
5.9 Syntaktische Überprüfungen	38
5.10 Semantische Überprüfungen	39
5.11 Generalisierungen	39
5.11.1 Statistische Auswertungen	39
5.11.2 Entscheidungsbaum	41
5.11.3 Ausdrucksbaum	42
6 Implementierung	45
6.1 Java	45
6.1.1 Allgemeine Einführung	45
6.1.2 Der Java CompilerCompiler	46
6.1.3 Vorüberlegungen	47
6.1.4 Datenstrukturen und Methoden von Java	47
6.2 Klassen des Wörterbuchs	48
6.2.1 Die Klasse <code>singleDico.java</code>	49
6.2.2 Die Klasse <code>singleEntry.java</code>	49
6.2.3 Die Klasse <code>ECD.java</code>	50
6.2.4 Die Klasse <code>Dico.java</code>	51
6.2.5 Die Klasse <code>Entry.java</code>	51
6.2.6 Die Klassen <code>Tok*.java</code>	52
6.2.7 Die Klasse <code>Undo.java</code>	53
6.2.8 Die Klasse <code>Search.java</code>	54
6.2.9 Die Klasse <code>SearchExpr.java</code>	54
6.2.10 Die Klasse <code>Check.java</code>	55
6.2.11 Die Klasse <code>Buffer.java</code>	55
6.2.12 Die Klasse <code>ECDFrame.java</code>	56
6.2.13 Weitere Klassen	57
7 Zusammenfassung und Ausblick	58
7.1 Allgemeine Ergebnisse	58
7.2 Resultate	58
7.3 Ausblick	59

Literatur

60

A Die Klasse `singleEntry.java`

62

B Die Klasse `singleDico.java`

65

C Grammatik für die Ausdruckssuche

67

Abbildungsverzeichnis

1	Veranschaulichung der Implikation	16
2	Beispiel für einen Ausdrucksbaum	22
3	Spezifikation des Menüsystems: Übersicht 1	26
4	Spezifikation des Menüsystems: Übersicht 2	27
5	Beispiel: Dateiauswahlbox	28
6	Spezifikation des Menüsystems: das Eintrags-Menü	28
7	Spezifikation: das Such-Menü	29
8	Spezifikation: das Check-Menü	29
9	Spezifikation: das Puffer-Menü (Beispiel)	29
10	Spezifikation: das Rahmen-Menü	30
11	Übersicht über den Entwurf der Klassen	33

Tabellenverzeichnis

1	Beispiel für semantische Dimensionen	6
2	Beispiel für ein Valenzenschema	7
3	Beispiel für eine Lexem \times LFvalue-Matrix	12
4	Korrelation LFid \times semantisches Merkmal	16
5	Korrelation LFvalue \times semantisches Merkmal	17
6	Korrelation semantisches Merkmal \Rightarrow semantisches Merkmal	18
7	Korrelation semantisches Merkmal \times semantisches Merkmal	19

1 Einleitung

1.1 Problemstellung

Ein Erklärendes Kombinatorisches Wörterbuch EKW (engl. *Explanatory Combinatorial Dictionary*, ECD) unterscheidet sich von einem herkömmlichen Wörterbuch zum einen durch die größere Breite der erfaßten Informationen und zum anderen durch eine stärkere Formalisierung der Repräsentation der Informationen.

EKWs werden im Bereich der Linguistik und der Sprachverarbeitung u. a. verwendet

- für die Erfassung von Kollokationen¹, Wortdefinitionen und Valenzen (zur Benutzung sowohl durch Menschen als auch durch Software),
- für die maschinelle, bedeutungsorientierte Übersetzung (lexikalische Funktionen als eine Art bedeutungsorientierte Zwischensprache (engl. *interlingua*),
- für die Textgenerierung² als Hilfsmittel bei der Wahl der spezifischen Kommunikationsstruktur einer zu produzierenden Äußerung und das Erlangen von Kohäsion.

Die Verbreitung von EKWs wird wegen verbesserter Nutzungsmöglichkeiten (z. B. online-Abfrage übers WWW³, Einbindung in größere Software-Pakete z. B. als Synonym-Wörterbücher) künftig sicherlich zunehmen.

Um diesem Nachfragezuwachs gerecht zu werden, kann versucht werden, EKWs (semi)automatisch aus Ansammlungen von Textkorpora⁴ zu generieren. Diese Technik ist allerdings fehlerbehaftet; beispielsweise können auf diese Art Kollokationen nicht sicher zugeordnet werden — ein Mensch und sein Sprachverständnis oder gar eine Gruppe von Menschen, die dieselbe Muttersprache sprechen und sich aufgrund ihrer Erfahrung im Umgang mit der Sprache auf “gültige” Äußerungen bzw. Einträge einigen, werden weiterhin benötigt.

Deshalb ist eine entsprechende Unterstützung des Lexikographen⁵ bei der Erstellung von Wörterbucheinträgen notwendig. Denkbar sind die Entwicklung eines Editors zur online-Eingabe (und nicht nur *Abfrage*) von Wörterbuchdaten ebenfalls übers WWW, die gleichzeitige Arbeit und Koordination mehrerer Benutzer an einem Projekt zur Erstellung eines Wörterbuchs sowie die spezifische Unterstützung eines einzelnen Benutzers bei der Eingabe und Verwaltung der Daten, um ihn von lästigen, sich wiederholenden, fehleranfälligen Aufgaben zu befreien und seine Aufmerksamkeit auf die eigentliche kreative Tätigkeit zu lenken. Darüberhinaus kann sogar versucht werden, Teile dieser schöpferischen Tätigkeit von der Maschine übernehmen zu lassen, indem beispielsweise Vorhersagen von Kollokationen versucht werden.

Letztere Art der Unterstützung einer einzelnen Person, um die sich diese Studienarbeit bemüht, kann mehrere Punkte umfassen:

¹inhaltliche Kombinierbarkeit sprachlicher Einheiten miteinander, z. B. Biene + summen, dick + Buch, aber nicht: dick + Haus

²siehe hierzu auch [Melcuk1996], S. 91 – 96 und [Wanner1996], S. 2 und S. 27

³einfache Beispiele hierfür sind die Angebote von <http://www.iicm.edu/TWE> (Langenscheidts Taschenwörterbuch Englisch), <http://www.gsmuc.de/look.html> (Langenscheidts Handwörterbuch) und <http://babelfish.altavista.digital.com/cgi-bin/translate?URL> (AltaVistas Übersetzungsservice für u. a. komplette Webseiten — HyperLinks bleiben erhalten)

⁴Referenztexten

⁵im folgenden gilt natürlich immer auch die weibliche Variante, siehe [DeinigerEtAl1992], S. 10

- Hilfe bei der eigentlichen Aufgabe des Editierens von Texten, z. B. Bereitstellen einer Eingabemaske, “cut/copy-and-paste”),
- Hilfe beim Navigieren innerhalb der Datenbasis (z. B. komfortable Suchfunktionen unter Berücksichtigung der Struktur des Wörterbuchs),
- Unterstützung bei der Vermeidung bzw. Offenbarmachung vielfältigster menschlicher (Eingabe)Fehler oder Warnung bei möglichen Fehlern (z. B. inhaltliche Inkonsistenzen bei der Festlegung von Einträgen, und zwar sowohl syntaktischer (z. B. nicht unterstützte Bezeichner) als auch semantischer (z. B. inhaltlich inkonsistente Einträge) Art) als auch — sozusagen als kreative Unterstützung
- statistische, intelligente und wissensbasierte Verfahren zur Prüfung, Untersuchung und Auswertung des Datenbestandes (Versuch, weitere Informationen abzuleiten und zu generalisieren).

1.2 Zielsetzung

Diese Studienarbeit beschäftigt sich mit der Entwicklung und Implementierung einer intelligenten Schnittstelle für maschinell-nutzbare Erklärende Kombinatorische Wörterbücher in Java.

Vorhanden ist ein exemplarischer Datenbestand von über einhundert Wörterbucheinträgen, der mittels eines (externen) Editors (emacs) “manuell” erstellt wurde; die Navigation erfolgt bislang mittels Betriebssystem-Befehlen; die Kohärenz- und Konsistenz-Überprüfungen sind bisher voll von den Fähigkeiten des Benutzers abhängig. Darauf aufbauend soll eine komfortable, “intelligente” Benutzerschnittstelle geschaffen werden, mit deren Menüsystem die Auswahl eines Datenbestandes, die Navigation darin, die Modifikation von Teilen desselben sowie syntaktische und semantische Überprüfungen des Inhalts erlaubt sein sollen.

Diese Überprüfungen sollen den Test auf Kohärenz und Konsistenz der Wörterbuchartikel untereinander, einen Abgleich der Artikel in Bezug auf Vollständigkeit und Hypothesenbildung hinsichtlich der Ähnlichkeit spezifischer Informationen in verschiedenen Artikeln umfassen, wobei Algorithmen aus dem Gebiet des maschinellen Lernens verwendet werden sollen.

Folgende Aufgaben sind zu lösen:

- Die Schnittstelle zum Betriebssystem soll stärker vor dem Benutzer verborgen werden.
- Die Verwaltung der Daten soll einheitlich präsentiert werden und auch ungeübteren Benutzern zugänglich sein.
- Mögliche Fehleingaben des Benutzers (z. B. unwahrscheinliche Kollokationen eines Lexems) sollen erkennbar werden.
- Die bisherige “intuitive” Optimierung, Generalisierung, etc. der Einträge soll möglichst automatisiert werden.

Zwei weitere Parameter sollen variabel gehalten werden:

- *Internationalisierung*: Die Benutzermeldungen (z. B. Menübeschriftungen) sollen initial (beim Programmstart) sprachspezifisch konfigurierbar sein.

- *variable, synonyme tokens*: Unabhängig von obiger Internationalisierung u. a. der Menübeschriftungen soll es erlaubt sein, beliebige tokens (also auch verschiedene Sprachen gemischt) innerhalb eines Eintrags (d. h. im Datenbestand) synonym zu verwenden. Die Anordnung derselben soll beliebig möglich sein. Es müssen nicht alle der tokens für ein bestimmtes Lexem vorhanden sein.

Beispielsweise soll es möglich sein, das “Beispiel”-Feld jedes Eintrags sowohl mit **EXAMPLE:** als auch mit ***Beispiel*:** einzuleiten. Die Anordnung dieses Feldes zwischen den anderen möglichen Feldern (Definition des Lexems, Bereich der semantischen Merkmale, etc.) soll beliebig sein; es soll also keine feste Reihenfolge vorgegeben sein. Diese Festlegung bietet den Vorteil, daß bereits vorhandene Wörterbuchbestände leichter mit dem zu entwickelnden Programm verwendet werden können — statt den Datenbestand (d. h. den Eintrag für jedes einzelne Lexem) anzupassen, müssen dem Programm nur die synonymen tokens angegeben werden.

Java soll als Programmiersprache verwendet werden, die Portabilität des Programms gewährleistet; das Programm soll eventuell zu einem späteren Zeitpunkt unter *Netscape* (als Java-applet) laufen.

1.3 Übersicht

Ausgehend von einer Problembeschreibung wird im Rahmen dieser Studienarbeit entlang des Wasserfallmodells der Softwareentwicklung (siehe z. B. [Melchisedech1997]) über die Phasen der Spezifikation und des Entwurfs ein lauffähiger Prototyp in Java implementiert. Dessen grundlegende Funktionalität wird ausreichend dokumentiert.

Innerhalb der Spezifikation wird eine genauere Beschreibung der verwendeten wissensbasierten Algorithmen zur Überprüfung der syntaktischen und semantischen Konsistenz von lexikalischen Einträgen gegeben, die einen Benutzer bei der Arbeit mit dem EKW intelligent unterstützen sollen.

Die Ergebnisse von Programmläufen werden zusammenfassend beurteilt. Es wird sich zeigen, daß diese Algorithmen sehr wohl Inkonsistenzen aufdecken können, daß aber andererseits aufgrund fehlender Information bei weitem nicht alle möglichen Fehler entdeckt werden (können), und auch beim Versuch der statistischen Auswertung bestimmter Häufigkeiten können nur erste Hinweise auf mögliche Generalisierungen gegeben werden, die weiter “durch Geschick und Intuition” analysiert werden müssen.

1.4 Struktur der Studienarbeit

Das nachfolgende Kapitel 2 beschäftigt sich mit Erklärenden Kombinatorischen Wörterbüchern, erklärt deren Komponenten und gibt Beispiele für Einträge. Daraufhin werden im Kapitel 3 (S. 9) syntaktische und semantische Überprüfungen sowie Verallgemeinerungen der in einem Wörterbuch enthaltenen Informationen motiviert und Beispiele gegeben.

Im Kapitel 4 (S. 24) wird durch eine *Analyse* eine Bestandsaufnahme der gegenwärtigen Situation vollzogen und die *Spezifikation* für ein Benutzerinterface für EKWs gegeben. Das Kapitel 5 (S. 33) enthält den objektorientierten *Entwurf* der Software, dem im Kapitel 6 (S. 45) die Beschreibung der *Implementierung* in Java folgt.

Das Kapitel 7 (S. 58) enthält eine abschließende *Zusammenfassung* und einen Ausblick über mögliche Erweiterungen.

Im Anhang A (S. 62) findet sich die Definition eines einzelnen Wörterbucheintrags als Java-Klasse `singleEntry.java`; Instanzen dieser Klasse werden aus den vom Dateisystem her übernommenen Einträgen im Hauptspeicher erzeugt. Im Anhang B (S. 65) findet sich die entsprechende Definition eines ganzen Wörterbuchs als Java-Klasse `singleDico.java`; Instanzen dieser Klasse bestehen aus mehreren einzelnen Einträgen (`singleDico`) und stellen zusätzlich über weitere Datenstrukturen zwischen diesen Querbezüge her. Der Anhang C (S. 67) enthält die Grammatikdefinition für eine komfortable Abfragesprache zur Suche von Einträgen, die ein bestimmtes (eventuell rekursiv geschachteltes) boolesches Muster erfüllen.

1.5 Verwendete Terminologie

Artikel: ein (Wörterbuch)Eintrag (engl. *entry*)

Ausdruck: ein logischer Ausdruck unter Verwendung der Operatoren `&` (Ampersand, alternativ Semikolon) für Durchschnitts-, `|` (Pipe, alternativ Komma) für Vereinigungs- und `-` (Neg) für Komplementbildung (bezüglich einer Grundmenge), der zu einer Menge von Schlüsselwörtern, deren Einträge diesen Ausdruck erfüllen, evaluiert werden kann

Ausdrucksbaum: siehe hierzu Kapitel 3.3.7, S. 21

Explanatory Combinatorial Dictionary, ECD:

(deutsch: Erklärendes Kombinatorisches Wörterbuch, EKW); ECD ist der Name des Hauptprogramms der entwickelten Software

Entscheidungsbaum: siehe hierzu Kapitel 3.3.6, S. 20

Expansion: sinnvolle Ergänzung einer Eingabe auf einen (bereits vorhandenen) gültigen Wert hin, beispielsweise bei Eingabe von “Anfr” Erweiterung auf “Anfrage”

Lexem, Schlüssel: synonym verwendet für die innerhalb desselben Wörterbuchs eindeutige “Überschrift” eines Eintrags zusammen mit der Lesart, beispielsweise `Gefühl 1a`

lexikalische Funktion, LF: näheres zu diesem Konzept siehe nachfolgend im Kapitel 2.4, S. 7

LFid: der Bezeichner (engl. *identifier*) einer LF

LFvalue: der (Rückgabe)Wert (engl. *(return) value*) einer LF

Rahmen (engl. *frame*): analog einem Emacs-frame, der ein abgegrenztes (eigenes) Bildschirmfenster darstellt, enthält ein Rahmen eine Instanz des Programms

Puffer(bereich) (engl. *buffer*): analog einem Emacs-Buffer, der eine zu bearbeitende Datei enthält, die in ein Bildschirmfenster eingeblendet werden kann, enthält ein Buffer einen Eintrag, der in einem Bildschirmfenster dargestellt wird

token: eine abgrenzbare Bereichsüberschrift innerhalb eines Eintrags, z. B. `LEX:`,
Definition:

Wörterbuch (engl. *dictionary*) : eine Ansammlung von Wörterbuchartikeln

2 Erklärende Kombinatorische Wörterbücher

2.1 Übersicht

Wie eingangs erwähnt (siehe Kapitel. 1.1, S. 1), unterscheidet sich ein EKW von einem herkömmlichen Wörterbuch zum einen durch die größere Breite der erfaßten Informationen und zum anderen durch eine stärkere Formalisierung der Repräsentation der Informationen.

Gemäß [MelcukWanner1996] wird für jedes *Lexem* (vereinfacht: ein zu beschreibendes Wort) ein *Eintrag* im Wörterbuch vorgenommen. Jeder Eintrag besteht hierbei aus drei Hauptbereichen, die im Nachfolgenden näher erklärt werden:

- einer *semantischen Zone* (engl. *semantic zone*), die die (semantische, bedeutungsmäßige) Definition des Lexems enthält,
- einer *syntaktischen Zone* (engl. *syntactic zone*), die sein Valenzschema (engl. *government pattern*) enthält, sowie
- einer *Zone lexikalischer Ko-Occurrenz* (engl. *lexical co-occurrence zone*) im speziellen Fall syntagmatischer LFs bzw. einer Zone der (auch paradigmatischen) LFs⁶, der die lexikalischen Funktionen, die auf das Lexem anwendbar sind, sowie die möglichen (Rückgabe)Werte (engl. *values*) angibt.

Im folgenden gehe ich auf die Bedeutung dieser Bereiche näher ein und werde im Kapitel 3 (S. 9) mögliche Verallgemeinerungen durch einen intelligenten Editor motivieren.

2.2 Der semantische Bereich: die Definition

Die Definition eines Lexems legt die Bedeutung desselben anhand einer “semantischen Dekomposition”, die sich auf andere Lexeme stützt, fest, und codiert auch die Positionen der semantischen Aktanten, die mit X, Y, Z, ... bezeichnet werden.

Eine solche Definition ist beispielsweise für das englische Verb [to] HELP (siehe [MelcukWanner1996], S. 214, wobei “=” als “:=” gelesen werden sollte):

X helps Y to Z with W = 'Y trying to do or doing Z, X uses X's resources W, adding W to Y's efforts such that W causes that doing Z becomes possible or easier for Y'.

Im vorliegenden Wörterbuch sieht der Bereich der Definition beim Lexem LIEBE beispielsweise so aus:

DEFINITION: *Liebe von X zu Y wegen Z 'X's love towards Y because of Z'*

Diese Form der Definition ist — im Gegensatz zu Definitionen in traditionellen EKW mit ihren propositionalen (satzähnlichen) Formen — bereits stark formalisiert. Trotzdem genügt diese Modellierung noch nicht den Anforderungen an die Einträge und deren Definitionen, die im Hinblick z. B. auf automatische Generalisierung von Einträgen gestellt werden.

Da die in einer semantischen Zone enthaltene Information umfangreich sein kann, wird zu Zwecken der “Standardisierung” oder auch “Diskretisierung” bei [MelcukWanner1996],

⁶näheres hierzu siehe [Wanner1996], S. 3 bzw. S. 12

S. 216ff, die Notation der *semantischen Dimensionen* (engl. *semantic dimensions*) eingeführt. Hierbei werden einige wenige (nützlich und charakteristisch erscheinende) semantische Dimensionen festgelegt, die jeweils zwei bis drei (sich wechselseitig ausschließende) Wertausprägungen — die semantischen Merkmale (engl. *semantic features*) — haben können. Ein Eintrag wird daraufhin durch das Vorhandensein einiger dieser Werte charakterisiert bzw. “in aller Kürze” definiert. Dies hat den Vorteil, daß Einträge im Hinblick auf z. B. Generalisierungen mit geringerem (z. B. kombinatorischem) Aufwand verarbeitet werden können, indem eine Art Vorauswahl oder Vor-Klassifizierung vom Lexikographen durchgeführt wird.

Beispielsweise gilt für das Lexem WUT und eine semantische Dimension ‘directionality’ (Gerichtetheit) mit den möglichen Ausprägungen { ‘directed-at’, ‘directionality-neutral’⁷ }, daß *Wut* ‘gerichtet-auf’ ist, da sie sich zumeist auf ein bestimmtes Objekt bezieht.

Da eine geringe Anzahl semantischer Dimensionen mit nur jeweils zwei bis drei Ausprägungen gewählt werden kann, ist eine übersichtliche tabellarische Darstellung der Form SCHLÜSSELLEXEM × semantische Dimension → ‘SD-Wert’ bzw. SD: LEX → ‘SD-Wert’ möglich; Tabelle 1 zeigt hierzu ein Beispiel.

Tabelle 1: *Beispiel für semantische Dimensionen*⁸

Schlüssellexem	Dimension										
	1	2	3	4	5	6	7	8	9	10	11
ACHTUNG		+	–	+	+		+	+			+
ANGST		–	–	+		+		+	+	+	±
ÄRGER		–	–	+		+			+		–
AUFREGUNG		–	+						+		–
BEDAUERN		–		+							±
BEGEISTERUNG	+	+	+					+	+	+	–
...

Legende:

1. Intensity: ‘intense’ (‘+’), ‘moderate’ (‘–’), ‘intensity-neutral’ (‘(leer)’) — Intensität: ‘stark’, ‘gemäßigt’ bzw. ‘Intensitäts-neutral’
2. Polarity: ‘pleasant’ (‘+’), ‘unpleasant’ (‘–’), ‘polarity-neutral’ (‘(leer)’) — Polarität: ‘erfreulich’, ‘unerfreulich’ bzw. ‘Polaritäts-neutral’
3. Manifestability: ‘manifested’ (‘+’), ‘manifestable’ (‘–’), ‘manifestation-neutral’ (‘(leer)’) — Manifestation: Gefühlslexem ist bei seinem Auftreten (z. B. in der Mimik oder Handlung) manifestiert, manifestierbar bzw. hat eine neutrale Manifestation
4. ... (weitere Beispiele ausgelassen)

Im vorliegenden Wörterbuch sieht der Bereich der semantischen Merkmale beim Lexem LIEBE beispielsweise so aus:

FEATURES: from_X, intense, pleasant, manifestable, attitudinal, approving, excited-state, permanent, Gefühl, caused_by_Y's_Z, directed_at_Y

⁷mit ‘... neutral’ ist immer “keine Aussage” (bezüglich dieser Dimension) assoziiert

2.3 Der syntaktische Bereich: das Valenzschema

Das Valenzschema (engl. *government pattern*, GP) eines Lexems gibt den Zusammenhang zwischen den semantischen Aktanten (benannt mit den Variablen X, Y, Z, ...) und den tiefsyntaktischen Aktanten (engl. *deep-syntactic actants*)⁹ (benannt mit den Variablen I, II, III, ...) an.

Gemäß [MelcukWanner1994] wird das Valenzschema durch eine Matrix repräsentiert, die aus m Spalten und n Zeilen besteht, wobei m die Anzahl der semantischen Aktanten des Lexems und n die maximale Anzahl der unterschiedlichen *surface means* für den Ausdruck eines syntaktischen Aktanten sind.

Als Beispiel wird in Tabelle 2 das Valenzschema für das Verb [to]HELP (im obigen Sinne *X helps Y to Z with W*) angegeben.

Tabelle 2: *Beispiel für ein Valenzschema*¹⁰

X = I	Y = II	Z = III	W = IV
1. N	1. N	1. V _{inf} 2. to V _{inf} 3. with N 4. PREP _{dir} N	1. with N 2. by V _{ger}

Im vorliegenden Wörterbuch sieht der Bereich beim Lexem LIEBE beispielsweise so aus:

VALENZ: X = I = N_{gen}, von N_{dat}, PRON_{poss}
 Y = II = zu N_{dat}, gegenüber N_{dat}
 Z = III = wegen N_{gen}

2.4 Der Bereich der lexikalischen Funktionen

Das Konzept der lexikalischen Funktionen LF (engl. *lexical functions*) ist beschrieben bei [Melcuk1996]. Formalisiert ausgedrückt ist eine lexikalische Funktion LF_i eine Relation zwischen einem gegebenen lexikalischen Ausdruck X (engl. *lexical expression*) als Argument (Schlüsselwort, engl. *argument*, *keyword*) und einer Menge {Y₁, Y₂, ..., Y_n} (weiterer) lexikalischer Ausdrücke, die als der (Rückgabe)Wert (engl. *value*) von LF_i bezeichnet werden, wobei abhängig von X mit LF_i eine spezifische Bedeutung assoziiert wird: LF_i : X → {X_j}

Insgesamt wurden 64 solcher lexikalischer Funktionen von Mel'čuk definiert. Mögliche Bedeutungen sind beispielsweise "Synonymie" (Syn, z. B. Syn(*radfahren*) = { *Fahrrad fahren* }) oder "Verbform" (V₀, z. B. V₀(*Schlag*) = { *schlagen* }).

Im vorliegenden Wörterbuch sieht der Bereich beim Lexem LIEBE beispielsweise so aus:

LEX. FUNKTIONEN:

Oper₁: | requires a modifier,
 empfinden, fühlen [*acc*],
 entgegenbringen [N_{dat} DET *acc*]

⁹ein Begriff aus Mel'čuk's MTT, siehe [Melcuk1988]

Magn + IncepFact₁: überwältigen[N_{acc}]
 IncepPredMinus: nachlassen
 Liqu₁Func₀: überwinden [PRON_{poss}/DET_{acc}]
 Liqu₁Fact₀: unterdrücken [PRON_{poss}/DET_{acc}]

[Melcuk1996], S. 47 bzw. S. 56, und [Wanner1996], S. 3ff bzw. S. 12ff, unterscheiden ferner zwischen paradigmatischen bzw. syntagmatischen LF_s. Erstere bezeichnen alle konträren und Substitutionsbeziehungen, die zwischen lexikalischen Einheiten in bestimmten Kontexten bestehen können — Beispiele hierfür sind die Relationen “Synonymie” (SYN), Antonymie (ANTI) und Konversion (CONV). Letztere bezeichnen Beziehungen LF_{*i*} zwischen Lexemen L₁ und L₂ in der Art, daß L₁ und L₂ in Texten auch kookkurieren, d. h. (meistens) gemeinsam im selben Satz auftauchen — Beispiele hierfür sind die Relationen “Größe” (Magn), “mehr” (Plus), “gut” (Bon), konkret z. B. Magn(*Geduld*) = *unendliche*, IncepPredPlus(*Temperatur*) = { *steigt*, *nimmt zu* }, Bon(*Hilfe*) = *wertvolle*¹¹. Letztere Beziehungen sind auch diejenigen, die für Generalisierungen in Frage kommen.

Die Indizes der Bezeichner von lexikalischen Funktionen beziehen sich auf Aktanten: Oper₁ bezieht sich also z. B. auf die Verwendung der lexikalischen Funktion Oper_{*i*} mit einem spezifischen Schlüsselwort auf den *ersten* Aktanten, Func₃ auf den *dritten*, etc. Es gilt beispielsweise Oper₁(*Bestellung*) = *geben*, Oper₂(*Bestellung*) = { }, Oper₃(*Bestellung*) = *empfangen* (siehe [Melcuk1996], S. 61).

¹¹siehe [Melcuk1996], S. 56f

3 Überprüfungen und Generalisierungen

3.1 Syntaktische Überprüfungen

3.1.1 Motivation

Jeder Wörterbucheintrag sollte einer bestimmten Struktur folgen. Hiervon abweichende Einträge benutzen z. B. fehlerhafte tokens, besitzen undefinierte Felder, oder es werden “ungültige” LFids oder Bezeichner von semantischen Merkmalen verwendet.

Weiterhin können beispielsweise in Funktionsbezeichnern codierte Aktanzahlen automatisiert mit der Anzahl der Aktanten im Valenzschema abgeglichen und so Fehler vermieden werden. Beispielsweise kann die lexikalische Funktion `Func3` innerhalb eines Lemmas nicht verwendet werden, wenn das zum Lemma gehörige Valenzschema nur *zwei* Aktanten spezifiziert!

Solche syntaktischen Fehler erschweren die automatische Verarbeitung von Einträgen oder verhindern diese gar ganz; sie implizieren eine undefinierte Semantik. Außerdem sind syntaktische Fehler natürlich auch immer ein Hinweis auf “völlig fehlerhafte” Einträge, die auch für den Benutzer nicht zu gebrauchen sind.

3.1.2 Resultate

Im Zuge der Entwicklung der Software und einer damit einhergehenden Festlegung der grammatikalischen Struktur von Einträgen (tokens, etc.) sowie dem Einlesen derselben und deren Verteilung auf interne Datenstrukturen konnten als “Seiteneffekt” vom Benutzer verursachte Inkonsistenzen aufgedeckt werden.

fehlerhafte Eintragsstruktur Bei fehlenden oder falsch geschriebenen tokens warnt `TokenEntry` bereits während des Ladevorgangs oder bei Neueingabe und Änderung (“update”) beim Parsen. Eingabefehler dieser Art kamen z. B. vor bei den Einträgen zu den Lemmata `ANKLAGE 1B` (ohne `LEX.FUNKTIONEN:-token`), `LOB` (`DEFINTION:-` statt `DEFINITION:-token`), `VERBOT 2` (doppelter `LEX:-`Eintrag) und anderen.

Ein schwerwiegender Fehler ist ein nicht erkennbares `LEX:-token`, denn dieses wird benötigt, um einen Eintrag eindeutig (per Lexem) zu benennen.

fehlerhafte Feldeinträge Innerhalb der Suchfunktionen können explizit die bereits vorhandenen Werte für Lexeme, Funktionsbezeichner, Funktionswerte und semantische Merkmale als scrollbare, sortierte Liste angezeigt werden. Hier werden viele Fehleingaben und Inkonsistenzen offenbar, die teilweise allerdings auch noch auf eine fehlerhafte Eintragsstruktur (siehe vorhergehender Absatz) zurückgeführt werden können, der für eine falsche Zuordnung der Werte verantwortlich war; beispielsweise kann ein fehlendes `LEX.FUNKTIONEN:-token` dafür verantwortlich sein, daß lexikalische Funktionen als semantische Merkmale interpretiert werden, falls die Zone der semantischen Merkmale der (intendierten) Zone der lexikalischen Funktionen voranging. Solche Fehler können zurückverfolgt werden, indem der entsprechende, als falsch angenommene Wert angeklickt wird; daraufhin wird der oder werden die Einträge angezeigt, die das entsprechende semantische Merkmal, den Bezeichner der lexikalischen Funktion oder den Funktionsrückgabewert enthalten.

Beispiele für gefundene fehlerhafte oder inkonsistente Einträge sind z. B.

- bei den semantischen Merkmalen: ‘self-control-loss inflicting’ sowie ‘self-control-loss-inflicting’ und ‘speech act’ sowie ‘speech-act’ — Einträge, die bei einer späteren Generalisierung hinderlich gewesen wären —, sowie
- bei den Bezeichnern lexikalischer Funktionen: “(geh.)” (durch den Eintrag “(geh.):”) versehentlich von TokEntry als Bezeichner einer lexikalischen Funktion mißinterpretiert), “? Magn” (wohl fraglicher Eintrag), “AntiMAGn” (statt “AntiMagn”) und weitere, sowie
- bei den Rückgabewerten von Funktionen: wiederum mehrere mit Fragezeichen, aber auch *anstrengen/ einbringen/ einreichen ...* (falsche Trennzeichen) und weitere.

Unwahrscheinliche Anzahl von Aktanten Über das Suchmenü (Ausdruckssuche) kann nach Anzahlen von Aktanten gesucht werden, die als unwahrscheinlich angenommen werden, z. B. # > 4.

Inkonsistenz zwischen Valenzschema und Funktionsbezeichnern Über das Prüfmenü kann nach solchen Inkonsistenzen gesucht werden; beispielsweise wurde beim Lexem ANKLAGE 1A mit drei Aktanten die Verwendung der lexikalischen Funktion Func₄ angemahnt; gleiches gilt für das Lemma des Lexems BITTE (zwei Aktanten) und die hierin verwendete lexikalische Funktion AntiReal₃, für VERBOT (zwei Aktanten) und Real₃ und AntiReal₃ sowie VORSCHRIFT (ein Aktant) und A₂.

Diese Art der Überprüfung kann nach drei Kriterien ausgelöst werden: auf einen einzelnen Eintrag bezogen, auf eine einzelne lexikalische Funktion (und alle Einträge, die diese Funktion verwenden) bezogen, sowie auf den gesamten Datenbestand des aktuellen Wörterbuchs.

3.2 Semantische Überprüfungen

3.2.1 Motivation

Semantische Überprüfungen der Einträge können sich auf falsche Valenzschemata und falsch eingetragene Kollokationen beziehen. Des weiteren sind bestimmte Kombinationen von semantischen Merkmalen nicht sinnvoll möglich.

Dies ist automatisiert schwer einzuschätzen; es kann immer zu Ausnahmen von irgendwelchen Annahmen kommen (wie z. B. der Annahme einer starken “einseitigen” Korrelation zwischen den Merkmalen ‘excited-state’ (Erregtheit einer Person) und zwei Werten der Dimension “Manifestiertheit” (zum-Ausdruck-Kommen dieser Erregtheit) bei Gefühlslexemen (siehe z. B. [MelcukWanner1996], S. 222): Alle ‘excited-state’-Lexemesind (mit wenigen Ausnahmen) ‘manifestable’ oder ‘manifested’; die Umkehrung gilt allerdings nicht. Wenn also ein Benutzer ein Lexem durch ‘excited-state’ klassifizieren würde, so dürfte (mit einer entsprechenden Wahrscheinlichkeit) *nicht* das semantische Merkmal ‘manifestation-neutral’ (d. h. der dritte mögliche Wert der semantischen Dimension “Manifestiertheit”) auftreten. Speziell bei neu einzugebenden Datenbeständen wird es aber schwierig sein, hier von sinnvollen initialen Annahmen auszugehen.

In der vorliegenden Studienarbeit werden im Hinblick auf semantische Überprüfungen nur Generalisierungen untersucht, die im nachfolgenden näher untersucht werden.

3.3 Generalisierungen

3.3.1 Motivation

Generalisierungen (Verallgemeinerungen) sind inhaltlich zu den semantischen Überprüfungen zu rechnen (siehe vorhergehender Abschnitt), da sie Aussagen über die Bedeutung von Wörterbucheinträgen ermöglichen. Sie sollen zum einen

- eine Informationsverdichtung (*Kompression*) ermöglichen, und sie sollen zum anderen
- die Struktur der Wörterbucheinträge (Gemeinsamkeiten zwischen Einträgen, *Klassifizierung*) augenscheinlicher machen.

Allgemeine semantische Überprüfungen der vorgenannten Art machen erst dann Sinn, wenn nach einem initialen Generalisierungslauf ein Klassifikationsmechanismus bereitsteht, der als Grundlage für Überprüfungen zum Zwecke des Erkennens semantischer Fehler dient. Beispielsweise kann die Verwendung einer bestimmten lexikalischen Funktion (d. h. ihres Funktionsbezeichners oder auch eines bestimmten Rückgabewertes) erst dann als “möglicherweise fehlerhaft” angemahnt werden, wenn bekannt ist, mit welcher Wahrscheinlichkeit diese lexikalische Funktion in einem Lemma (beispielsweise abhängig von den dort verwendeten semantischen Merkmalen) auftritt.

3.3.2 Vorüberlegungen

[MelcukWanner1996], speziell S. 226 und 236, versuchen eine Generalisierung der Einträge von Lexemen über eine vermutete Korrelation zwischen semantischen Merkmalen und Werten lexikalischer Funktionen. Sie beschränken sich hierbei auf 40 Gefühlslexeme (speziell: Substantive) der deutschen Sprache sowie 25 LFvalues (Verben); es handelt sich also um Kollokationen von LF-Verben und Substantiven. Ihre Vorgehensweise ist wie folgt:

Zunächst spezifizieren sie Wörterbuch-Einträge für jene 40 Substantive, die in ihrer LF-Zone dann u. a. die entsprechenden LFvalues nennen. Sie geben sie eine Matrix von Schlüssellexemen und semantischen Dimensionen an (siehe ebendort, S. 218); die Komponenten dieser Matrix sind die semantischen Merkmale (siehe Tabelle 1, S. 6). Aus dieser Matrix lassen sich durch spaltenweisen Vergleich direkt Korrelationen zwischen Paaren semantischer Merkmale ablesen.

In einer zweidimensionalen Gegenüberstellung der Schlüssellexeme auf der einen und ausgewählter LFvalues auf der anderen Achse (siehe Tabelle 3), wird mittels eines Flags (ein “boolesches” Kreuzchen) das Vorhandensein des entsprechenden LFvalues beim entsprechenden Lexem angezeigt. Durch Kenntnis der Gesamtanzahl der Lexeme und Aufsummieren der Anzahl der Flags lassen sich so Häufigkeiten von LFvalues bestimmen.

Nach dem Vergleich der Häufigkeiten kann oftmals eine Korrelation zwischen semantischen Merkmalen und LFvalues abgeleitet werden.

Daraufhin werden die in diesen beiden Matrizen enthalten Informationen ausgewertet, gemeinsame LFvalues extrahiert und in dem Eintrag des generischen Lexems unter der Angabe der semantischen Merkmale, die mit diesem LFvalue korrelieren, spezifiziert, d. h. die LFvalues werden zum generischen Lexem des Gebiets (hier: das Gebiet der Gefühlslexeme und das Lexem GEFÜHL) verschoben und dort konzentriert. Dadurch wird die Redundanz gewisser LFs reduziert — beispielsweise wird $Oper_1(L) = empfinden$ für alle Lexeme L beim

Tabelle 3: Beispiel für eine Lexem \times LFvalue-Matrix¹²

LFid \rightarrow	1					2			3
Schlüssellexem \downarrow	val _a	val _b	val _c	val _d	val _e	val _f	val _g	val _h	val _i
Achtung	x	x	x	x	x		x		
Angst	x		x	x			x		x
Ärger	x								
Aufregung						?		x	x
...

Legende:

- 1 = Oper₁, 2 = IncepOper₁, 3 = CausOper₁
- val_a = empfinden, val_b = entgegenbringen, val_c = fühlen, val_d = haben, val_e = hegen, val_f = ausbrechen, val_g = bekommen, val_h = geraten, val_i = versetzen

generischen Eintrag notiert. Die im Eintrag des generischen Lexems angegebenen LFvalues werden an die Lexeme mit den entsprechenden selben semantischen Merkmalen *vererbt*. Etwaige Ausnahmen dieser Generalisierung werden bei den individuellen Einträgen vermerkt, beispielsweise ein Hinweis beim Lexem ENTZÜCKEN, daß dieses (als Wert der lexikalischen Funktion Oper₁) *nicht empfinden* haben könne.

Hierzu wird die Notation der *semantischen Bedingungen* (engl. *semantic conditions*) (siehe ebendort, S. 233, S. 236 und S. 238) eingeführt: Mittels einer *booleschen Formel über Werte von semantischen Merkmalen* wird festgelegt, ob eine bestimmte lexikalische Funktion mit einem bestimmten Wert beim generischen oder den spezifischen Einträgen zur Anwendung kommen kann. Dadurch soll die Redundanz im Datenbestand vermindert werden.¹³

Beispielsweise können solche generischen Einträge vereinfacht so aussehen:

Oper₁: empfinden, fühlen, entgegenbringen
Magn + IncepOper₁: geraten | ‘manifested’
ausbrechen | ‘intense’ \wedge ‘manifested’
Liqu₁Func₀: überwinden | \neg ‘moderate’

Der Beispieleintrag besagt, daß die lexikalische Funktion Oper₁ beim betrachteten Datenbestand den Rückgabewert { *empfinden, fühlen, entgegenbringen* } hat. In anderen Worten: Alle Lemmata (sofern sie in ihren eigenen Einträgen selbst keine Ausnahme hiervon definieren) enthalten in ihrem LF-Bereich Oper₁ mit dem entsprechenden Rückgabewert, und zwar unabhängig von den Werten der semantischen Dimensionen.

Er besagt weiterhin, daß die Kombination von Magn + IncepOper₁ beim Auftreten des semantischen Merkmals ‘manifested’ beim Schlüsselwort (d. h. dem Argument der LF) zum Rückgabewert *geraten* führt. In anderen Worten: Wenn in einem Lemma das semantische Merkmal ‘manifested’ auftritt, kann auf das Vorhandensein der lexikalischen Funktion Magn + IncepOper₁ mit dem Rückgabewert *geraten* gefolgert werden.

Des weiteren kann beim Vorhandensein der semantischen Merkmale ‘intense’ und ‘manifested’ von einem Rückgabewert *ausbrechen* bei Magn + IncepOper₁ (unabhängig vom Lexem) ausgegangen werden, etc.

¹³Dieses Konzept der semantischen Bedingungen wird zusätzlich auch auf das Valenzschema angewendet.

3.3.3 Das Auffinden einfacher Korrelationen

Eine “einfache” Korrelation sei eine Korrelation zwischen nur *einem* semantischen Merkmal und einem hiermit korrelierenden Wert.

Das Auffinden einfacher Korrelationen zwischen der Semantik von Schlüsselwörtern (d. h. den semantischen Merkmalen) und deren Kollokationen (grob gesagt: den LFvalues der diese semantischen Merkmale erfüllenden Lemmata) ist ein Optimierungsprozeß — siehe [MelcukWanner1996], S. 231: (übersetzt) “Beginnend mit diesen Daten [semantischen Merkmalen und LFvalues] versuchen wir, die optimale Korrelation zwischen den LFvalues und semantischen Merkmalen in den verkürzten Definitionen zu finden. (Mit “optimal” meinen wir eine solche Korrelation, die die maximale Generalisierung mit einer minimalen Anzahl individueller Ausnahmen sicherstellt, das heißt die bestmögliche Informationskompression.)”¹⁴

Um eine Idee davon zu vermitteln, wie eine solche Optimierung aussehen könnte und um mit einer bestimmten Notation vertraut zu machen, gebe ich hier ein Beispiel, das sich auf den von Mel’čuk und Wanner verwendeten Datenbestand von 40 Gefühlslexemen des Deutschen bezieht:

Sei M_k die Menge aller Schlüsselwörter (keys) eines Lexikons. Sei lf ein Bezeichner (LFid) oder Wert (LFvalue) einer lexikalischen Funktion; sei M_{lf} die Menge der Schlüsselwörter, die lf in ihren Einträgen (genauer: in der Zone der lexikalischen Funktionen) verwenden¹⁵.

Ich fasse LFid und LFvalue unter dem Begriff lf zusammen, weil ich davon ausgehe, daß sowohl zwischen semantischen Merkmalen und LFvalues als auch zwischen semantischen Merkmalen und LFids Korrelationen gefunden werden können — ersteres ist durch [MelcukWanner1996] belegt, letzteres ist ein Spezialfall des ersteren: Wenn zwischen einem LFvalue und einer semantischen Bedingung eine Korrelation besteht, dann impliziert dies (unter der Annahme einer eindeutig zuordenbaren LFid) auch eine Korrelation zwischen derselben semantischen Bedingung und einer LFid. Mehr noch: Es werden wohl eher Korrelationen mit LFids als mit LFvalues gefunden werden können, weil es deutlich weniger LFids als LFvalues gibt und jedem LFid in Abhängigkeit vom Lexem mehrere LFvalues als zurückgegebener (Mengen)Wert zugeordnet werden können.

Sei f ein semantisches Merkmal (feature); sei M_f die Menge der Schlüsselwörter, die f in ihren Einträgen (genauer: in der Zone der Definition) verwenden.

Wählt man speziell $lf = \text{geraten}$ (als Wert von IncepOper_1 , angewendet auf alle Lexeme, in deren Einträgen diese LF definiert ist; siehe Tabelle 3, S. 12) und $f = \text{‘manifested’}$ (siehe Tabelle 1, S. 6), dann ergeben sich die beiden folgenden Mengen, die zur leichteren Abzählbarkeit der Schlüsselwörter indiziert sind, wobei sich der Index an einer alphabetischen Ordnung der Elemente der Vereinigungsmenge $M_{lf} \cup M_f$ orientiert:

$$M_{lf} = \{ \text{AUFREGUNG}_1, \text{BEGEISTERUNG}_2, \text{ENTZÜCKEN}_4, \text{ERREGUNG}_5, \text{FREUDE}_6, \\ \text{PANIK}_7, \text{STAUNEN}_9, \text{VERLEGENHEIT}_{10}, \text{VERZWEIFLUNG}_{11}, \text{WUT}_{12}, \text{ZORN}_{13} \}$$

$$M_f = \{ \text{AUFREGUNG}_1, \text{BEGEISTERUNG}_2, \text{ENTSETZEN}_3, \text{ENTZÜCKEN}_4, \text{FREUDE}_6, \text{PANIK}_7, \}$$

¹⁴eine weniger wissenschaftliche Abhandlung hierzu siehe bei [Mintert1998]

¹⁵Im folgenden sei aus Effizienzgründen und aus Gründen der leichteren Lesbarkeit immer angenommen, daß ein LFvalue eindeutig einer (einzigen) LFid zugeordnet werden kann, z. B. immer $\text{Oper}_1(l) = \text{machen}$, d. h. “machen” tritt immer nur zusammen mit Oper_1 auf, nie mit anderen lexikalischen Funktionen. Sollte diese Eindeutigkeit einmal nicht mehr gewahrt sein, so kann durch eine zusätzliche Unterscheidung von LFid und LFvalue (durch eine Konjunktion bzw. einer Schnittbildung über Mengen oder eine textuelle Konkatenation beider Teilstrings) in den nachfolgenden Abhandlungen die Eindeutigkeit wieder hergestellt werden.

SCHEU₈, STAUNEN₉, VERZWEIFLUNG₁₁, WUT₁₂, ZORN₁₃ }
(und $|M_K| = 40$)

Eine optimale Korrelation von lf und f würde sich ergeben bei $M_{lf} \cap M_f = M_{lf} = M_f$; daraus könnte “hundertprozentig” die semantische Bedingung $lf|f$ gefolgert werden.

$$p := \frac{\text{Setze } |M_{lf} \cap M_f| \times 100}{|M_{lf} \cup M_f|}$$

Im konkreten Beispiel sind $|M_{lf} \cup M_f| = 13$ (siehe größter Index) und $|M_{lf} \cap M_f| = |\{ \text{AUFREGUNG}_1, \text{BEGEISTERUNG}_2, \text{ENTZÜCKEN}_4, \text{FREUDE}_6, \text{PANIK}_7, \text{STAUNEN}_9, \text{VERZWEIFLUNG}_{11}, \text{WUT}_{12}, \text{ZORN}_{13} \}| = 9$.

Man erhält so:

$$p = \frac{9 \times 100}{13} \approx 69.2 \text{ Prozent}$$

Das bedeutet daß die relative Wahrscheinlichkeit für das Auftreten sowohl von $lf = \text{geraten}$ als auch von $f = \text{‘manifested’}$ etwa zwei Drittel beträgt — relativ zum Auftreten des semantischen Merkmals f .

Zur Auswahl der relevanten semantischen Merkmale muß nun noch ein Schwellwert t (engl. *threshold*) definiert werden: Sobald $p \geq t$ gilt, wird eine hinreichende Korrelation angenommen. Geeignete Werte für t müssen empirisch ermittelt werden.

3.3.4 Das Auffinden komplexerer Korrelationen

Eine “komplexe” Korrelation sei eine Korrelation zwischen einer booleschen Formel über semantischen Merkmalen und einem weiteren damit korrelierenden Wert.

Eine solche boolesche Formel über semantische Merkmale kann wie folgt exemplarisch abgeleitet werden:

Gegeben seien zwei Mengen M_{f_1} bzw. M_{f_2} , die die Schlüsselwörter aller Einträge enthalten, die zwei bestimmte semantische Merkmale f_1 bzw. f_2 in ihrer Definition aufweisen. Ferner sei eine Menge M_{lf} gegeben, die die Schlüsselwörter aller Einträge enthalten, die eine bestimmte LFid oder einen bestimmten LFvalue lf enthalten.

Konjunktion Sei M_{f_\cap} die Menge $M_{f_\cap} := M_{f_1} \cap M_{f_2}$, und sei p ein Prozentsatz.

Die Zuweisung

$$p := \frac{|M_{f_\cap} \cap M_{lf}| \times 100}{|M_{f_\cap} \cup M_{lf}|} \quad (1)$$

ermittelt die Wahrscheinlichkeit p für das Auftreten von f_1 und f_2 (Konjunktion) zusammen mit lf . Überschreitet p einen Schwellwert t , so enthalten M_{f_\cap} und M_{lf} “nahezu dieselben Elemente”. Daraus kann eine positive Korrelation von f_1 und f_2 bezüglich lf vermutet werden, und es bietet sich an, immer beim Auftreten von f_1 und f_2 auf das Auftreten von lf zu schließen — d. h. die semantische Bedingung ist $f_1 \wedge f_2$.

Disjunktion Sei M_{f_\cup} die folgende Menge:

$$M_{f_\cup} := M_{f_1} \cup M_{f_2}$$

Die Zuweisung

$$p := \frac{|M_{f \cup} \cap M_{lf}| \times 100}{|M_{f \cup} \cup M_{lf}|} \quad (2)$$

ermittelt die Wahrscheinlichkeit p für das Auftreten von f_1 *oder* f_2 (Disjunktion) zusammen mit lf . Überschreitet p einen Schwellwert t , so enthalten $M_{f \cup}$ und M_{lf} “nahezu dieselben Elemente”. Daraus kann eine positive Korrelation von f_1 und f_2 bezüglich lf vermutet werden, und es bietet sich an, immer beim Auftreten von f_1 *oder* f_2 auf das Auftreten von lf zu schließen — d. h. die semantische Bedingung ist $f_1 \vee f_2$.

Die beiden Prozentsätze (1) oder (2) können Analog für Mengen $M_{f \cap}$ und $M_{f \cup}$ mit mehrstelliger Durchschnitts- und Vereinigungsbildung ermittelt werden.

Negation Sei $\neg M$ die folgende Menge:

$$\neg M_f := \neg M_{f_1} := M_k \setminus M_{f_1},$$

d. h. das Komplement von M_{f_1} bezüglich der Menge aller Schlüssel M_k .

Die Zuweisung

$$p := \frac{|\neg M_f \cap M_{lf}| \times 100}{|\neg M_f \cup M_{lf}|} \quad (3)$$

läßt analog dem vorher gesagten auf eine entsprechende Korrelation des Nicht-Auftretens von f bei lf schließen (Negation). Durch Kombination mit den Ausdrücken in (1) und (2) lassen sich so entsprechend komplexe boolesche Ausdrücke bilden. Da das logische Und, Oder und Nicht eine Junktorbasis bilden und speziell durch disjunktive Formeln alle Wertkombinationen von semantischen Merkmalen abgedeckt werden, ist dadurch eine umfassende Beschreibungsmöglichkeit gefunden worden; über die Minimalität der erzeugten Formeln im Hinblick auf ihre Länge ist allerdings noch keine Aussage gemacht.

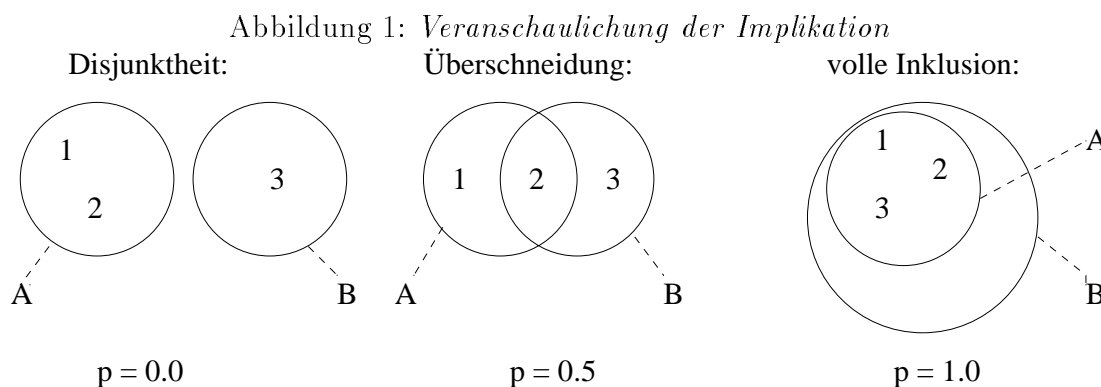
Implikation Auf einen weiteren Fall soll noch eingegangen werden: Die Zuweisung

$$p := \frac{|M_{f_1} \cap M_{f_2}| \times 100}{|M_{f_1}|} \quad (4)$$

ermittelt die “Wahrscheinlichkeit”, mit der M_{f_1} in M_{f_2} enthalten ist. Dies ist gleichbedeutend mit der Wahrscheinlichkeit, daß vom Vorhandensein von f_1 auf das Vorhandensein von f_2 gefolgert werden kann.

Ein Beispiel:

- Disjunktheit: $M_{f_1} = \{1, 2\}$, $M_{f_2} = \{3\}$; dann gilt $p = 0/2 = 0.0$.
- partielle Überschneidung: $M_{f_1} = \{1, 2\}$, $M_{f_2} = \{2, 3\}$; dann gilt $p = 1/2 = 0.5$.
- volle Inklusion: $M_{f_1} = \{1, 2\}$, $M_{f_2} = \{1, 2, 3\}$; dann gilt $p = 2/2 = 1.0$.



Eine Veranschaulichung hiervon siehe in Abbildung 1 (S. 16).

Geeignete Schwellwerte $t \geq p$ müssen wiederum durch empirische Studien gefunden werden.

Im Folgenden sollen mehrere unterschiedliche Ansätze diskutiert werden, die das Auffinden solcher semantischer Bedingungen erlauben. Die statistischen Auswertungen (folgendes Kapitel 3.3.5) erkennen nur “fest eingebaute” Muster von Ausdrücken, während die allgemeineren Generalisierungen mittels Bäumen (ab dem übernächste Kapitel 3.3.6, S. 20) versuchen, sich (als Programm zur Laufzeit) an ein optimales Muster heranzutasten.

3.3.5 Statistische Auswertungen

Nachfolgend wird die Auswertung eines Testlaufs auf dem Datenbestand mit den 40 Gefühlswörter gegeben. Verwendet wurden einfache Algorithmen, die Korrelationen (Wahrscheinlichkeiten für Kookkurenzen) zwischen verschiedenen Merkmalen nachweisen sollten. Der Schwellwert lag bei 70 Prozent; es wurden nur LFids und LFvalues betrachtet, deren absolute Häufigkeit (bezogen auf alle Einträge) größer als zehn Prozent beträgt (d. h. vier Mal oder öfters vorkamen); bei den semantischen Merkmalen lag dieser Schwellwert für die Auswahl bei fünf Prozent (wiederum bezogen auf das Vorkommen dieses semantischen Merkmals in Einträgen).

Korrelation LFid \times semantisches Merkmal Tabelle 4 (S. 16) gibt die Wahrscheinlichkeiten, mit der Bezeichner lexikalischer Funktionen und semantische Merkmale kookkuriern, an. Hieraus können direkt einfache semantische Bedingungen angegeben werden:

Tabelle 4: *Korrelation LFid \times semantisches Merkmal*

	LFid	semantisches Merkmal f	$\frac{ M_f \cap M_l }{ M_f \cup M_l } = p\%$
1	fast FinFunc ₀	‘excited-state’	22/30 = 73.3
2	FinFunc ₀	‘excited-state’	22/28 = 78.6
3	IncepPredMinus	‘excited-state’	22/27 = 81.5
4	Magn + fast IncepFunc ₁	‘self-control-loss-inflicting’	11/14 = 78.6
5	Magn + IncepOper ₁	‘manifested’	10/13 = 76.9

zugrundeliegende Daten:

	Schlüsselwörter
1	ANGST, AUFREGUNG, BEGEISTERUNG, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, FREUDE, FURCHT, LEIDENSCHAFT, LIEBE, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
2	ANGST, AUFREGUNG, BEGEISTERUNG, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, FREUDE, FURCHT, LEIDENSCHAFT, LIEBE, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
3	ANGST, AUFREGUNG, BEGEISTERUNG, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, FREUDE, FURCHT, LEIDENSCHAFT, LIEBE, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
4	ANGST, BEGEISTERUNG, ENTSETZEN, FREUDE, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN
5	AUFREGUNG, BEGEISTERUNG, ENTSETZEN, ENTZÜCKEN, FREUDE, PANIK, STAUNEN, VERZWEIFLUNG, WUT, ZORN

Korrelation LFvalue × semantisches Merkmal Die Tabelle 5 (S. 17) ist analog zur vorhergehenden, jetzt allerdings für Rückgabewerte lexikalischer Funktionen und semantische Merkmale:

Tabelle 5: *Korrelation LFvalue × semantisches Merkmal*

	LFvalue	semantisches Merkmal f	$\frac{ M_{lf} \cap M_f }{ M_{lf} \cup M_f } = p\%$
1	<i>packen</i> [N_{acc}]	‘self-control-loss-inflicting’	11/13 = 84.6
2	<i>geraten</i> [in_{acc}]	‘manifested’	9/10 = 90.0
3	<i>entgegenbringen</i> [$N_{dat} DET_{acc}$]	‘attitudinal’	6/6 = 100.0

zugrundeliegende Daten:

	Schlüsselwörter
1	ANGST, BEGEISTERUNG, ENTSETZEN, FREUDE, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN
2	AUFREGUNG, BEGEISTERUNG, ENTZÜCKEN, FREUDE, PANIK, STAUNEN, VERZWEIFLUNG, WUT, ZORN
3	ACHTUNG, HASS, LIEBE, MITLEID, VERACHTUNG, ZUNEIGUNG

Korrelationen semantisches Merkmal ⇒ semantisches Merkmal Es wurden für den Schwellwert $p = 70$ Prozent sehr viele Korrelationen gefunden. Erhöht man p auf 80 Prozent, so erhält man die in Tabelle 6 (S. 18) genannten *Implikationen*:

zugrundeliegende Daten:

Tabelle 6: *Korrelation semantisches Merkmal \Rightarrow semantisches Merkmal*

	semantisches Merkmal f_1	semantisches Merkmal f_2	$\frac{ M_{f_1} \cap M_{f_2} }{ M_{f_1} } = p\%$
1	'temporary'	'excited-state'	16/20 = 80.0%
2	'intense'	'excited-state'	11/13 = 84.6%
3	'self-control-loss-inflicting'	'excited-state'	10/11 = 90.9%
4	'self-control-loss-inflicting'	'active'	11/11 = 100.0%
5	'self-control-loss-inflicting'	'intense'	9/11 = 81.8%
6	'manifested'	'excited-state'	9/10 = 90.0%
7	'manifested'	'reactive'	8/10 = 80.0%
8	'manifested'	'temporary'	9/10 = 90.0%
9	'manifested'	'intense'	8/10 = 80.0%
10	'permanent_or_temporary'	'unpleasant'	8/9 = 88.9%
11	'attitudinal'	'active'	5/6 = 83.3%
12	'attitudinal'	'permanent'	6/6 = 100.0%
13	'moderate'	'mental'	3/3 = 100.0%
14	'moderate'	'manifestable'	3/3 = 100.0%
15	'moderate'	'temporary'	3/3 = 100.0%
16	'approving'	'permanent'	3/3 = 100.0%
17	'approving'	'attitudinal'	3/3 = 100.0%

	Schlüsselwörter
1	$M_{f_1} = \{ \text{AUFREGUNG, BEGEISTERUNG, EMPÖRUNG, ENTSETZEN, ENTTÄUSCHUNG, ENTZÜCKEN, ERREGUNG, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, STAUNEN, VERÄRGERUNG, VERDRUSS, VERLEGENHEIT, VERWUNDE- RUNG, VERZWEIFLUNG, WUT, ZORN, ÄRGER} \}$ $M_{f_2} = \{ \text{ANGST, AUFREGUNG, BEGEISTERUNG, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, FREUDE, FURCHT, LEIDENSCHAFT, LIEBE, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER} \}$
2	$M_{f_1} = \{ \text{BEGEISTERUNG, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, HASS, LEIDENSCHAFT, LIEBE, PANIK, SCHRECK, STAUNEN, VERZWEIFLUNG, WUT, ZORN} \}$ $M_{f_2} = \{ \text{ANGST, AUFREGUNG, BEGEISTERUNG, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, FREUDE, FURCHT, LEIDENSCHAFT, LIEBE, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER} \}$

3	$M_{f_1} = \{ \text{ANGST, BEGEISTERUNG, ENTSETZEN, FREUDE, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN} \}$ $M_{f_2} = \{ \text{ANGST, AUFREGUNG, BEGEISTERUNG, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, FREUDE, FURCHT, LEIDENSCHAFT, LIEBE, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER} \}$
4	$M_{f_1} = \{ \text{ANGST, BEGEISTERUNG, ENTSETZEN, FREUDE, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN} \}$ $M_{f_2} = \{ \text{ACHTUNG, ANGST, BEGEISTERUNG, EKEL, ENTSETZEN, FREUDE, FURCHT, HASS, LEIDENSCHAFT, MITLEID, PANIK, RÜHRUNG, SCHAM, SCHEU, SCHRECK, VERACHTUNG, VERLEGENHEIT, VERZWEIFLUNG, WUT, ZORN, ZUNEIGUNG} \}$
5	$M_{f_1} = \{ \text{ANGST, BEGEISTERUNG, ENTSETZEN, FREUDE, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN} \}$ $M_{f_2} = \{ \text{BEGEISTERUNG, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, HASS, LEIDENSCHAFT, LIEBE, PANIK, SCHRECK, STAUNEN, VERZWEIFLUNG, WUT, ZORN} \}$
...	
17	$M_{f_1} = \{ \text{ACHTUNG, LIEBE, MITLEID} \}$ $M_{f_2} = \{ \text{ACHTUNG, HASS, LIEBE, MITLEID, VERACHTUNG, ZUNEIGUNG} \}$

Korrelationen semantisches Merkmal \times semantisches Merkmal Es wurden für den Schwellwert $p = 70$ Prozent *keine* signifikanten Korrelationen gefunden! Senkt man p ab, so erhält man beispielsweise die in Tabelle 7 (S. 19) genannten wahrscheinlichen *Konjunktionen*:

Tabelle 7: *Korrelation semantisches Merkmal times semantisches Merkmal*

	semantisches Merkmal f_1	semantisches Merkmal f_2	$\frac{ M_{f_1} \cap M_{f_2} }{ M_{f_1} \cup M_{f_2} } = p\%$
1	'unpleasant'	'reactive'	17/32 = 53.1%
2	'excited-state'	'reactive'	17/27 = 63.0%
3	'excited-state'	'temporary'	16/26 = 61.5%
4	'reactive'	'temporary'	15/27 = 55.6%
5	'active'	'self-control-loss-inflicting'	11/21 = 52.4%
6	'intense'	'self-control-loss-inflicting'	9/15 = 60.0%
7	'intense'	'manifested'	8/15 = 53.3%
8	'self-control-loss-inflicting'	'manifested'	7/14 = 50.0%
9	'permanent'	'attitudinal'	6/11 = 54.5%
10	'attitudinal'	'approving'	3/6 = 50.0%

zugrundeliegende Daten:

	Schlüsselwörter
1	ANGST, EIFERSUCHT, EKEL, EMPÖRUNG, ENTSETZEN, ENTTÄUSCHUNG, FURCHT, GROLL, PANIK, SCHAM, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
2	ANGST, EIFERSUCHT, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, FREUDE, FURCHT, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
3	AUFREGUNG, BEGEISTERUNG, EMPÖRUNG, ENTSETZEN, ENTZÜCKEN, ERREGUNG, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
4	EMPÖRUNG, ENTSETZEN, ENTTÄUSCHUNG, ENTZÜCKEN, PANIK, RÜHRUNG, SCHADENFREUDE, SCHRECK, STAUNEN, VERÄRGERUNG, VERDRUSS, VERZWEIFLUNG, WUT, ZORN, ÄRGER
5	ANGST, BEGEISTERUNG, ENTSETZEN, FREUDE, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN
6	BEGEISTERUNG, ENTSETZEN, HASS, LEIDENSCHAFT, PANIK, SCHRECK, VERZWEIFLUNG, WUT, ZORN
7	BEGEISTERUNG, ENTSETZEN, ENTZÜCKEN, PANIK, STAUNEN, VERZWEIFLUNG, WUT, ZORN
8	BEGEISTERUNG, ENTSETZEN, FREUDE, PANIK, VERZWEIFLUNG, WUT, ZORN
9	ACHTUNG, HASS, LIEBE, MITLEID, VERACHTUNG, ZUNEIGUNG
10	ACHTUNG, LIEBE, MITLEID

Korrelationen semantisches Merkmal \times semantisches Merkmal \times semantisches Merkmal Hierfür wurden selbst für ein $p = 50$ Prozent beim Datenbestand mit den 40 Gefühlslexemen *keine* Korrelationen ermittelt.

3.3.6 Ansatz 1: Entscheidungsbäume

Ein Entscheidungsbaum ist ein Hilfsmittel für die Klassifizierung von “Fällen”. Ausgegangen wird von einer Menge von Datensätzen mit einer bestimmten Struktur, die aus Attributen und zugehörigen Werten besteht, wobei eines dieser Attribute ein Zielattribut darstellt, d. h. das wichtigste Attribut, das aufgrund der Werte der anderen Attribute bei neuen Fällen “prophezeit” werden soll. Gebildet wird ein solcher Entscheidungsbaum mit einer Menge von “Trainingsdaten”.

Alle inneren Knoten eines Entscheidungsbaums sind mit einem Attribut gekennzeichnet; jeder Knoten hat sovielen “herausgehende” Kanten, wie sein zugehöriges Attribut Werte besitzt. Die äußeren Knoten (Blätter) geben den Wert des wahrscheinlichsten Zielattributs an.

Beispiele für Entscheidungsbäume finden sich in [Quinlan1992]¹⁶; eine erschöpfendere

¹⁶siehe auch eine Kurzfassung auf <http://yoda.cis.temple.edu:8080/UGAIWWW/lectures95/learn/C45/>

Definition wird in [InfoDuden1993], S. 235, gegeben.

Der Aufbau eines *optimalen* binären Entscheidungsbaumes ist NP-vollständig; siehe hierzu [HyafilRivest1976]. Demzufolge müssen geeignete, effiziente *Heuristiken* für den Aufbau gesucht werden, die eine entsprechend eingeschränkte Komplexität besitzen:

- Der von Quinlan verwendete Algorithmus ID3¹⁷ verwendet beispielsweise *kein* backtracking, entscheidet sich also aufgrund lokalen Wissens für einen Teilbaum (ein Attribut) und ändert diese Wahl später *nicht* mehr (Quinlan: “the choice is cast in concrete”; siehe [Quinlan1992], S. 20); dadurch wird u. a. eine “kombinatorische Explosion” vermieden.
- Ein einmal ausgewähltes Attribut steht später nicht mehr zur Wahl; jedes Attribut taucht höchstens einmal im Baum auf. Dies schränkt die Anzahl möglicher Kombinationen weiter ein.
- Jede Entscheidung aufgrund eines Attributwertes an einem Knoten ist von der Art entweder-oder bzw. einer switch()-Anweisung in der Programmiersprache C. Ein weiterer Abstieg Richtung Blatt entspricht einer Konjunktion (logisches Und) der Attributwerte entlang des Pfades¹⁸.

Ein Entscheidungsbaum für Generalisierungen von Wörterbucheinträgen könnte als Attribute die semantischen Dimensionen verwenden; den Werten von Attributen entsprechen dann die Werte der semantischen Dimensionen. Ein zugehöriges Zielattribut ist nicht so augenfällig — es könnten z. B. einerseits *Mengen von Schlüsselwörtern* sein, die mit größerer Wahrscheinlichkeit bei diesen semantischen Merkmalen auftreten; ein menschlicher Betrachter könnte so semantisch “verwandte” Gruppen von Lexemen identifizieren (und sie eventuell unter einem Oberbegriff zusammenfassen wollen, falls dieser existierte) und daraufhin gezielt deren Valenzschemata und Kollokationen auf Ähnlichkeiten hin vergleichen.

Andererseits könnte als Zielattribut die *Menge wahrscheinlicher Rückgabewerte von lexikalischen Funktionen* (bei jener Merkmalskombination) gewählt werden; dadurch könnten in Blickrichtung Blatt-nach-Wurzel die obengenannten semantischen Bedingungen für die betrachtete lexikalische Funktion abgelesen werden. Taucht eine bestimmte lexikalische Funktion als Element mehrmals in verschiedenen Mengen an den Blättern des Entscheidungsbaumes auf, so wäre dies als ein logisches Oder der zugehörigen Pfadausdrücke zu interpretieren; es entstünde für die semantischen Bedingungen eine Disjunktion über Konjunktionen. — Ob diese zweite Überlegung mit den gegebenen Daten und einem auszuwählenden Algorithmus Sinn macht, müssen Experimente zeigen.

3.3.7 Ansatz 2: Ausdrucksbäume

Ein direkterer Ansatz zur Ermittlung von semantischen Bedingungen könnte darin bestehen, eine disjunktive Normalform für eine gegebene lexikalische Funktion aus dem Datenbestand abzuleiten zu versuchen; eine disjunktive Normalform ist hierbei eine Disjunktion

¹⁷für eine Kurzfassung siehe obige URL (<http://yoda.cis.temple.edu:8080/UGAIWWW/lectures95/learn/C45/>), tag #3

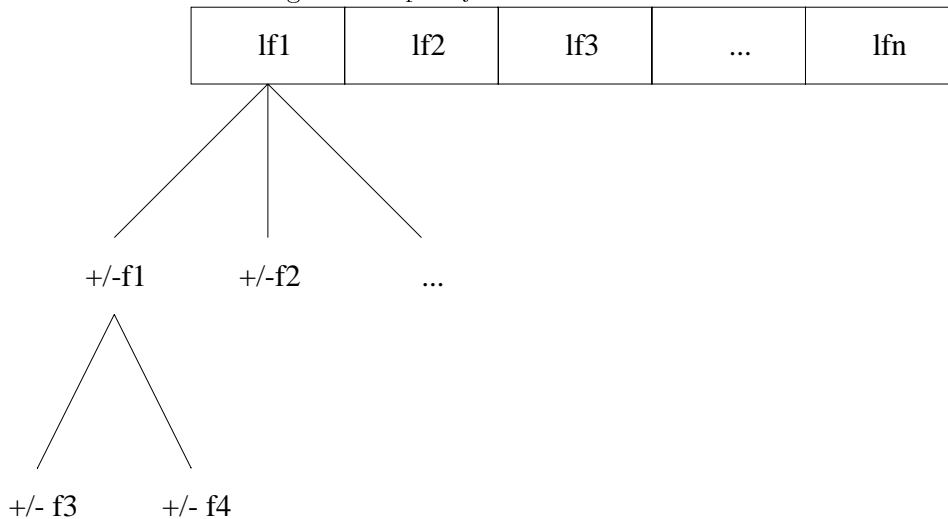
¹⁸Quinlan, bezogen auf Produktionsregeln, die aus Entscheidungsbaumen abgeleitet werden: “The programs use a simplified form of production rule $L \rightarrow R$ in which the left-hand side L is a conjunction of attribute-based tests and the right-hand side R is a class.”; siehe ebendort, S. 9

(logisches Oder) über Konjunktionen (logisches Und) von (möglicherweise negierten) semantischen Merkmalen.

Wird ein Baum aufgebaut, so werden zu einer lexikalischen Funktion lf zuerst wahrscheinliche “einstellige” (d. h. noch ohne Operationssymbole für das logische Und oder Oder) semantische Bedingungen $\pm f$ (d. h. f negiert und unnegiert) gesucht (das sind die semantischen Merkmale); dies können mehrere wahrscheinliche sein. *Sind* es mehrere, dann sind sie offensichtlich mit einem logischen *Oder* verknüpft — lf ist wahrscheinlich beim Vorkommen jedes einzelnen semantischen Merkmals vorhanden.

In einem Rekursions- bzw. Iterationsschritt könnte nun noch versucht werden, weitere wahrscheinliche semantische Merkmale $\pm f_2$ hinzuzunehmen. Diese könnten mittels eines logischen *Unds* mit den weiteren semantischen Merkmalen auf dem Pfad von der Wurzel zu einem Blatt verknüpft werden. Hierdurch ergibt sich ein “Ausdrucksbaum”¹⁹, wie er in Abbildung 2, S. 22, dargestellt ist.

Abbildung 2: *Beispiel für einen Ausdrucksbaum*



Ausdrucksbaum für $lf_1 | (\pm f_1 \wedge (\pm f_3 \vee \pm f_4)) \vee (\pm f_2) \dots$

Zwei exemplarischer Ausdrucksbäume, wie sie von der Software erzeugt werden könnten, sähen so aus (Blätter zur leichteren Bezugnahme durchnummeriert):

expression tree for Caus\$_2\$Func\$_1\$:einfl\ "o\ss en [N\$_{dat}\$~\$\sim_{acc}\$]:

- (1) --- mental
- (2) ----- permanent (2)
- (3) ----- pleasant (3)
- (4) ----- pleasant (4)
- (5) --- permanent
- (6) ----- pleasant
- (7) --- pleasant

¹⁹Dies ist meines Wissens kein offizieller Begriff in der Literatur; ich habe ihn in Anlehnung an Bäume für arithmetische Ausdrücke gewählt.

Dies entspricht

$\text{Caus}_2\text{Func}_1:\text{einflößen } [N_{dat} \sim_{acc}] |(((1) \wedge (((2) \wedge (3)) \vee (4))) \vee ((5) \wedge (6)) \vee (7)).$

expression tree for $\text{Caus}_{(2)}\text{Func}_{1}:\text{wecken } [\text{in } N_{dat} \sim_{acc}] :$

- (1) --- from_X
- (2) ----- mental
- (3) ----- permanent
- (4) ----- pleasant
- (5) ----- pleasant
- (6) ----- permanent
- (7) ----- pleasant
- (8) ----- pleasant
- (9) --- mental
- (10) ----- permanent
- (11) ----- pleasant
- (12) ----- pleasant
- (13) --- permanent
- (14) ----- pleasant
- (15) --- pleasant

Dies entspricht

$\text{Caus}_{(2)}\text{Func}_1:\text{wecken } [\text{in } N_{dat} \sim_{acc}] |((1) \wedge \dots) \vee ((9) \wedge \dots) \vee ((15) \wedge \dots).$

Dieser zweite Ansatz hätte im Vergleich zu Quinlans Überlegungen wohl den Vorteil, daß zum einen keine Gruppierung oder Zuordnung von semantischen Merkmalen zu semantischen Dimensionen gegeben sein muß. Außerdem könnte ein semantisches Merkmal mehrfach in Teilformeln auftreten, und dies sowohl negiert als unnegiert; letzteres wird bei Quinlan indirekt über die Auswahl eines anderen Werts desselben Attributs ausgedrückt.

3.3.8 Ansatz 3

Semantische Bedingungen der im Vorgehenden beschriebenen Art lassen sich auch über einen anderen Ansatz gewinnen:

Es sei eine bestimmte LFid oder ein bestimmter LFvalue lf festgehalten. Dann wird die Menge M_{lf} der Schlüsselwörter ermittelt, deren Einträge lf enthalten. In einem nächsten Schritt werden deren semantische Merkmale extrahiert und die beiden Mengen $M_{f\cap}$ bzw. $M_{f\cup}$ der "Schnitt-" bzw. "Vereinigungs-Merkmale" gebildet.

Sind sich $M_{f\cap}$ und $M_{f\cup}$ sehr "ähnlich", läßt dies auf eine Konjunktion der Elemente in $M_{f\cap}$ schließen.

Dieser Ansatz erscheint jedoch im Vergleich zum Vorhergehenden sehr grob, da durch die Schnittmengenbildung viele semantische Merkmale eliminiert werden könnten. Er wird deshalb nicht weiterverfolgt.

4 Spezifikation

4.1 Ist-Analyse

Durch die Vorgabe des Datenbestandes ist implizit auch ein Datenformat vorgegeben. Es beruht auf einer dateiweisen Speicherung der mit einem Texteditor erzeugten einzelnen Einträge, wobei verschiedene Lesarten ein und desselben Lexems auf verschiedene Dateien mit fortlaufendem numerischem Suffix verteilt sind.

Ein beispielhafter Eintrag sieht folgendermaßen aus (siehe auch Kapitel 2.3, S. 7ff):

LEX: Liebe, fem

DEFINITION:

Liebe von X zu Y wegen Z ‘X’s love towards Y because of Z’

FEATURES:

from_X, intense, pleasant, manifestable, attitudinal, approving,
excited-state, permanent, Gefühl, caused_by_Y’s_Z, directed_at_Y

VALENZ:

X = I = *N_{gen}*, *von* *N_{dat}*, PRON_{poss}
Y = II = *zu* *N_{dat}*, *gegenüber* *N_{dat}*
Z = III = *wegen* *N_{gen}*

LEX.FUNKTIONEN:

Oper ₁ :	requires a modifier, empfinden, fühlen [<i>acc</i>], entgegenbringen [<i>N_{dat}</i> DET <i>acc</i>]
Magn + IncepFact ₁ :	überwältigen [<i>N_{acc}</i>]
IncepPredMinus:	nachlassen
Liqu ₁ Func ₀ :	überwinden [PRON _{poss} /DET <i>acc</i>]
Liqu ₁ Fact ₀ :	unterdrücken [PRON _{poss} /DET <i>acc</i>]

Jede einzelne Datei ist also in Teilbereiche aufgespalten, die jeweils das eigentliche Lexem (LEX), dessen Definition, Beispiele und die darauf anwendbaren lexikalischen Funktionen und deren Rückgabewerte enthalten.

Diese Bereiche werden durch spezielle tokens eingeleitet; beendet werden sie implizit durch das Auftreten eines anderen tokens. Die tokens können sprachspezifisch anders notiert sein — beispielsweise BEISPIEL: im Deutschen oder EXAMPLE: im Englischen.

- Der Lexem-Bereich besteht aus dem eigentlichen Schlüsselwort, gefolgt eventuell von einem Leerzeichen und der Lesart (Zahl oder Zahl mit angehängtem Buchstaben). Tritt ein Komma auf, so befindet sich dahinter zusätzliche, für die Zwecke der Studienarbeit zu ignorierende Information.
- Der Definitions-Bereich wie auch der Beispiel- und (im obigen Beispiel nicht genannt) Kommentar-Bereich werden als unstrukturierter (d. h. natürlichsprachlicher) “Fließtext” angesehen.

- Der Bereich mit den Valenzschemata wird in Dreiergruppen betrachtet, die durch ein *Gleichheitszeichen* getrennt sind; die erste Gruppe (X, Y, Z, W) bezeichnet semantische Aktanten, die zweite Gruppe (I, II, III, IV) die zugeordneten syntaktischen Aktanten und die dritte Gruppe schließlich mögliche Werte.
- Der Bereich der lexikalischen Funktionen nennt für einen Eintrag nur (einige wenige passende) ausgewählte lexikalische Funktionen, und diese können einen oder mehrere Rückgabewerte besitzen²⁰. Im Falle mehrerer Rückgabewerte sind diese durch Komma oder das Zeichen “neue Zeile” getrennt.

Der Datenbestand wird “von Hand” verwaltet — d. h. der Benutzer fügt im Text spezielle Bezeichner ein und formatiert die Daten übersichtlich durch Einrückungen. Die Verwaltung der Dateien wird direkt über das Betriebssystem (shell-Kommandos) erledigt, wobei Dateien für Einträge sich in Verzeichnissen befinden, die mit dem Anfangsbuchstaben des entsprechenden Lexems benannt sind.

Auch die semantischen Überprüfungen erfolgen bislang nur intuitiv durch den Benutzer, der linguistisches Hintergrundwissen besitzt.

Ein initialer Datenbestand wurde im Laufe der Studienarbeit noch erweitert um ein zweites Wörterbuchfragment, das 40 Gefühlslexeme des Deutschen enthält, die zu Zwecken einer geplanten Generalisierung im Bereich der semantischen Dimensionen und der lexikalischen Funktionen relativ exakt erfaßt waren.

4.2 Anforderungen

Wie bereits in der Einleitung beschrieben, soll eine intelligente Schnittstelle (oder Editor) für ein Erklärendes Kombinatorisches Wörterbuch, das in Kapitel 1.1 (S. 1) eingeführt wurde, erstellt werden.

Um Portabilität zu gewährleisten, ist eine Implementierung in Java (als Java application) angestrebt, und das Programm soll zu einem späteren Zeitpunkt unter *Netscape* (als Java applet) laufen.

Bereits vorhanden ist ein Datenbestand von über einhundert Wörterbucheinträgen bzw. Dateien, für die eine komfortable Benutzerschnittstelle geschaffen werden soll. Das Menüsystem soll das Auswählen eines beliebigen Datenbestandes, die Modifikation von Teilen desselben sowie semantische Überprüfungen des Inhalts erlauben.

Der Editor soll einen Benutzer also bei den Aufgaben

- des Editierens,
- der Navigation
- und bei syntaktischen und semantischen Überprüfungen unterstützen.

Die syntaktischen Überprüfungen umfassen einen Test auf die Einhaltung grundlegender “grammatikalischer” Eigenschaften, beispielsweise die Definiertheit von Feldern in einem Wörterbucheintrag (z. B. Vorhandensein bestimmter Schlüsselwörter), die Übereinstimmung der Anzahl der Aktanten, etc.

²⁰falls es keinen Rückgabewert gibt, sollte die lexikalische Funktion nicht erscheinen

Diese semantischen Überprüfungen sollen den Test auf Kohärenz und Konsistenz der Wörterbuchartikel untereinander, einen Abgleich der Artikel in Bezug auf Vollständigkeit, Hypothesenbildung hinsichtlich der Ähnlichkeit spezifischer Informationen in verschiedenen Artikeln, etc. unter Verwendung von Algorithmen aus dem Gebiet des maschinellen Lernens umfassen.

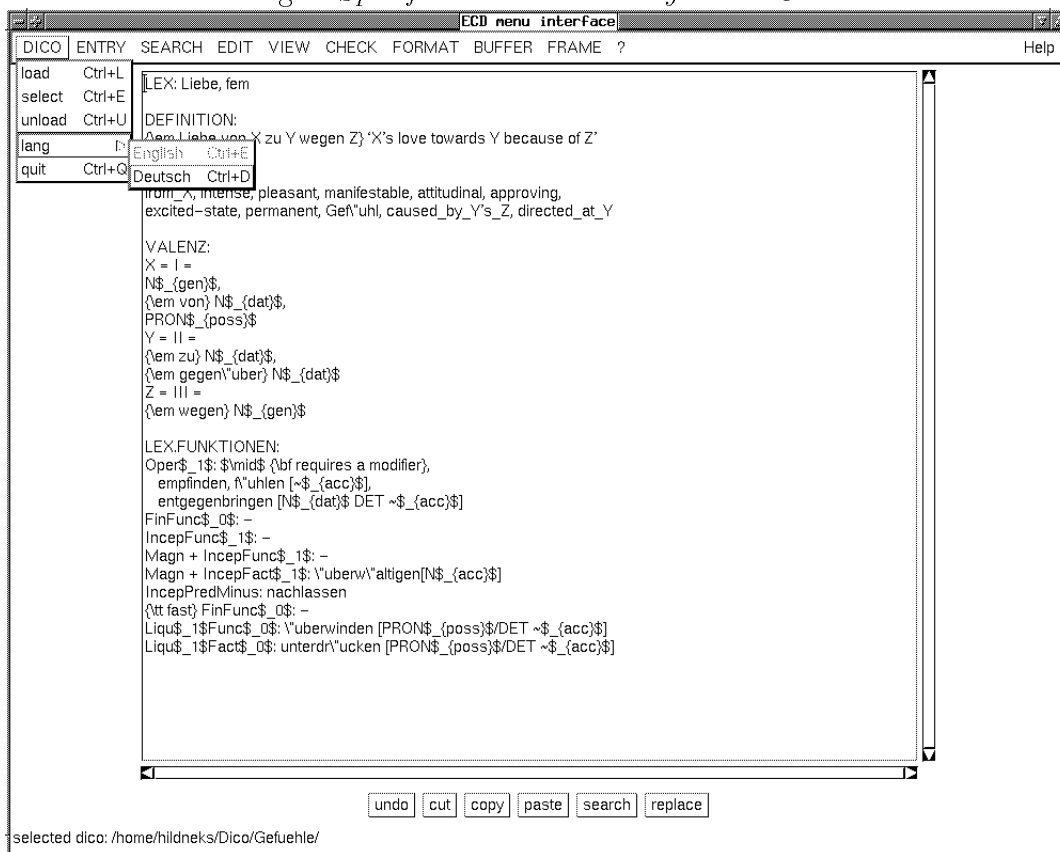
Dies wird im Nachfolgenden noch näher ausgeführt.

4.3 Interaktionsform

Es soll eine Benutzerschnittstelle geschaffen werden, die über Tastatur und Maus bedient werden kann. Der Benutzer sieht das Programm in einem oder mehreren Rahmen ablaufen, die jeweils eine Menüleiste besitzen, über die entsprechende Funktionen aufgerufen werden können.

4.4 Funktionalität des Menüsystems

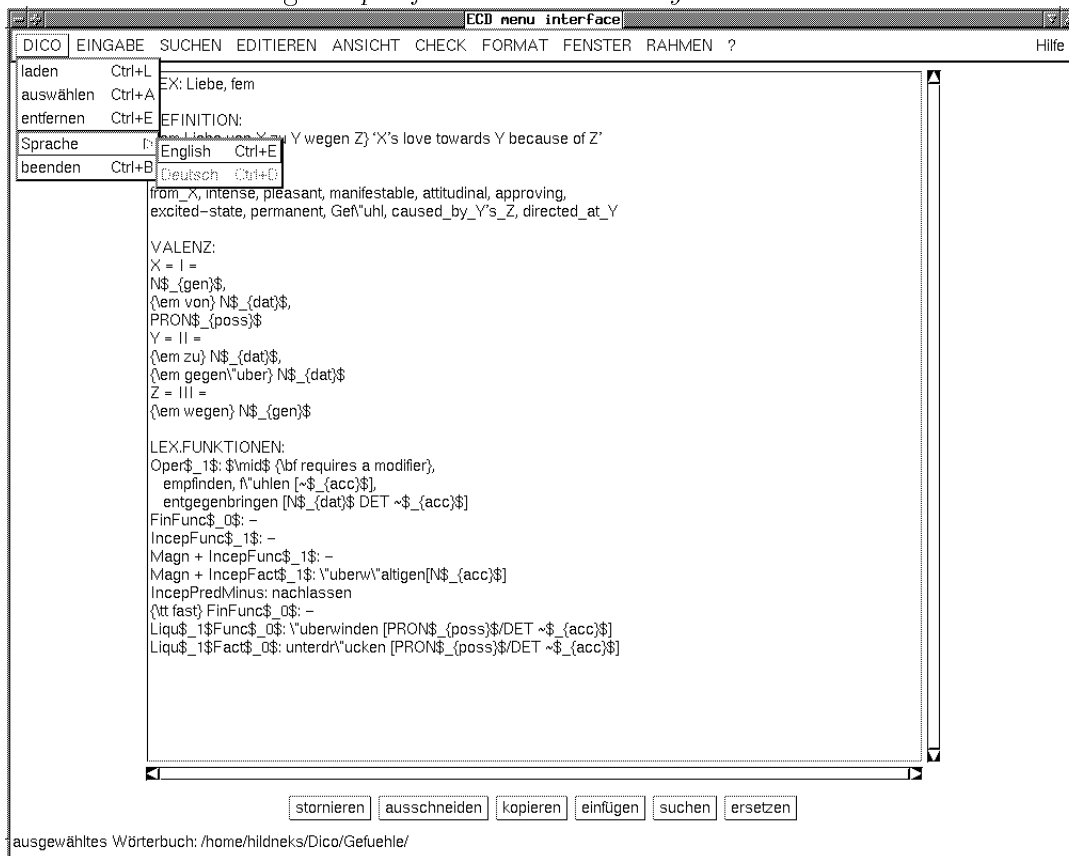
Abbildung 3: Spezifikation des Menüsystems: Übersicht 1



das Menüsystem in Englisch

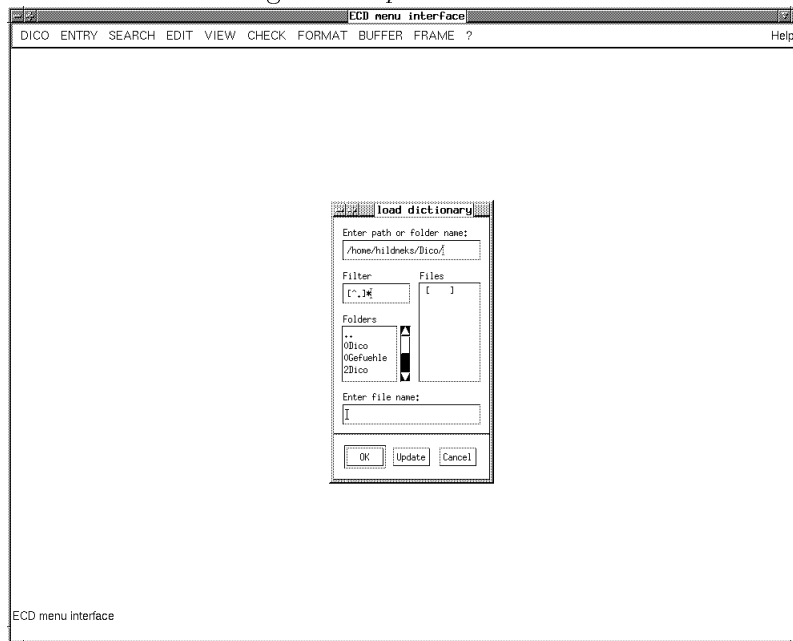
- Menüeintrag Dico:
(eine grafische Darstellung des Dico-Menüs siehe in Abbildung 3, S.26, und Abbildung 4, S.27)

Abbildung 4: Spezifikation des Menüsystems: Übersicht 2



das Menüsystem in Deutsch

- Wörterbücher von Festplatte in Speicher laden (load): Einlesen des Datenbestandes und Ablegen bzw. parsen in eine interne Struktur, die für spätere Verarbeitungen geeignet ist
 - Wörterbuch aus Speicher entfernen (unload): Freigabe des belegten Speichers
 - geladenes Wörterbuch selektieren (select): Auswahl eines Wörterbuchs zur Bearbeitung
 - Sprachumschaltung (language): Untermenü mit Auswahlmöglichkeit der zu verwendenden Dialogsprache (für Menübeschriftungen etc.)
 - Programm verlassen (quit)
- Menüeintrag Entry:
 - neuen Wörterbucheintrag erzeugen (new) und anzeigen; hinzufügen zum Pufferbereich (buffer)
 - bestehenden Wörterbucheintrag lokalisieren (open) und anzeigen; hinzufügen zum Pufferbereich (buffer)
 - während des Anzeigens: Editiermöglichkeit mit cut, copy, paste und undo; Suchfunktion auf dem angezeigten Eintrag

Abbildung 5: *Beispiel: Dateiauswahlbox*

laden von Wörterbüchern

Abbildung 6: *Spezifikation des Menüsystems: das Eintrags-Menü*

neu	Ctrl+N
öffnen...	Ctrl+O
schließen	Ctrl+L
speichern	Ctrl+S
speichern als...	Ctrl+A
auffrischen	Ctrl+U
drucken...	Ctrl+D
kopieren...	Ctrl+K

- Eintrag schließen (close): entfernen aus dem Pufferbereich
 - Eintrag speichern bzw. “speichern als” (save bzw. save as): Möglichkeit zur Umbenennung
 - Eintrag auffrischen (update): Übernahme der Änderungen vom Pufferbereich in die intern aufgebauten Datenstrukturen (Sichtbarwerden dieser Änderungen)
 - Eintrag drucken (print): Ausgabe des Eintragtextes auf Drucker oder Datei
 - copy: Kopieren des Eintragtextes
- Menüeintrag Suchen:
 - Lexem: direkte textuelle Eingabe: case-insensitive, Teilstring (mehrere Treffer möglich), Expansion eines Teilstrings auf matchenden Eintrag (z. B. durch Drücken der Tabulator-Taste); alternativ: Anzeige einer scrollbaren Liste aller Lexeme
 - Definition, Beispiel, Kommentar: jeweils Ausgabe einer Suchmaske; zusätzlich: substring/full match; case sensitive/insensitive; danach Anzeige einer (scrollbaren) Liste aller aufgefundenen Einträge

Abbildung 7: *Spezifikation: das Such-Menü*

Lemma
Definition
Features
Beispiel
Valenz
LF-Bezeichner
LF-Wert
Kommentar
AUSDRUCK

- Valenz/government pattern (GP): siehe unten (Suche eines Ausdrucks)
- semantische Merkmale, Bezeichner lexikalischer Funktionen, Rückgabewerte lexikalischer Funktionen: direkte textuelle Eingabe eines semantischen Merkmals, eines Bezeichners oder Rückgabewerts einer lexikalischen Funktion: case-sensitive, volles String-matching, Expansion; alternativ: Anzeige aller möglichen semantischen Merkmale etc.; danach Ausgabe aller Schlüssel, die dieses semantische Merkmal etc. enthalten
- Suche eines Ausdrucks: Suche eines Ausdrucks der Art “(LEX = Antrag, Ansprache & ID = Magn & VAL = große) | # = 4” — Lexem-Eintrag lautet auf “Antrag” oder “Ansprache”, und gleichzeitig ist eine lexikalische Funktion “Magn” definiert, und Magn gibt “große” zurück, oder aber die Anzahl der Aktanten ist gleich vier. Eine genaue Spezifikation der verwendbaren Ausdrücke ist in Anhang C (S. 67) gegeben.
- Menüeintrag Überprüfung (check): Durchführen von syntaktischen bzw. semantischen Überprüfungen des Datenbestandes; diese werden nachfolgend im Kapitel 4.5 (S. 30) bzw. 4.6 (S. 31) noch genauer ausgeführt

Abbildung 8: *Spezifikation: das Check-Menü*

GP/LF-Bezeichner
Statistik
Entscheidungsbaum
Ausdrucksbaum

- Menüeintrag Puffer (buffer): Umschalten des anzuzeigenden Eintrags

Abbildung 9: *Spezifikation: das Puffer-Menü (Beispiel)*

Liebe
Leidenschaft
Entz\u00fcken
Hoffnung

- Menüeintrag Rahmen (frame):

Abbildung 10: *Spezifikation: das Rahmen-Menü*

neu	
schließen	
importieren	Ctrl+I
exportieren	Ctrl+E

- Erzeugen eines neuen (open) oder schließen (close) eines bestehenden Rahmens
 - Ex- und Import von (Teil)Einträgen in andere Rahmen
- ein Hilfsmenü

4.5 Syntaktische Überprüfungen

Die syntaktischen Überprüfungen sollen die korrekte Struktur eines Eintrags sicherstellen.

Definiertheit der tokens Es sollen diejenigen Einträge angezeigt werden, bei denen manche tokens nicht erkannt werden konnten.

Definiertheit der zu den tokens gehörenden Felder Es sollen diejenigen Einträge angezeigt werden, bei denen einzelnen Felder eine ungültige Struktur besitzen oder (vermeintlich) leer sind.

Übereinstimmung der Anzahl der Aktanten gemäß GP und LFid Es sollen diejenigen Einträge angezeigt werden, bei denen die Anzahl der Aktanten gemäß des Valenzschemas echt kleiner ist als die in einem LFid codiert genannte Aktantenummer.

Exemplarische Definition syntaktischer Prüfungen Ein Wörterbucheintrag soll in seine Bestandteile (Lexem, Definition, Beispiel, Bereich der der semantischen Merkmale, lexikalische Funktionen und Kommentar) aufgespalten werden können. Diese Bestandteile sollen wiederum sinnvoll weiter zerlegt werden. Hierbei festgestellte syntaktische Fehler sollen ausgegeben werden.

Beispiel für eine Zerlegung des Bereichs der lexikalischen Funktionen:
Eingabe: (absichtlich etwas “unförmig”)

LFa: value1, value2, value3

LFb: value4

 Lfc: value5, value6

 Lfd: value7

value8, value9

LFe: value10

value11

AntiReal3: ignorieren

func: sich genießen (zwei Woerter), zweiter Eintrag

Ausgabe:

```
[Lfa, [value1, value2, value3], Lfb, [value4], Lfc, [value5, value6],
Lfd, [value7, value8, value9], Lfe, [value10, value11], AntiReal3,
[ignorieren], func, [sich genieren (zwei Woerter), zweiter Eintrag]]
```

Beispiel für eine Zerlegung des Bereichs semantischer Merkmale:

Eingabe:

```
statement, active, pleasant, anything
```

Ausgabe:

```
[statement, active, pleasant, anything]
```

Beispiel für eine Zerlegung des Valenzschemas:

Eingabe: (wiederum absichtlich etwas “unförmig”)

```
X = I = value1 ... value2 ... value3
Y = II = value4
  Z   =   III   =   value5   value6
```

Ausgabe:

```
[X, I, value1 ... value2 ... value3, Y, II, value4, Z, III,
value5, value6]
```

Eingabe:

```
X = I = N; von N; POSS [\$sim\$]
Y = II = ueber N, darueber, da{\ss} PREP
Z = III = vor N
W = IV = zu NP
```

Ausgabe:

```
[X, I, N; von N; POSS [\$sim\$], Y, II, ueber N, darueber,
da{\ss} PREP, Z, III, vor N, W, IV, zu NP]
```

Eingabe: (fehlerhaft)

```
X = I = N
Y = II
```

Ausgabe:

```
[X, I, N]
```

4.6 Semantische Überprüfungen

Semantische Überprüfungen finden nur statt im Hinblick auf Generalisierungen.

4.7 Generalisierungen

Statistische Auswertungen Unter der Annahme einer Korrelation von semantischen Merkmalen und LFids bzw. von semantischen Merkmalen und LFvalues sollen für relevante²¹ semantische Merkmale, LFids und LFvalues folgende Korrelationen bestimmt werden:

- Korrelation LFid \times semantisches Merkmal
- Korrelation LFvalue \times semantisches Merkmal
- Korrelation LFvalue \Rightarrow semantisches Merkmal
- Korrelation semantisches Merkmal \times semantisches Merkmal
- Korrelation semantisches Merkmal \times semantisches Merkmal \times semantisches Merkmal

Entscheidungsbaum Unter der Annahme einer Korrelation von semantischen Merkmalen und LFvalues soll versucht werden, für das Auftreten bestimmter LFvalues boolesche Formeln über semantische Merkmale anzugeben, so daß aus dem Vorhandensein bestimmter semantischer Merkmale auf das Vorhandensein von LFvalues gefolgert werden kann. Siehe hierzu auch 3.3.6, S. 20. — Das ist in dieser Studienarbeit *fehlgeschlagen*, weil ich Quinlans Theorie nicht auf mein Problemgebiet übertragen konnte.

Ausdrucksbaum Für jeden LFvalue soll in der Form eines Baumes ein boolescher Ausdruck über semantische Merkmale angegeben werden, die für diesen LFvalue “wahrscheinlich” sind. semantische Merkmale auf einer Ebene entsprechen dabei einer Oder-Verknüpfung, semantische Merkmale entlang eines Pfades hin zu einem Blatt einer Und-Verknüpfung. Negationen von semantischen Merkmalen sollen möglich sein. Siehe hierzu auch Kapitel 3.3.7, S. 21.

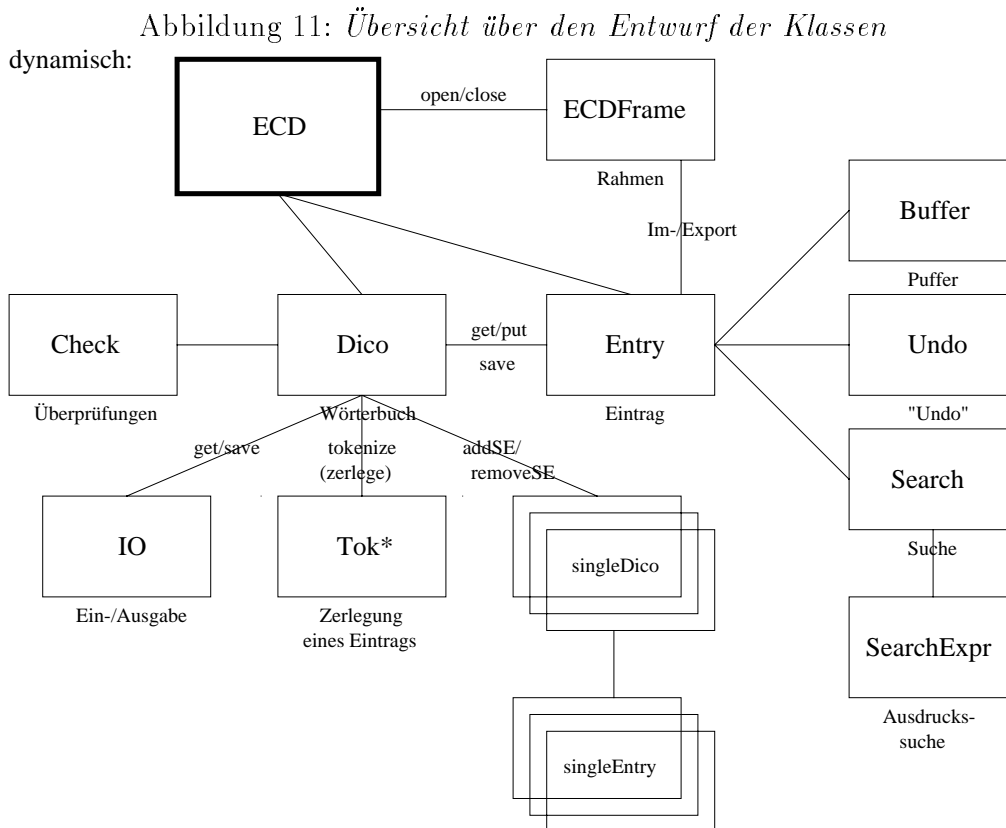
4.8 Weitere Funktionalitäten

- weitgehende Konfigurierbarkeit der Erscheinung des Programms an einer zentralen Stelle (im Quellcode): numerische und Zeichenketten-Konstanten, Schriftarten und -größen, sprachspezifische Menüeinträge, Benutzermeldungen und tokens
- Wichtige Funktionen sollen direkt über Tastendrucke aufgerufen werden können.

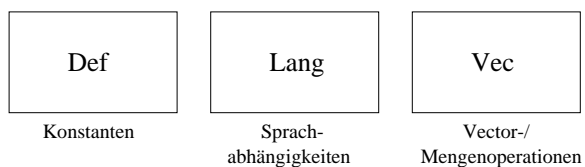
²¹d. h. solche, deren absolute Häufigkeit des Auftretens einen Schwellwert überschreitet

5 Entwurf

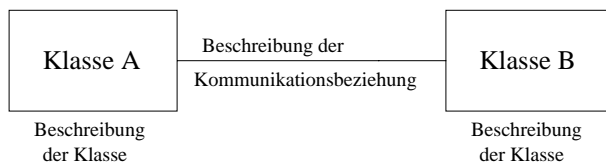
Eine Veranschaulichung des Makro-Entwurfs wird in Abbildung 11 (S. 33) gegeben. Diese Darstellung ist grob gehalten und soll nur einen Überblick vermitteln und die Orientierung erleichtern; eine detailliertere Ausführung findet sich (in Schriftform) im nachfolgenden Text.



statisch:



Legende:



5.1 Abstrakte Datentypen

Listen

- sequentielle Anordnung von Objekten
- Methoden: `Object getFirst()`, `Object getNext()`, `void putAfter(Object)`, `Boolean isFirst()`, `isLast()`, `void remove()`
- Effizienz: linearer zeitlicher Such- und Einfügeaufwand; Speichereffizienz abhängig von garbage collector (Allokation und Deallokation bewirken Speicherfragmentierung)
- Zweck: Speicherung der Reihenfolge bestimmter Einträge

hash-Tabellen

- feldartige Anordnung von Objekten; Charakter eines arrays, das mit Strings statt mit Integer-Werten indiziert wird
- Methoden: `Object get(Key)`, `void put(Key)`, `Boolean isFull()`
- Effizient: asymptotisch konstanter Such- und Einfügeaufwand (bei eventuell hohem, aber vernachlässigbarem konstantem Zugriffsaufwand), dadurch gute Skalierung; Speichereffizienz abhängig von hash-Funktion (ideal: gleichverteilende Streuung bei beliebigen Schlüsselverteilungen)

Bäume

- Datenfeld(er) für die in einem Knoten zu speichernden Werte
- Nachbildung mittels Listen für die Kinder (`child`) und Brüder (`brother`) eines Knotens

5.2 Datenstrukturen

Abhängig von den zu realisierenden semantischen Überprüfungsfunktionen müssen zeitlich effiziente Zugriffsmöglichkeiten durch geeignete Datenstrukturen zur Verfügung gestellt werden.

Die Kombination von Lexem und Lesart identifiziert einen Eintrag eindeutig; diese Kombination wird im folgenden als *Schlüssel* verwendet.

Auf den Daten treten zum einen Operationen des Einfügens (Schreibens) neuer Einträge und zum anderen des *Lesens* bestehender Einträge auf, und zwar sowohl in Form des sequentiellen Durchsuchens bzw. Anzeigens als auch des Auffindens bestimmter Teileinträge. Hierfür und für einen effizienten Zugriff auf Teileinträge müssen geeignete Datenstrukturen zur Verfügung gestellt werden.

- *Listen* scheiden wegen der fehlenden Möglichkeit zum Direktzugriff auf bestimmte Elemente aus — es ergibt sich ein ineffizientes zeitliches Verhalten beim sequentiellen Durchsuchen.

- *Felder* (arrays) scheiden wegen der zumeist schwach besetzten Matrizen (z. B. bezogen auf eine lexikalische Funktion: Argument-Wert-Matrix) aus — ineffiziente Speicherausnutzung.
- *hash-Tabellen* erscheinen prinzipiell geeignet; ihre Vorteile liegen im idealerweise asymptotisch konstanten Zugriffsaufwand (ähnlich Feldern) bei gleichzeitiger idealer Ausnutzung des Speicherplatzes bei einer “gut” gewählten Hash-Funktion. Andererseits können hash-Tabellen aber nur über *ein* (“eindimensionales”) Schlüsselwort — beispielsweise ein Lexem — indiziert werden, nicht aber noch zusätzlich über z. B. eine lexikalische Funktion. Dieses Problem kann aber durch eine “Schachtelung” (bezogen auf das Enthaltensein) bzw. “Reihenschaltung” (bezogen auf die Zugriffsart) *zweier* oder mehrerer hash-Tabellen gelöst werden; soll beispielsweise eine Abbildung

$$LF \times Lex \rightarrow \{LFvalue\}$$

(Ermittlung des Wertes *LFvalue* einer lexikalischen Funktion in Abhängigkeit der Argumente Lexem *Lex* und lexikalische Funktion *LF*, so kann dies folgendermaßen gelöst werden:

Es wird eine hash-Tabelle LF angelegt, die als Zugriffsschlüssel den Bezeichner einer lexikalischen Funktion (z. B. *v0*) mitbekommt; dadurch ist die erste Dimension in $LF \times Lex$ bestimmt. Diese hash-Tabelle enthält ihrerseits nun *weitere* hash-Tabellen als zurückzugebende Elemente — eben eine hash-Tabelle pro lexikalischer Funktion. Solche enthaltenen hash-Tabellen können speziell in Java dynamisch (zur Laufzeit) allokiert bzw. konstruiert werden.

Diese enthaltenen hash-Tabellen werden dann mit dem Lexem, für das der Wert der zuvor angegebenen lexikalischen Funktion gesucht wird, indiziert; dadurch ist nun auch die zweite Dimension in $LF \times Lex$ bestimmt. Zurückgegeben werden kann danach z. B. eine Liste aller Werte dieser LF mit diesem Lexem, eben $\{LFvalue\}$.

In Java-/objektorientiertem Pseudocode (`object.message()`) sieht dies folgendermaßen aus:

```
String LF, Lex, // Deklaration zweier Strings
Hashtable LFid, // hash-Tabelle aller LFids, enthaltend die
                // hash-Tabellen fuer die einzelnen Argumente
                LFtmp; // hash-Tabelle einer spezifischen LFid
List values; // Liste der Rueckgabewerte

in.read(LF); // Eingabe
in.read(Lex);

LFtmp = LFid.get(LF); // Auslesen der gesuchten LF (1)
values = LFtmp.get(Lex); // Auslesen des gesuchten Lexems (2)

out.print(values);
```

Die Zeilen (1) und 2 können durch Substitution der temporären Variablen *LFtmp* noch abgekürzt werden zu:

```
values = LFid.get(LF).get(Lex)
```

Konkret z. B.:

```
out.print(LFid.get('V0').get('Angst'));
// Ausgabe: { sich aengstigen, ... }
```

Ähnlich verlaufen Zugriffe auf “mehrdimensionalere” hash-Tabellen sowie Schreibzugriffe mit `put`.

5.3 Zentrale Datenstrukturen

Grundprinzipien:

- eine sortierte Liste zur Verwaltung der Schlüssel; Schlüssel bestehen aus Lexem und Lesart (Zahl)
- “geschachtelte” hash-Tabellen der oben beschriebenen Art zur Realisierung von effizienten Zugriffen auf Werten in der Art einer Abbildung $X \times Y \times \dots \rightarrow value$; Iteration über die Werte X, Y, \dots mittels Schleifen; falls zusätzlich zur Totalenumeration auch eine Anordnung benötigt wird, so wird diese mit zusätzlichen Listen realisiert, die sortierte Schlüssel für hash-Tabellen enthalten.

konkrete Strukturen:

- eine datenorientierte Klasse zur Aufnahme eines einzelnen Eintrags und mit diesem verbundene Attribute (z. B. eindeutiger Schlüssel, Modifikations-flags): `singleEntry`
- `entries`: zentrale hash-Tabelle pro geladenem Wörterbuch zur Speicherung des kompletten Datenbestandes eines Wörterbuchs, indiziert über Schlüssel
- `features`: hash-Tabelle; indiziert mit semantischen Merkmalen; gibt Liste der Schlüssel zurück; realisiert $SF \rightarrow \{ \text{Schlüssel} \}$
- `LFid`: hash-Tabelle; indiziert mit Bezeichnern (engl. *identifiers*) von lexikalischen Funktionen (z. B. `V0`); gibt (weitere!) hash-Tabelle `LFkey.tmp` zurück:
- `LFkey.tmp`: hash-Tabelle; indiziert mit Schlüssel; gibt Liste der Werte der zugehörigen lexikalischen Funktion, aufgerufen mit dem entsprechenden Schlüssel als Argument, zurück; realisiert $LF \times \text{Schlüssel} \rightarrow \text{LF-Wert}$
- `LFvalue`: hash-Tabelle; indiziert mit Rückgabewerten von lexikalischen Funktionen (z. B. `machen`); gibt (weitere!) hash-Tabelle `LFid.tmp` zurück:
- `LFid.tmp`: hash-Tabelle; indiziert mit Bezeichnern lexikalischer Funktionen; gibt Liste der Schlüssel zurück, die diese `LFid/LFvalue`-Kombination verwenden
- `LFidKey`: hash-Tabelle; indiziert mit Bezeichner einer lexikalischen Funktion; gibt Liste mit Schlüssel zurück, deren Einträge diese lexikalische Funktion verwenden
- `LFvalueKey`: hash-Tabelle; indiziert mit dem Rückgabewert einer lexikalischen Funktion; gibt Liste mit Schlüssel zurück, in deren Einträge dieser Rückgabewert auftritt

- **LFvalueKey2**: wie **LFvalueKey**, aber indiziert mit LFid “:” LFvalue
- **LFidKeyRev**: hash-Tabelle; indiziert mit Schlüsseln; gibt Liste aller zugehörigen LFids zurück (“reverse”)
- **LFvalueKeyRev**: hash-Tabelle; indiziert mit Schlüsseln; gibt Liste aller zugehörigen LFvalues zurück (“reverse”)
- **LFvalueKeyRev2**: wie **LFvalueKeyRev**, aber indiziert mit LFid “:” LFvalue
- **keysSorted**: sortierte Liste aller Schlüssel
- **featuresSorted**: sortierte Liste aller semantischen Merkmale
- **LFidSorted**: sortierte Liste aller Bezeichner von lexikalischen Funktionen
- **LFvalueSorted**: sortierte Liste aller Rückgabewerte lexikalischer Funktionen
- **LFvalueSorted2**: wie **LFvalueSorted**, aber enthält LFid “:” LFvalue

Eine Kurzübersicht hierzu siehe auch auf im Anhang B, S. 65.

5.4 Funktionale Anforderungen

(siehe auch “Soll-Konzept”)

- Laden (open/new), Bearbeiten (copy, cut, paste, undo), Speichern (save) und Drucken (print) eines Eintrags
- “sinnvolle” Suchfunktionen auf den Einträgen eines Wörterbuches, beispielsweise nach bestimmten Lexemen oder Einträgen mit bestimmten semantischen Merkmalen
- Überprüfung einzelner Einträge nach erfolgter Eingabe (close)
- Überprüfung eines gesamten Wörterbuches nach bestimmten Kriterien

5.5 Externe Schnittstellen

- Schnittstellen zum Betriebssystem für Ein-/Ausgaben von/auf Permanentpeicher
- grafische Benutzerschnittstelle für die Benutzerinteraktion (Maus- und Tastatur-events)
- eventuell: command line-interface für batch-Betrieb

5.6 Leistungsanforderungen

- bei Benutzerdialog (Menüauswahl, Editieren) übliche (unmerkliche) Verzögerungen, jedoch akzeptable Wartezeiten bei Operationen, die sich auf Teile oder die Gesamtheit des Datenbestandes beziehen (Überprüfungen)
- Wahl von Datenstrukturen und Algorithmen, die auch für größere Datenbestände (1 000 Einträge und mehr) noch “gut skalieren”

5.7 Entwurfseinschränkungen und Portierbarkeit

Der Entwurf als solcher unterliegt keinen Einschränkungen; die Implementierung hingegen könnte Probleme aufwerfen: Die gegebene Datenstruktur eines Eintrags, verbunden mit dem Zugriffsmechanismus (dateiweise Anforderung über das Betriebssystem) und der Forderung, daß synonyme tokens in beliebiger Anordnung geparsed werden sollen, dürfte ein zeitlich effizientes Einlesen eines Wörterbuchs erschweren (siehe “Soll-Konzept”).

Durch die Verwendung von Java für die Implementierung sollte der Editor auf allen Plattformen lauffähig sein, die die Java Virtual Machine emulieren.

Aufgrund der großen Vielfalt von Plattformen (Java unter verschiedenen *ix-Oberflächen, unter Windows 3.01 und NT) ist mit nicht-konformem Verhalten zu rechnen. Beispielsweise unterstützen manche window manager eine Fensterbeschriftung, wie sie in Java mit der `Frame.setLabel(String)`-Methode vorgesehen ist, während andere Fenstermanager gar keine Beschriftung durchführen. Des weiteren kann es zu unterschiedlichem (und unübersichtlichem) grafischem Layout diverser AWT-Komponenten (abstract windowing toolkit) kommen, beispielsweise bei der Komponente `java.awt.List`, die auf manchen Oberflächen bzw. Konfigurationen richtigerweise *eine* Zeile pro `MenuItem` reservieren, während bei anderen jedem `MenuItem` eine Leerzeile voran- und nachsteht.

5.8 Modularisierung

Der von mir gewählte Entwurf besteht im Wesentlichen aus drei Klassen:

- dem sichtbaren Benutzeroberflächen-Objekt in künftig: `ECD.java`: Dieser Teil beherbergt den zentralen Frame, in dem das Menüsystem aufgebaut wird, sowie das event-handling für diesen Frame. — Dieser Teil könnte später eventuell lokal beim client laufen.
- einer “Wörterbuch”-Klasse in künftig: `Dico.java`: Dieses Objekt wird für die jeweiligen zu bearbeitenden Wörterbücher (Datenbasen) instanziiert. Er umfaßt das Laden und Speichern von ganzen Wörterbüchern bzw. Wörterbucheinträgen. — Dieser Teil könnte später eventuell auf einem server laufen.
- einer “Eintrag”-Klasse in künftig: `Entry.java`: Dieses Objekt wird für die zu bearbeitenden Einträge eines Wörterbuchs instanziiert. Er umfaßt das Neuanlegen bzw. Öffnen von einzelnen Einträgen, das Schließen derselben, die eigentlichen textuellen Editierfunktionen, Suchfunktionen sowie weitere Funktionen wie z. B. das Drucken.
- (weitere Objekte zum handling von mehreren Eintragsfenstern etc.)

Das Wörterbuch-Objekt in `Dico.java` benötigt weitere Klassen `InOut.java` bzw. `Tok*.java` zur Ein-/Ausgabe (Festplatte – Hauptspeicher) bzw. zur Zerlegung der einzelnen Einträge in ihre Bestandteile (z. B. Lexem-, Definitions-, LF-, etc.- Teil).

5.9 Syntaktische Überprüfungen

Die syntaktischen Überprüfungen können aufgrund der Spezifikation sofort implementiert werden. Es wird eine Routine zur Zerlegung des Eintrags in seine Bestandteile benötigt; als

Eingabe erhält sie den Text eines Eintrages, wie er aus einer Datei oder einem Eingabefenster gelesen wird; die Ausgabe erfolgt als Zahlpaare (Anfang, Ende), die jeweils den Anfang und das Ende der unterschiedenen Bereiche angeben. Diese Zahlpaare dienen wiederum als Eingabe für die Routinen zur weiteren Zerlegung, die dann entsprechend aufgebaute Listen (siehe Spezifikation, Kapitel 4.5, S. 30) zurückgeben.

5.10 Semantische Überprüfungen

Semantische Überprüfungen finden im Rahmen dieser Studienarbeit nur im Hinblick auf Generalisierungen statt.

5.11 Generalisierungen

Im nachfolgenden werden einige Algorithmen kurz in Pseudocode skizziert.

5.11.1 Statistische Auswertungen

Die nachfolgenden Algorithmen sollten nur auf Mengen von LFids, LFvalues und semantischen Merkmalen aufsetzen, deren Anzahl eine bestimmte absolute Häufigkeit (bezogen auf das Vorkommen in allen Einträgen) übersteigt, d. h. in einem Vorlauf sind die relevanten Mengen mittels eines einstellbaren Schwellwertes zu ermitteln.

Sie benötigen als Grundlage weitere Algorithmen, die Daten kombinieren und Mengen bereitstellen:

- Es werden geordnete Menge aller semantischen Merkmale, LFids und LFvalues benötigt.
- Jedes semantische Merkmal, jede LFid und jeder LFvalue muß auf die Menge der Schlüsselwörter abgebildet werden können, deren Einträge dieses semantische Merkmal, diese LFid oder dieses LFvalue verwenden.

(siehe hierzu auch Anhang B, S. 65)

Korrelation LFid \times semantisches Merkmal Der Algorithmus ist sehr direkt und hat quadratischen Aufwand in der Anzahl der semantischen Merkmale und LFids:

```
t = 0.75; // empirisch zu ermittelnder Schwellwert
FOR i := 1 TO Anzahl(LFids)
  l := i-te LFid
  Ml := Menge aller Schlüssel, deren Einträge l enthalten
  FOR j := 1 TO Anzahl(semantische Merkmale)
    f := j-tes semantisches Merkmal
    Mf := Menge aller Schlüssel, deren Einträge f enthalten
    MUnion := Ml  $\cup$  Mf;
    MInter := Ml  $\cap$  Mf;
    p := Anzahl(MUnion) / Anzahl(MInter);
    IF p  $\geq$  t THEN
```

```

                gib Korrelation  $lf \wedge f$  aus
        ENDFOR j
    ENDFOR i

```

Korrelation LFvalue \times semantisches Merkmal Der Algorithmus ist analog zum vorhergehenden:

```

t = 0.75; // empirisch zu ermittelnder Schwellwert
FOR i := 1 TO Anzahl(LFvalues)
    l := i-ter LFvalue
    Ml := Menge aller Schlüssel, deren Einträge l enthalten
    FOR j := 1 TO Anzahl(semantische Merkmale)
        f := j-tes semantisches Merkmal
        Mf := Menge aller Schlüssel, deren Einträge f enthalten
        MUnion := Ml  $\cup$  Mf;
        MInter := Ml  $\cap$  Mf;
        p := Anzahl(MUnion) / Anzahl(MInter);
        IF p  $\geq$  t THEN
            gib Korrelation  $lf \wedge f$  aus
        ENDFOR j
    ENDFOR i

```

Korrelation semantisches Merkmal \Rightarrow semantisches Merkmal Im Gegensatz zum nachfolgenden Algorithmus müssen alle Paare (a, b) und (b, a) von semantischen Merkmalen getestet werden, da eine Implikation *nicht* kommutativ ist.

```

t = 0.75; // empirisch zu ermittelnder Schwellwert
FOR i := 1 TO Anzahl(semantische Merkmale)
    f1 := i-tes semantisches Merkmal
    Mf1 := Menge aller Schlüssel, deren Einträge f1 enthalten
    FOR j := 1 TO Anzahl(semantische Merkmale)
        f2 := j-tes semantisches Merkmal
        Mf2 := Menge aller Schlüssel, deren Einträge f2 enthalten
        MInter := Mf1  $\cap$  Mf2;
        p := Anzahl(MInter) / Anzahl(Mf1);
        IF p  $\geq$  t THEN
            gib Implikation  $f1 \Rightarrow f2$  aus
        ENDFOR j
    ENDFOR i

```

Korrelation semantisches Merkmal \times semantisches Merkmal Da die Konjunktion kommutativ ist, müssen nur Paare (a, b) und nicht auch Paare (b, a) von semantischen Merkmalen a, b getestet werden.

```

t = 0.75; // empirisch zu ermittelnder Schwellwert
FOR i := 1 TO Anzahl(semantische Merkmale)
    f1 := i-tes semantisches Merkmal

```

```

Mf1 := Menge aller Schlüssel, deren Einträge f1 enthalten
FOR j := i + 1 TO Anzahl(semantische Merkmale)
  f2 := j-tes semantisches Merkmal
  Mf2 := Menge aller Schlüssel, deren Einträge f2 enthalten
  MUnion := Mf1  $\cup$  Mf2;
  MInter := Mf1  $\cap$  Mf2;
  p := Anzahl(MInter) / Anzahl(Mf1);
  IF p  $\geq$  t THEN
    gib Konjunktion f1  $\wedge$  f2 aus
  ENDFOR j
ENDFOR i

```

Korrelation semantisches Merkmal \times semantisches Merkmal \times semantisches Merkmal Auch hier müssen aufgrund der Kommutativität und Assoziativität der Mengenoperationen nur Tripel $a < b < c$ von semantischen Merkmalen a, b, c getestet werden. Der Algorithmus ist analog dem vorhergehenden:

```

t = 0.75; // empirisch zu ermittelnder Schwellwert
FOR i := 1 TO Anzahl(semantische Merkmale)
  f1 := i-tes semantisches Merkmal
  Mf1 := Menge aller Schlüssel, deren Einträge f1 enthalten
  FOR j := i + 1 TO Anzahl(semantische Merkmale)
    f2 := j-tes semantisches Merkmal
    Mf2 := Menge aller Schlüssel, deren Einträge f2 enthalten
    MUnion := Mf1  $\cup$  Mf2;
    MInter := Mf1  $\cap$  Mf2;
    FOR k := j + 1 TO Anzahl(semantische Merkmale)
      f3 := k-tes semantisches Merkmal
      Mf3 := Menge aller Schlüssel, deren Einträge f3 enthalten
      MUnion := MUnion  $\cup$  Mf3;
      MInter := MInter  $\cap$  Mf3;
      p := Anzahl(MInter) / Anzahl(Mf1);
      IF p  $\geq$  t THEN
        gib Konjunktion f1  $\wedge$  f2  $\wedge$  f3 aus
      ENDFOR k
    ENDFOR j
  ENDFOR i

```

5.11.2 Entscheidungsbaum

Zur Spezifikation dessen, was ein Entscheidungsbaum leisten soll, siehe 4.7, S. 32.

Ein Algorithmus für einen Entscheidungsbaum bereitet mir große Probleme, da ich Quinlans Theorie zur Klassifizierung von Daten nicht sinnvoll auf das vorliegende Themengebiet (Klassifizierung von Wörterbucheinträgen) übertragen kann; es sind Parallelen aufzufinden, problematisch ist aber die Wahl eines Zielattributs. Deshalb habe ich Algorithmen variiert, um abzuschätzen, was Sinn machen könnte, war aber erfolglos. Deshalb erfolgt hier keine Entwurfsangabe.

5.11.3 Ausdrucksbaum

Zur Spezifikation dessen, was ein Ausdrucksbaum leisten soll, siehe 4.7, S. 32.

An Java angelegte Pseudonotation für die Generierung eines Ausdrucksbaum:

Knotendefinition:

```
class ExprNode {
    boolean isNeg = false;
    String feature = "";
    double p;

    ExprNode
        brother = null,
        child = null;
} // class
```

Eigentlicher Algorithmus:

```
double ExprThreshold = 0.7; // Schwellwert für Hinzunahme zum Baum
ExprNode buildExprTree(
    String lf, // die betreffende lf
    Vector fVTry, // die noch zu prüfenden features
    Vector fVKeys, // der angesammelte Schnitt aller Schlüssel
                // aller features entlang des Pfades von der Wurzel
    ExprNode n // Knoten, an den die Kinder angehängt werden sollen
) {
    double
        p = 0.0,
        notp = 0.0;
    ExprNode nTmp = n; // der Startknoten

    for(int i = 0; i < |fVTry|; i++) {
        String f = fVTry(i); // das zu betrachtende feature
        Vector
            fVKeysTmp = fVKeys ∩ features(f),
            notfVKeysTmp = fVKeys ∩ ¬ features(f);
        p =  $\frac{LFvalueKey2(lf) \cap fVKeysTmp}{LFvalueKey2(lf) \cup fVKeysTmp}$ ;
        notp =  $\frac{LFvalueKey2(lf) \cap notfVKeysTmp}{LFvalueKey2(lf) \cup notfVKeysTmp}$ ;

        if(p ≥ ExprThreshold OR notp ≥ ExprThreshold) {
            hänge_neues_Kind_an;
            nTmp = neues_Kind;
            nTmp.feature = f;
        } // if
        if(p ≥ ExprThreshold) {
            nTmp.isNeg = false;
            nTmp.p = p;
        } // if
    }
}
```

```

        if(notp ≥ ExprThreshold) {
            nTmp.isNeg = true;
            nTmp.p = notp;
        } // if
    } // for

    // Sprung in die Rekursion
    nTmp = n.child; // reset
    while(nTmp ≠ null) {
        if(! nTmp.isNeg)
            nTmp = buildExprTree(lf, fVTry\nTmp.feature,
                                fVKeys ∩ features.nTmp.feature), nTmp);
        else
            nTmp = buildExprTree(lf, fVTry\nTmp.feature,
                                fVKeys ∩ ¬features(nTmp.feature), nTmp);
        nTmp = nTmp.brother;
    } // while

    return(n);
} // buildExprTree

ExprNode getExprTree(String lf) {
    ExprNode n = new ExprNode();

    n = buildExprTree(lf, aDico.sd.featuresSorted, aDico.sd.keysSorted, n);

    return(n);
} // getExprTree

void printExprTree(ExprNode en) {
    // benutzt indent, indStep

    while(en ≠ null) {
        ruecke_ein(indent);
        if(en.isNeg)
            drucke("NOT");
        drucke(en.feature, en.p * 100);
        indent += indStep;
        printExprTree(en.child);
        indent -= indStep;

        en = en.brother;
    } // while
} // printExprTree
void exprTree() {
    for(int i = 0; i < |LFvalueSorted2|; i++) {
        String lf = LFvalueSorted2(i);
        ExprNode r = getExprTree(lf);
    }
}

```

```
        drucke("Ausdrucksbaum für "+ lf + ": ");
        if(r.child  $\neq$  null)
            printExprTree(r.child);
        else
            drucke("(kein Baum gefunden)");
    } // for
} // exprTree
```

6 Implementierung

6.1 Java

6.1.1 Allgemeine Einführung

[Flanagan1997], S. 3ff beschreibt Java als (übersetzt) eine ”einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, Architektur-neutrale, portable, hoch-performante, multi-threaded und dynamische Sprache”.

Ohne dies im Detail auszuführen, möchte ich diese grundlegenden Eigenschaften kurz skizzieren: *Einfachheit* ergibt sich aus der geringen Anzahl von Sprachkonstrukten und der Anlehnung der Notation an Sprachelemente der Programmiersprache C. *Objektorientiertheit* kommt zum Ausdruck im Vorhandensein von Objekten, die sich gegenseitig Nachrichten (engl. *messages*) zuschicken; dies geschieht in der Form `object.message`. *Verteiltheit* ergibt sich durch die Möglichkeit des entfernten Methodenaufrufs (engl. *remote method invocation*, RMI) und einer sehr weitreichenden Unterstützung von Netzwerkaktivitäten, beispielsweise durch die URL-Klasse. Eng damit gekoppelt ist das Prinzip der *dynamischen* Sprache — eine Java-Quellprogramm (Endung `.java`) wird zunächst compiliert, wodurch eine Klassen-datei (Endung `.class`) mit Code für die virtuelle Java-Maschine (engl. *Java virtual machine*, JVM) entsteht, dann aber zumeist interpretiert²²; damit verknüpft sind dynamische Speicherverwaltung, automatische garbage collection und dynamisches Nachladen von Code z. B. über das Internet und dessen Ausführung. *Robustheit* und *Sicherheit* ergeben sich u. a. durch die Abwesenheit von Zeigervariablen und expliziter Allokation und Deallokation von Speicherplatz, durch strenge Typisierung, ein “Sandkasten-Konzept” für applets etc. *Architektur-Neutralität* und *Portabilität* wurden durch die Definition der Java virtual machine erzielt, die von Eigenschaften realer hard- und software abstrahiert; mittlerweile existieren Java-Interpreter unter Windows und Windows NT (Netscape, Microsoft Internet Explorer, VisualAge, etc.) und diversen Unix-Plattformen (Netscape, Sun’s Java Development Kit (JDK), etc.). Die Eigenschaft, *mehrere threads* (leichtgewichtige Prozesse, die einer Java-Applikation zugeordnet sind) erlauben Parallelität innerhalb einer Anwendung.

Java unterstützt vom Sicherheitskonzept her grundsätzlich zwei Arten von Programmen:

- Applikationen (engl. *applications*) und
- “Applikatiönchen” (engl. *applets*)

Letztere sind Programme, die in einer HTML-Seite eingebettet werden können, bestimmten Sicherheitsbeschränkungen unterliegen (“sand box principle”) und ein festgelegtes Interface implementieren müssen, um von einem Browser wie z. B. dem Microsoft Internet Explorer oder dem Netscape Navigator oder Communicator abgearbeitet werden zu können. Die Sicherheitsbeschränkungen können durch sogenannte “trusted applets” insofern vermieden werden, als daß applets, die von bestimmten, vorher festgelegten “vertrauenswürdigen” Websites geladen werden, erhöhte Zugriffsrechte zugebilligt werden. — Die hier gewählte Implementation ist jedoch in der Form einer (eigenständigen) Applikation unter Ausnutzung der Methoden aus (vorwiegend) `java.awt`, des abstract windowing toolkits von Java.

²² mittlerweile existieren JIT-compiler (just in time), die den Code vor der Ausführung nochmals speziell für die Zielmaschine compilieren und damit optimieren

Es soll das Java Development Kit JDK 1.1 von Sun (siehe [SunJDK]) zur Implementierung verwendet werden, wie es auf dem Disney- und IS-Pool der Abteilung Informatik installiert ist. Nähere Informationen hierzu finden sich unter [SunJDK], das eine nützliche online-Referenz darstellt. Für die desktop-Referenz wurde [Flanagan1997] gewählt; als Tutorial diente [Keller1997].

6.1.2 Der Java CompilerCompiler

Der Java CompilerCompiler JavaCC[SunJCC] ist eine "100 % pure Java application", d. h. er läuft auf allen Systemen, die die Java virtual machine emulieren.

JavaCC nimmt als Eingabedatei eine Grammatikdefinitionsdatei *.jj und erzeugt daraus mehrere *.java-Dateien, die in einem weiteren Durchlauf mit dem herkömmlichen Java compiler javac in ausführbare Klassendateien *.class gewandelt werden.

Innerhalb einer *.jj-Datei wird eine Grammatik-Produktion der Form $A \rightarrow B^{23}$ dargestellt als ein Aufruf der Methode A(), in deren Rumpf der reguläre Ausdruck B steht; B gibt hierbei den Ausdruck an, den JavaCC zu matchen versuchen soll. Mit jedem Ausdruck B (bzw. Nonterminal) kann Java-Code assoziiert werden, der beim Auftreten dieses Nonterminals ausgeführt wird.

Ein einfaches Beispiel einer *.jj-Datei, die korrekt rekursiv geschachtelte Strukturen von geschweiften Klammern erkennt²⁴, ist:

```
options {
    LOOKAHEAD = 1;
}

PARSER_BEGIN(Simple)

public class Simple {

    public static void main(String args[]) throws ParseException {
        Simple parser = new Simple(System.in); // instanziiert sich selbst
        parser.Input(); // Aufruf des Startsymbols
    } // main
} // class Simple

PARSER_END(Simple)

void Input() :
{
{
    MatchedBraces() ("\n"|"r")* <EOF>
} // Input

void MatchedBraces() :
{}
```

²³mehr zu diesem komplexen Thema bei [AhoEtAl1986], S. 26

²⁴in Anlehnung an /JavaCC/examples/SimpleExamples/Simple1.jj

```
{
  "{" [ MatchedBraces() ] "}" // [optionale] Rekursion
} // MatchedBraces
```

Das Softwarepaket JJTree ist ein front-end für JavaCC. JJTree generiert explizit aus einer *.jjt-Datei einen Parse-Baum, dessen Knoten den Nonterminalen einer *.jjt-Datei entsprechen können. Die Ausgabe von JJTree dient als Eingabe für JavaCC. Die Ausgabe von JavaCC durchläuft wiederum die oben beschriebene Vorgangsfolge; zusammenfassend ergibt sich also folgende Sequenz:

```
jjtree *.jjt → javacc *.jj → javac *.java → java * [.class] → (Ausführung)
```

Die Wurzel des Parse-Baums kann nach dem erfolgreichen Einparsen als Knoten vom Typ SimpleNode oder auch eines beliebigen, selbst definierten Typs, zurückgegeben und der Baum daraufhin traversiert werden.

Die im JavaCC-Paket enthaltenen Beispiele umfassen unter anderem Grammatiken für einfache Infix-Taschenrechner, für die Sprache Java selbst, für JavaCC, aber auch für die HyperText Markup Language HTML und manche andere Sprachen. Damit ist JavaCC ein ungeheuer mächtiges Werkzeug.

6.1.3 Vorüberlegungen

Im Hinblick auf eine spätere Transformation des Programms in eine netzbasierte Version ist zu beachten, daß der "Flaschenhals Netzwerk" sowie möglicherweise copyright-überlegungen es erforderlich machen, daß der Hauptteil der Daten physisch auf dem server verbleibt und nur ein unbedingt notwendiges Quantum wirklich zum Benutzer transferiert wird. Eine solche Transformation könnten (ausgehend von einer Implementierung in Java) naheliegenderweise entweder zwei Java-Applikationen (je eine auf server-, die andere auf client-Seite) sein, die über remote method invocation RMI kommunizieren, oder aber ein Java applet, das über eine HTML-Seite als Träger innerhalb eines Java-fähigen browsers auf den client heruntergeladen wird und über noch zu bestimmende Kommunikationsmechanismen (möglicherweise common gateway interface (CGI); Problem: Sicherheitsbeschränkungen; mögliche Lösung: trusted applets) mit dem server kommunizieren könnte.

6.1.4 Datenstrukturen und Methoden von Java

Zur Realisierung der bei der Spezifikation angegebenen abstrakten Datentypen der Liste und der hash-Tabelle stehen in Java die beiden folgenden konkreten Datentypen bereit:

(geordnete) Liste Java-Klasse: `java.util.Vector`

Auswahl der vorhandenen Methoden:

- `public Vector(int initialCapacity, int capacityIncrement);`
Konstruktor
- `public final synchronized Object elementAt(int index);`
- `public final synchronized Enumeration elements();`
- `public final synchronized insertElementAt(Object obj, int index);`
- `public final synchronized removeElementAt(int index);`

hash-Tabelle Java-Klasse: `java.util.Hashtable`

Auswahl der vorhandenen Methoden:

- `public Hashtable(int initialCapacity, float loadFactor);`
Konstruktor; hash-Tabelle wächst dynamisch bei berschreitung des loadFactors
- `public synchronized Object put(Object key, Object value);`
Schreiboperation, “assoziatives Feld” (adressiert über einen beliebigen Schlüsselwert)
- `public synchronized Object get(Object key);`
Leseoperation
- `Object remove(Object key);`
- weitere Methoden: `Boolean containsValue`, `Boolean containsKey`, `Enumeration elements`, `toString`

Bäume Java unterstützt Bäume nicht direkt als Datentyp. Sie können jedoch einfach mittels einer Knotenklasse nachgebildet werden. Das nachfolgende Beispiel veranschaulicht die zugrundeliegende Idee:

```
public class Node {

    SomeType data = new SomeType(); // irgendein Datentyp
    Node
        brother = null, // Liste der Brueder
        child = null; // Liste der Kinder
} // Node
```

Anwendung im Programmcode z. B. mittels:

```
Node
    r = new Node(), // Wurzel (root)
    n; // Laufzeiger
n = r;
n.data = 1;
n.brother = new Node();
n = n.brother;
n.data = 2;
n.brother = new Node();
n = n.brother;
n.data = 3
```

Dies erzeugt einen Baum mit der Wurzel “1” und zwei Kindern mit den Werten “2” und “3”.

6.2 Klassen des Wörterbuchs

Eine übersichtliche grafische Darstellung der Klassen findet sich in Abbildung 11, S. 33.

6.2.1 Die Klasse `singleDico.java`

Diese Klasse wird für jedes einzelne geladene Wörterbuch instanziiert. Sie enthält die eigentlichen Wörterbuchdaten in `Hashtable entries`, wie sie vom Betriebssystem eingelesen wurden, sowie diverse weitere Listen und Hashtabellen, die eine effiziente Suche auf der Wörterbuchstruktur ermöglichen sollen.

Diese Klasse ist im Anhang B, S. 65, im Original angegeben.

6.2.2 Die Klasse `singleEntry.java`

Diese Klasse wird für jeden einzelnen Wörterbucheintrag instanziiert. Sie enthält zunächst die Wörterbucheinträge im Feld `String entry`, wie sie von `InOut.java` erhalten werden, aber auch verschiedene boolean-flags für Editier-Zustände sowie integer-Werte, die von den `Tok*`-Klassen gesetzt werden und die Start- und Endpositionen bestimmter token-Bereiche angeben; diese integer-Werte sollen jedoch nicht direkt gelesen werden, weil dafür spezielle `get*`-Zugriffsoperationen zur Verfügung gestellt werden (siehe unten).

Diese Klasse ist im Anhang A, S. 62, im Original angegeben. Im folgenden werden die Felder und Methoden eingehender erklärt:

Felder

- `String entry`: der komplette Eintragstext, wie er auch im Editierfenster dargestellt wird
- `String dir, file`: der zugehörige Pfad- und Dateiname
- `Boolean isModified, isSaved, isNew, isSelected`: verschiedene Zustände; letzterer dient als Auswahlkriterium bei der Darstellung in Listen (`isSelected == true` z. B. wenn der entsprechende Eintrag ein Suchkriterium erfüllt hat und nun noch mit anderen Einträgen zusammen zur endgültigen Selektion in einer Liste dargestellt wird)
- `int lex.start, lex.stop, def.start, def.stop, ...`: von `TokEntry` gesetzt; Auslesen allerdings nicht direkt, sondern über:
- für Leseoperationen:
 - `public String getEntry()`: obiger `String entry`
 - `public String getLex()`: das volle Lexem in der Form *Lexem i, s.*, wobei *Lexem* der Titel des Eintrages, *i* die Lesart und *s* das Geschlecht (oder bei nicht-Substantiven andere Zusatzinformation) sein kann
 - `public String getLex2()`: wie bei `getLex()`, allerdings ohne den *,s*-Zusatz; dieser Wert wird im Folgenden auch als *Schlüssel* bezeichnet, da er einen Wörterbucheintrag eindeutig bezeichnet
 - `public String getLex3()`: wie bei `getLex2()`, allerdings zusätzlich ohne den *i*-(Lesart-)Zusatz, d. h. das "reine" Lexem
 - `public String getDef()`, `public String getFeatures()`, `public String getExample()`, `public String getGP()`, `public String getLF()`, `public String getComment()`: die entsprechenden Teileinträge (ohne tokens)

– `public int getNr()`: die Lesart als Zahl (wo vorhanden), oder `int 0`, falls keine Lesart angegeben

- Alle Strings (außer `getEntry()`) sind von den tokens bereinigt.

Methoden außer dem default-Konstruktor keine; lade- und speicher-Methoden befinden sich in `Entry.java`

6.2.3 Die Klasse `ECD.java`

Die Mutter dieser Klasse ist die Klasse `java.awt.Frame`, die einen plattformabhängigen Rahmen darstellt, der eine Menüleiste aufnehmen kann.

Diese Klasse übernimmt den Auf- und Ab- bzw. Umbau (bei Wechsel der Sprache) der Menüstruktur sowie das zentrale event-handling für die Menüleiste — mit Ausnahme der Menüs, die unter dem Menüpunkt “Puffer” auftauchen; diese müssen dynamisch auf- und abgebaut werden und hängen eng mit der Pufferverwaltung zusammen, so daß diese Funktionalität in die Klasse `Buffer.java` verlagert wurde.

Felder

- `Dico aDico`, `Entry anEntry`, `Search aSearch`, ..., `Buffer aBuffer`, `ECDFrame anECDFrame`, `Help aHelp`: jedes `ECD`-Objekt instanziiert sich jeweils eine der vorgenannten Klassen
- (diverse Variablen für das Menüsystem)

Methoden

- `void addPanel(Panel p)`: Setzen des Hauptfensters
- `void addSLine(Label l)` bzw. `void addSLine(String s)`: polymorphes Setzen der Statuszeile
- `void enableMenu(String[] s)` bzw. `void enableMenu(String s)`: polymorphe Funktion zum Aktivieren von Menüeinträgen; die zugehörigen eindeutigen “event strings” Strings sind definiert in `Def.java`
- `void disableMenu(String[] s)` bzw. `void disableMenu(String s)`: analog dem vorherigen Punkt

Beziehungen zu anderen Klassen Es existieren diverse Beziehungen zu `Dico`, `Entry`, `Search`, `Edit`, `View`, `Check`, `Format`, `Buffer`, `ECDFrame`, `QMark` und `Help`.

`ECD` instanziiert:

- `Dico` für einzelne Wörterbücher
- `Entry` für einzelne Einträge
- `Search` zur Suche von Einträgen

- Check zur Durchführung von Überprüfungen
- Buffer zur Umschaltung zwischen Einträgen
- ECDFrame zum Erzeugen und Beseitigen von weiteren ECD-Instanzen bzw. Eintragsfenstern

6.2.4 Die Klasse Dico.java

Felder

- `public singleDico sd`: momentan bearbeitetes Wörterbuch
- `public Vector dicoBuffer`: Liste der geladenen Wörterbücher

Methoden

- `public void addSE(singleEntry se, singleDico sd)`: fügt einen Eintrag `se` in den Datenbestand in `sd` ein, d. h. also updaten der Listen und hash-Tabellen
- `public void removeSE(singleEntry se, singleDico sd)`: löscht einen Eintrag `se` aus dem Datenbestand in `sd`, d. h. also Durchsuchen und updaten der hash-Tabellen und als Spezialfall (wenn z. B. der letzte Eintrag mit einer bestimmten lexikalischen Funktion gelöscht wurde) auch update der entsprechenden Listen (hier `LFidSorted`)
- `public Vector getKeysSorted()`, `public singleEntry getEntry(String key)`, `public singleEntry getEntry(int pos)` (polymorph), `public String getKey(int pos)`, `public int size()`, `public void removeEntry` (löschen aus der Datenstruktur — bei “alten” Einträgen nach einem Editiervorgang), `public void putEntry(singleEntry se)` (speichern in der Datenstruktur — bei “neuen” Einträgen nach einem Editiervorgang), `public void saveEntry(singleEntry se)` (weiterreichen an `InOut` zum Speichern auf Betriebssystemebene): Kapseln den Zugriff auf das aktuelle Wörterbuch (in `Dico.sd`)

Beziehungen zu anderen Klassen verwendet:

- ECD zur Darstellung von Komponenten
- `InOut` zum Einlesen (load) von Wörterbüchern und Speichern einzelner Einträge (letzteres getriggert von `Entry`)

6.2.5 Die Klasse Entry.java

Felder

- `public singleEntry se`: momentan bearbeiteter Eintrag
- `public Vector entryBuffer`: Liste der bislang bearbeiteten Einträge

Methoden (keine wesentlichen)

Beziehungen zu anderen Klassen verwendet:

- **ECD** zur Darstellung von Komponenten
- **Dico** zum Auslesen, Schreiben und updaten von Wörterbucheinträgen (sowohl betriebssystemmäßig als auch auf internen Datenstrukturen von Dico)

6.2.6 Die Klassen Tok*.java

Diese Klassen sind Hilfsklassen zur Zerlegung eines Eintrags in seine Bestandteile. Ihre Benutzung ist angelehnt an die Klasse `java.util.StringTokenizer`, d. h. ihr Konstruktordient der initialen Belegung mit einem Eintrag, woraufhin auf diese Instanz bezogen die entsprechenden Elemente ausgelesen werden können.

`TokEntry.java`

- `public TokEntry(singleEntry se, boolean returnTokens)` (Konstruktor)
- `public TokEntry(singleEntry se)` (polymorpher Konstruktor)

`TokEntry` zerlegt einen Eintrag in seine Bestandteile Lexem, Definition, Beispiel, semantische Merkmale, lexikalische Funktionen und Kommentar. Unterscheidungskriterium sind hierbei die in `Lang.java` angegebenen `tokens`. Die ermittelten Werte (Start- und Stop-Positionen als integers der jeweiligen Bereiche) werden direkt in die übergebene Struktur `singleEntry` geschrieben. Die (Start-)Werte enthalten die `tokens`, wenn `returnTokens` den Wert `true` hat, ansonsten werden sie abgeschnitten; letzteres ist der default bei Benutzung des Konstruktors `TokEntry(singleEntry se)`.

Nach dem Aufruf dieses Konstruktors liefern die anderen `Tok*`-Klassen weitere Zerlegungen:

`TokLF.java`

- `public TokLF(String s)` (Konstruktor)
- `public Vector get()`

`TokLF` zerlegt einen übergebenen LF-Eintrag in seine Bestandteile. Unterscheidungskriterium ist hierbei ein *Doppelpunkt* nach dem Funktionsbezeichner sowie *Kommata* oder Beginn einer neuen Zeile zur Trennung der einzelnen Werte; *Leerzeichen* sind keine Trenner, da der Rückgabewert einer lexikalischen Funktion auch aus mehreren (nicht durch Kommata getrennten) Wörtern bestehen kann. Zurückgegeben wird eine Liste, die abwechselnd (d. h. an Position 0, 2, ...) den Bezeichner der lexikalischen Funktion als String und (an Position 1, 3, ...) eine (weitere, verschachtelte) Liste der Funktionswerte enthält.

`TokFeat.java`

- `public TokFeat(String s)` (Konstruktor)
- `public Vector get()`

`TokFeat` zerlegt einen übergebenen Eintrag eines semantischen Merkmals in seine Bestandteile. Unterscheidungskriterium sind hierbei *Kommata* und auch *Leerzeichen*. Zurückgegeben wird eine Liste, die die einzelnen Werte der semantischen Merkmale enthält.

TokGP.java

- `public TokGP(String s)` (Konstruktor)
- `public Vector get()`

TokGP zerlegt einen übergebenen GP-Eintrag in seine Bestandteile: Zurückgegeben wird eine Liste der syntaktischen und semantischen Aktanten und der darauffolgenden Feldeinträge.

Felder Es existieren keine sichtbaren Felder.

Methoden

- **TokEntry:** keine; der beim Aufruf übergebene `singleEntry` wird direkt (über Seiteneffekte) mit Werten belegt; dieser Ansatz wurde gewählt, um ein effizientes Einparsen mit möglichst wenig overhead durch Funktionsaufrufe und Rückkehr von denselben (Stackoperationen) zu verursachen; zudem wären Methoden zur Kapselung des Schreibzugriffs immer nur Einzeiler gewesen, die direkt die Felder belegt hätten
- **TokLF, TokFeat, TokGP:** `public Vector get()` zur Rückgabe der geparsten Werte; diese werden zum Einhashen in die verschiedenen hash-Tabellen verwendet

6.2.7 Die Klasse Undo.java

Diese Klasse realisiert einen “zyklischen” Undo-Puffer für den Editierbereich von Einträgen. Bei jedem neuen Eintrag wird der Undo-Puffer zurückgesetzt; “wesentliche Änderungen” (z. B. bei cut oder paste) können sukzessive rückgängig gemacht werden; ist der Anfangszustand durch mehrmaliges Ausführen von (Anklicken des Buttons) Undo erreicht, “wrapped” die Ersetzung um, und es wird der ursprüngliche Inhalt vor Ausführen von Undo angezeigt (usw.).

Felder (keine relevanten)

Methoden Zugriffsmethoden:

- **Konstruktor:** allokiert internen Pufferbereich, setzt Lese- und Schreib-“Cursor”
- **void reset():** zurücksetzen des Pufferbereichs und der Cursors; verwendet, wenn neuer Eintrag bearbeitet werden soll
- **void add(String s):** hinzufügen eines Strings zum Pufferbereich; wenn er dem zuletzt hinzugefügten (`s2`) textuell gleich (`s.equals(s2)`), wird er *nicht* mehr identisch hinzugefügt; aktualisieren des Lesecursors
- **void append(String s):** hinzufügen ohne aktualisieren des Lesecursors
- **String get():** sukzessives Auslesen des Pufferbereichs durch mehrmaligen Aufruf dieser Methode; wrap around

Beziehungen zu anderen Klassen Die Klasse `Undo` wird von der Klasse `Entry` instanziiert.

6.2.8 Die Klasse `Search.java`

Diese Klasse realisiert Suchfunktionen über den Einträgen. Entweder wird direkt nach Eingaben gepromptet, oder es werden markierte Bereiche im Editierbereich implizit als Suchstring angenommen.

Zusätzlich wird eine “expression search” durchgeführt, bei dem die Einträge gefunden werden sollen, die einem bestimmten Such-Ausdruck entsprechen.

Felder (keine relevanten)

Methoden (keine relevanten)

Beziehungen zu anderen Klassen

- Aufruf der Search-Methoden durch das event-handling in ECD.
- Search verwendet `Entry` zum Darstellen der Einträge.
- Search verwendet `SearchExpr` zum Auffinden der Einträge, die einem bestimmten Suchausdruck entsprechen.

6.2.9 Die Klasse `SearchExpr.java`

Diese Klasse wird mittels `JavaCC` aus `SearchExpr.jj` erzeugt und dient der Suche nach Ausdrücken (speziell auch nach bestimmten Valenzenschemata); näheres zu Ausdrücken siehe obiges Begriffslexikon.

`SearchExpr.jj` enthält letztlich eine Grammatikdefinitionen und Java-Code, der einen mit booleschen Operatoren aufgebauten Ausdruck entgegennimmt, diesen in seine Bestandteile zerlegt und als Ergebnis eine Liste der Schlüsselwörter zurückgibt, die diesen Ausdruck erfüllen.

Die Grammatik ist im Anhang C (S. 67) angegeben.

Unter Verwendung des Beispielcodes für einen einfachen Infix-Taschenrechner aus den `JavaCC`-Beispielen²⁵ wurden folgende Anpassungen vorgenommen:

- Änderung des Eingabestroms von `System.in` auf einen `CharStream`
- “Dynamisierung” des Parsers, damit er in mehreren Instanzen von ECD ohne Seiteneffekte allokiert sein kann
- Anpassung der Operatoren `+` (Addition) bzw. `*` (Multiplikation) auf Mengenvereinigung bzw. Schnittbildung; eine Menge enthält hierbei Schlüssel; das unäre `-` wurde ersetzt durch ein Mengenkomplement (bezüglich der Menge aller Schlüssel).

²⁵siehe sowohl `/JavaCC/examples/SimpleCalculators/Calc1i.jj` als auch [AhoEtAl1986], S. 38ff (Wandlung Infix nach UPN) und S. 62ff (Abstrakte Kellerautomaten)

- Bereitstellung dieser drei Mengenoperationen in der Klasse `Vec.java`.
- Erweiterung der `element()`-Routine um Code, der eine “atomare” Abfrage in linke Seite, (Un)Gleichheitszeichen und rechte Seite aufspaltet, die Suche auf dem Dico-Datenbestand tatsächlich durchführt und einen Vektor der diesen atomaren Ausdruck erfüllenden Schlüssel auf dem Stack ablegt (push).
- weitere Abwandlung der `element()`-Routine, um auch auf der rechten Seite von Ausdrücken weitere, ebenfalls geschachtelte Anfragen zuzulassen
- weitere Anpassung der `ls()`-Routine, um neben den eigentlichen tokens auch (gültige bzw. bereits verwendete) Bezeichner lexikalischer Funktionen erkennen zu können (Abgleich mit `ADico.sd.LFid` zur Laufzeit und “Werfen” einer entsprechenden Exception bzw. Fehlermeldung, falls der aktuelle Bezeichner nicht bekannt sein sollte

Die resultierende Abfragesprache sollte sehr mächtig sein, gleichzeitig aber auch intuitiv bedienbar sowie (durch entsprechende Ausnahmebehandlung) *sicher* und robust sein, was Syntaxfehler betrifft.²⁶

6.2.10 Die Klasse `Check.java`

Diese Klasse realisiert die syntaktischen und semantische Prüfungen bzw. Generalisierungen.

Felder (keine relevanten)

Methoden (keine relevanten)

Beziehungen zu anderen Klassen

- Aufruf der Check-Methoden durch das event-handling in ECD.
- Check verwendet Dico zum Zugriff auf Einträge.

6.2.11 Die Klasse `Buffer.java`

Diese Klasse ermöglicht ein komfortables Umschalten der gerade bearbeiteten Einträge mittels eines eigenen Menüs, das dynamisch (während der Programmlaufzeit) u. a. beim Öffnen eines Eintrags wächst und beim Schließen eines Eintrags schrumpft.

Felder

- Menu `bufferMenu`: das in ECD dargestellte Menü
- Hashtable `bufferSE`: hash-Tabelle für die gepufferten Einträge, indiziert über Schlüssel
- Hashtable `bufferMI`: “parallele” hash-Tabelle für die zugehörigen MenuItem, indiziert ebenfalls über Schlüssel

²⁶Die von JavaCC erzeugte Klasse `ASCIICharStream.java` benutzt “deprecated features” von Java 1.0; die diesbezüglich beim Compilieren ausgegebene Warnmeldung wird bei einem Übergang auf eine neuere Version von JavaCC jedoch verschwinden und wird zumindest von der aktuellen Version des Java-Compilers noch unterstützt.

Methoden

- `void update()`: wird von den event-handlern vor der event-Bearbeitung aufgerufen, um einen eventuell gerade angezeigten Eintrag in einer TextArea vor deren Ausblendung zu sichern
- `String getTA(String key)`: gibt den Inhalt der TextArea zurück, wie er war, kurz bevor sie das letzte mal ausgeblendet wurde, oder, falls der Eintrag noch nicht angezeigt wurde, den originalen Eintragstext aus Dico
- `void add(String key)`: fügt einen Eintrag in die eigenen Felder ein (erzeugt u. a. einen Menüeintrag) — aufgerufen von Dico beim Eröffnen eines neuen oder bestehenden Eintrags
- `void remove(String key)`: löscht einen Eintrag aus den eigenen Feldern (löscht u. a. einen Menüeintrag) — aufgerufen von Dico beim Schließen eines Eintrags
- `Vector getEntries()`: gibt einen Vektor der gespeicherten `singleEntries` zurück; wird von Dico beim Verlassen des Programms bzw. dem dabei getriggerten Speichern aller veränderter Einträge benötigt

Beziehungen zu anderen Klassen

- zu `ECD`, um den Menüeintrag dynamisch zu setzen
- zu `Dico`, um an die Einträge des aktuellen Wörterbuchs zu kommen (z. B. `update` des `isModified`-flags)
- zu `Entry`, um z. B. das Vorhandensein der TextArea zu prüfen und den dortigen Text vor dem Ausblenden zu sichern

6.2.12 Die Klasse `ECDFrame.java`

Felder

- `static Vector ECDBuffer`: speichert in einer statischen (d. h. für alle `ECDBuffer`-Instanzen gleich sichtbaren) `Vector`-Struktur alle bislang erzeugten `ECDs`
- `static String imExport`: der Puffer für den Im- und Export von Daten

Methoden (nur Methoden, die direkt vom `ECD`-event handler aufgerufen werden)

Beziehungen zu anderen Klassen beherbergt in `ECDBuffer` Zeiger auf alle erzeugten `ECD`-Instanzen; wird unter anderem benötigt, den Menüeintrag von “Import” bei diesen freizuschalten, sofern die Variable `imExport` mit einem Wert belegt ist

6.2.13 Weitere Klassen

Zusätzlich zu den bereits beim Entwurf aufgezählten Klassen werden weitere Helferklassen benötigt:

- ein (statisches) “Sprach”-Objekt in `Lang.java`: Es kapselt die sprachabhängigen Bestandteile des Programms, indem alternative Ausgabertexte zur Verfügung gestellt werden. Seit Java 1.1 wird ein solches “Internationalization” Feature offiziell unterstützt [noch konvertieren?]; hierbei werden “automatisch” die Zeichenein- bzw. -ausgaben in `InputStreamReader` bzw. `OutputStreamReader` nach bzw. von 16 bit Unicode angepaßt, es werden Mechanismen für landesspezifische Benutzertexte zur Verfügung gestellt, angepaßte Methodenaufrufe formuliert (z. B. für Zeit- und Datumsausgabe und Sortier Routinen) etc.
Die Konfiguration erfolgt beispielsweise durch das Auslesen einer Betriebssystem-Variablen, die eine default-”Locale” festlegen kann.
Momentan ist dieses topic noch *ohne* das offizielle Internationalization Feature gelöst.
- ein (statisches) “Definitions”-Objekt in `Def.java`: Es kapselt alle sonstigen, vor allem numerischen Bestandteile des Programms, indem Bezeichner für “magische” Zahlen (beispielsweise die default frame-Größe beim startup) etc. zur Verfügung gestellt werden, so daß eine zentrale änderung durchgeführt werden kann.
- eine Helferklasse `InOut.java` für die Ein-/Ausgabe von Einträgen
 - `public InOut(String dir, String mode)`: Konstruktor; initialisiert mit dem Betriebssystem-Verzeichnis, in dem sich ein Wörterbuch befindet, und dem gewünschten Operationsmodus (LOAD oder SAVE)
 - `public boolean eof()`, `public void reset()`, `public singleEntry getFirstEntry()`, `public singleEntry getLastEntry()`, `public singleEntry getNextEntry()`, `public singleEntry getPrevEntry()`, `public int size()`: zur Realisierung von Leseoperationen auf einer Instanz (bei LOAD)
 - `public void putEntry(singleEntry se)`: zur Realisierung von Schreiboperationen auf einer Instanz (bei SAVE)
- eine Helferklasse `Pop.java` für modale Popup-Menüs zur Bestätigung (engl. *confirmation*) und für scrollbare Textbereiche in einem extra Fenster
- eine Helferklasse `Str.java` für Stringoperationen (z. B. Zerlegung eines Strings in Zeilen mit einer maximalen Länge)
- eine Helferklasse `Vec.java`
 - zum effizienten Einfügen und Löschen aus sortierten Listen (Java-spezifisch: `Vector`),
 - für die Mengenoperationen Durchschnitts-, Vereinigungs- und Komplementbildung (bezogen auf die Grundmenge aller Schlüssel), sowie
 - für die “Expansion” von Stringausdrücken (hin auf ein sinnvolles Muster)

7 Zusammenfassung und Ausblick

7.1 Allgemeine Ergebnisse

Die hier dargestellte Entwicklung der Oberfläche für ein EKW ist geeignet für den lokalen single-user-Betrieb. Ihre Stärken liegen

- in der einfachen Portierbarkeit auf andere Plattformen (implizit durch die Verwendung von Java als Implementierungssprache)
- in der einfachen Bedienung auch für ungeübtere Benutzer, konkret
 - ein intuitives GUI,
 - effiziente “MenuShortcuts” über direkte Tasteneingabe,
 - copy/paste zur Reduzierung von Tippaufwand und Eingabefehlern,
- ihrer mächtigen Suchfunktion (rekursive Suchanfragen erlaubt; copy/paste aus einem Editierfenster heraus in die Eingabemaske), basierend auf
- effizienten internen Datenstrukturen (einfache und geschachtelte Hash-Tabellen: asymptotisch konstante Zugriffszeit auch bei großen Datenmengen, effiziente Speicherausnutzung abhängig von einem parametrisierten “Ladefaktor”, dynamische Reallokation),
- intelligenten, wissensbasierten und (wo unvermeidbar, weil NP-vollständige Probleme) heuristischen Algorithmen, sowie
- ihrer leichten Anpaßbarkeit auf geänderte Anforderungen:
 - intuitiv einfaches Ändern der *.jj-Grammatikdefinition
 - “Anstöpseln” anderer Tokenizer für andere Wörterbuchstrukturen, solange die `get()`-Zugriffsinterfaces zum Auslesen passend implementiert werden)
 - Erweiterung der Token- und Menüsprache durch Erweiterung des Sprach-Arrays in `Def.java` und `Lang.java` um die entsprechenden Strings

7.2 Resultate

Eine Diskussion der syntaktischen und semantischen Überprüfungen, zusammen mit Ergebnissen der Generalisierungen, siehe im Kapitel 3, S. 9.

Abschließend kann gesagt werden, daß die syntaktischen Überprüfungen naturgemäß sehr wirkungsvoll sind. Bei den semantischen Überprüfungen wurden nur die Generalisierungen näher untersucht, und deren Ergebnis hängt kritisch von der Qualität des zugrundegelegten Datenbestands, dem ausgewählten Themengebiet bzw. der Größe des Datenbestands ab²⁷:

- Je “besser” und “präziser” die Klassifizierung mittels semantischer Dimensionen (und auch LFids und LFvalues), desto eher können sinnvolle Korrelationen gefunden werden.

²⁷Zudem sind Generalisierungs- bzw. Klassifizierungsalgorithmen von Natur aus zumeist den “komplexeren” Problemen zuzordnen.

- Das ausgewählte Themengebiet bestimmt indirekt, ob eine sinnvolle Auswertung vorgenommen werden kann. Die 40 Gefühlslexeme des Deutschen zeigen hierbei eine gute Eignung für das Auffinden von sinnvollen Korrelationen.
- Eine Korrelation ist um so sicherer anzunehmen, je größer der zugrundegelegte Datenbestand ist.

7.3 Ausblick

Mögliche weitgehendere Erweiterungen der entwickelten Benutzerschnittstelle sind zum Einen die Aufspaltung des Programms in einen client-Teil, der beim Benutzer verbleibt und die (sichtbare) Benutzeroberfläche enthält, und einen server-Teil, der auf einen speziellen port lauscht, über ein TCP/IP-Netz angesprochen wird und die eigentliche Funktionalität enthält (Bereitstellung von Wörterbuchdiensten) und insbesondere den Datenbestand zentral speichert. Die möglichen Mechanismen hierzu wurden auf S. 38 und S. 47 geschildert.

Sinnvoll könnte auch die Erweiterung des servers hin auf eine query-Sprache sein, so daß er einfache requests verarbeiten und responses/replies zurückliefern kann. Eine Anwendung hiervon wäre ein Interface für die Ausdruckssuche oder speziell die Suche nach Synonymen.

Ein weiterer wichtiger Schritt in Richtung verbesserter Effizienz des Zugriffs würde im Konvertieren des Datenbestandes auf ein einheitlicheres Format mit leichteren Zugriffsfunktionen bestehen — konkret eine SQL-konforme Datenbank, auf die mittels Java/JDBC (Java data base connectivity kit) leichter zugegriffen werden könnte; dadurch könnten wohl auch Synchronisationsprobleme beim Mehrbenutzerbetrieb (Leser-Schreiber-Problematik) leichter ausgeschlossen werden.

Literatur

- [AhoEtAl1986] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*; Addison-Wesley, 1986
- [DeinigerEtAl1992] M. Deininger, H. Lichter, J. Ludewig, K. Schneider. *Studienarbeiten: ein Leitfaden zur Vorbereitung, Durchführung und Betreuung von Studien-, Diplom- und Doktorarbeiten am Beispiel Informatik*; B. G. Teubner, Stuttgart, 1992
- [Flanagan1997] David Flanagan. *Java in a Nutshell*; O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472
desktop-Referenz zur Java-Entwicklungsumgebung
- [HyafilRivest1976] Laurent Hyafil, Ronald L. Rivest. *Constructing Optimal Binary Decision Trees is NP-Complete*; Information Processing Letters, Volume 5, Number 1; S. 15 – 17
Angabe eines in polynomialer Zeit ablaufenden nichtdeterministischen Algorithmus' und einer Reduktion
- [InfoDuden1993] [Bearbeitet von] Prof. Dr. Volker Claus, Dr. Andreas Schwill. *Duden Informatik: Ein Sachlexikon für Studium und Praxis*; Dudenverlag, Mannheim, 1993
- [JavaCC] Sun Microsystems. *The Java Parser Generator JavaCC*;
<http://www.suntest.com/JavaCC/>
Angabe von formalen Grammatiken, Parsing, Aufbau von Syntaxbäumen
- [Keller1997] Hartmut Keller. *Java-Kompaktkurs*; Skript der Universität Stuttgart, Kompaktkurs gehalten vom 22. – 26. September 1997
Beispiele zu `java.awt`
- [Melchisedech1997] Ralf Melchisedech. *Konzeption und Aufbau objektorientierter Software (KAOS)*, Vorlesungsskript der Universität Stuttgart
Wasserfallmodell der Software-Entwicklung
- [Melcuk1988] Igor A. Mel'čuk. *The Meaning-Text Linguistic Model as the Framework for Dependency Syntax*, in: *Dependency Syntax: Theory and Practice*, editiert von Mark Aronoff, Universität von New York, S. 43 – 101
Einführung in das MTT-Modell
- [Melcuk1996] Igor A. Mel'čuk. *Lexical Functions: A Tool for the Description of Lexical Relations in a Lexicon*, in: *Lexical Functions in Lexicography and Natural Language Processing*, Leo Wanner (ed.), John Benjamins Publishing Company, Amsterdam/Philadelphia; S. 37 – 102
u. a. vollständige Auflistung aller lexikalischen Funktionen mit zahlreichen Beispielen
- [MelcukWanner1994] Igor A. Mel'čuk, Leo Wanner. *Lexical Co-occurrence and Lexical Inheritance*; Sonderdruck aus: *LEXIKOS 4*, Afrilex-Reeks/Series 4:1994; Buro Van die Wat, Stellenbosch, S. 87ff u. a. Erläuterungen zu Erklärenden Kombinatorischen Wörterbüchern; eine inhaltliche Mixtur aus [MelcukWanner1996] und [Melcuk1996]
- [MelcukWanner1996] Igor A. Mel'čuk, Leo Wanner. *Lexical Functions and Lexical Inheritance for Emotion Lexemes in German*, Sonderdruck aus: Leo Wanner (ed.) *Lexical*

Functions in Lexicography and Natural Language Processing, John Benjamins Publishing Company, Amsterdam/Philadelphia, 1996

Vererbungseigenschaften von Gefühlsexemen im Deutschen mit 40 Beispiellexemen

[Mintert1998] Stefan Mintert. *sgzip: Semantische Textkomprimierung — Mit Bedeutung*; “iX — Magazin für professionelle Informationstechnik”, Ausgabe 4/1998, S. 158 – 159, Verlag Heinz Heise; online-Version auf <http://www.ix.de/ix/artikel/1998/04/158/> Zeitungsentee zum 1. April 1998

online-Archiv von Artikeln aus “c’ t — Magazin für Computertechnik”, “iX — Magazin fuer professionelle Informationstechnik” und “Telepolis — Magazin für Netzkultur” auf <http://www.ix.de/>

[news] o. A. *USENET news betreffend Java*, `news:de.comp.lang.java` und `news:comp.lang.java.*`, z. B. `news:comp.lang.java.gui`
Klärung aktueller Implementierungs-/Realisierungsfragen

[Quinlan1992] J. Ross Quinlan. *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers, San Mateo, California, 1992

Algorithmen zu Entscheidungsbäumen etc.

siehe hierzu auch <http://yoda.cis.temple.edu:8080/UGAIWWW/lectures95/learn/C45/>

[SunJDK] Sun Microsystems. *The Java Development Kit JDK*, <http://java.sun.com/products/jdk/>

online-Referenz zur Java-Entwicklungsumgebung

[SunJCC] Sun Microsystems. *Java CompilerCompiler — The Java Parser Generator*, <http://www.suntest.com/JavaCC/>

online-Referenz zum Java-Parser-Generator

[Wanner1996] Leo Wanner. *Introduction [to Lexical Functions in Lexicography and Natural Language Processing]*, Sonderdruck aus: Leo Wanner (ed.) *Lexical Functions in Lexicography and Natural Language Processing*, John Benjamins Publishing Company, Amsterdam/Philadelphia, 1996, S. 1 – 27 und S. 32

Einteilung von LF-Beziehungen in syntagmatische und paradigmatische; vergleichende Gegenüberstellung von Beziehungen gemäß Lyons, Hausmann und Benson et al. mit Mel’čuks LF-Einteilung

A Die Klasse singleEntry.java

```
class singleEntry // represents a single dico entry
  implements Cloneable {
  String
    entry, // the actual data (corresponding to entry TextArea)
    key = "", // backup of the getLex2() return value
    dir, // the corresponding path name
    file; // the corresponding file name

  // additional information (entry content split up;
  // numbers represent offsets in entry field
  // valid values are set by TokEntry
  int
    lexStart = -1, lexStop = -1, // the full lexeme
    lex2Start = -1, lex2Stop = -1, // lexeme without sex
    lex3Start = -1, lex3Stop = -1, // lexeme without sex and nr.
    defStart = -1, defStop = -1,
    featuresStart = -1, featuresStop = -1,
    exampleStart = -1, exampleStop = -1,
    gpStart = -1, gpStop = -1,
    lfStart = -1, lfStop = -1,
    commentStart = -1, commentStop = -1;

  // int nr = 0; // number
  String subKey = "";

  boolean
    isModified = false,
    isSaved = true,
    isNew = false,
    isSelected = true; // visibility in List

  // public access methods
  // reading values
  public String getEntry() { // whole entry
    return(entry);
  } // entry

  public String getKey() { // key field only
    return(key);
  } // getKey

  public String getLex() {
    if(lexStart != -1)
      return(entry.substring(lexStart, lexStop));
    else
      return("");
  }
}
```

```
} // getLex

public String getLex2() {
    if(lex2Start != -1)
        return(entry.substring(lex2Start, lex2Stop));
    else
        return("");
} // getLex2

public String getLex3() {
    if(lex3Start != -1)
        return(entry.substring(lex3Start, lex3Stop));
    else
        return("");
} // getLex3

public String getDef() {
    if(defStart != -1)
        return(entry.substring(defStart, defStop));
    else
        return("");
} // getDef

public String getFeatures() {
    if(featuresStart != -1)
        return(entry.substring(featuresStart, featuresStop));
    else
        return("");
} // getFeatures

public String getExample() {
    if(exampleStart != -1)
        return(entry.substring(exampleStart, exampleStop));
    else
        return("");
} // getExample

public String getGP() {
    if(gpStart != -1)
        return(entry.substring(gpStart, gpStop));
    else
        return("");
} // getGP

public String getLF() {
    if(lfStart != -1)
        return(entry.substring(lfStart, lfStop));
    else
```

```
        return("");
    } // getLF

    public String getComment() {
        if(commentStart != -1)
            return(entry.substring(commentStart, commentStop));
        else
            return("");
    } // getComment

    public String getSubKey() {
        return(subKey);
    } // getSubKey

    // public access methods
    // writing values
    public void setLex(int start, int stop) {
        lexStart = start;
        lexStop = stop;
    } // setLex

    // ...
    // currently set directly (direct access to variables)
} // class
```

B Die Klasse singleDico.java

```

import java.util.*;

public class singleDico {

    Hashtable
        entries = null,          // (1)
        features = null,        // (2)
        LFid = null,            // (3)
        LFvalue = null,         // (4)
        LFidKey = null,         // (5)
        LFvalueKey = null,      // (6)
        LFvalueKey2 = null,     // (6.2)
        LFidKeyRev = null,     // (7)
        LFvalueKeyRev = null,  // (8)
        LFvalueKeyRev2 = null; // (8.2)
    Vector
        keysSorted = null,      // (9)
        featuresSorted = null,  // (10)
        LFidSorted = null,      // (11)
        LFvalueSorted = null,   // (12)
        LFvalueSorted2 = null;  // (12.2)

    /*
    notion:
    - "{ element }":
      a java.util.Vector() consisting of zero or more elements
    - "id: key --> data_object"
      identifier id of a java.util.Hashtable()
      access key used in ht.get(key)
      data_object returned by ht.get(key) which is either the name
      of a class (if no curly braces {} are used) or a Vector
      consisting of some elements (if these braces are used)

    explanation of data fields:

    1: entries: key --> singleEntry
    2: features: feature --> { key }
    3: LFid: LFid --> LFkey.tmp
       LFkey.tmp: key --> { LFvalue }
       LFid * key --> { LFvalue }
    4: nested hash table:
       LFvalue: LFvalue --> LFid.tmp
       LFid.tmp: LFid --> { key1 }
       results in:
       LFvalue * LFid --> { key1 }
    5: LFidKey: LFid --> { key }

```

```
6: LFvalueKey: LFvalue --> { key }
6.2: LFvalueKey2: LFid + ":" + LFvalue --> { key }
7: LFidKeyRev: key --> { LFid }
8: LFvalueKeyRev: key --> { LFvalue }
8.2: LFvalueKeyRev: key --> { LFid + ":" + LFvalue }
```

```
9: { key }
10: { feat }
11: { LFid }
12: { LFvalue }
12.2: { LFid + ":" + LFvalue }
```

the same numbering scheme is also used in class Dico.java
in methods addSE() and removeSE()

```
*/
```

```
String
```

```
    dir, file; // directory and filename (latter unimportant)
```

```
public singleDico() { // constructor
```

```
    // nothin'
```

```
    } // singleDico
```

```
} // class
```

C Grammatik für die Ausdruckssuche

Folgende (leicht vereinfachte) Grammatikdefinition wird der Suche nach Ausdrücken (expression search) zugrundegelegt:

```
// binäre Operatoren
AND → "&", AND2 → ";"
OR → "|", OR2 → ","

// unäre Operatoren
NOT → "-", NOT2 → "~", NOT3 → "!"

// Tests auf Enthaltensein in Strings
EQ → "=", EQ2 → "=="
UNEQ → "<>", UNEQ2 → "!="
LT → "<", LTEQ → "<="
GT → ">", GTEQ → ">="

LEX → "LEX", LEX2 → "LEMMA", LEX3 → "L"
DEF → "DEF", DEF2 → "D"
FEAT → "FEATURES", FEAT2 → "F"
EXAMPLE → "EXAMPLE", EXAMPLE2 → "E"
LFid → "LFid", LFid2 → "ID"
LFvalue → "LFvalue", LFvalue2 → "VAL"
COMMENT → "COMMENT", COMMENT2 → "C"

// linke Seiten von Valenzenschemata
X → "X", Y → "Y", Z → "Z", W → "W"
I → "I", II → "II", III → "III", IV → "IV"
NR → "#"

ALPHA → ["a" - "z", "A" - "Z", "_", "."]
NUM → ["0" - "9"]
SPECIAL → "!" | "?" | "$" | "
" | "/" | "+" | "*"
ANYCHAR → <ALPHA> | <NUM> | <SPECIAL>
STRING → (<ANYCHAR>)+

searchExpr → search() <EOL>

search() →
  term() (
    ( x = <OR> | x = <OR2>) term()
    {
      // poppe v1, poppe v2
      // pushe Vereinigung von v1 und v2
    }
  )*

term() →
  unary() (
```

```

    ( x = <AND> | x = <AND2>) unary()
    {
        // poppe v1, poppe v2
        // pushe Schnitt von v1 und v2
    }
)*

```

```

unary() →
    <NOT> element()
    {
        // poppe v
        // pushe Komplement von v
    }
|
    element()

```

```

element() →
    ls() eqRel() rs()
    {
        // keine Aktion; rs() pushed stattdessen
    }

    | ls1() (<EQ> | <EQ2>) ls2() (<EQ> | <EQ2>)
    {
        // ... }
    rs()
| lsNR() rel() rs()
| “(” search() “)” // Sprung in die Rekursion

```

```

ls() →
    (<LEX> | <LEX2> | <LEX3> | <DEF> | <DEF2>
    | <FEAT> | <FEAT2> | <EXAMPLE> | <EXAMPLE2>
    | <LFid> | <LFid2> | <LFvalue> | <LFvalue2>
    | <COMMENT> | <COMMENT2>
    | <X> | <Y> | <Z> | <W>
    | <I> | <II> | <III> | <IV>)
// (ohne <NR>)

```

```

ls1() →
    (<X> | <Y> | <Z> | <W>)

```

```

ls2() →
    (<I> | <II> | <III> | <IV>)

```

```

lsNR() →
    <NR>

```

```

rel() →

```

```
( <EQ> | <EQ2> | <UNEQ> | <UNEQ2>
  | <LT> | <LTEQ> | <GT> | <GTEQ> )
```

```
eqRel() →
  ( <EQ> | <EQ2> | <UNEQ> | <UNEQ2> )
```

```
rs() →
  rsTerm() (
    ( x = <OR> | x = <OR2> ) rsTerm()
    {
      // poppe v1, poppe v2
      // pushe Vereinigung von v1 und v2
    }
  )*
```

```
rsTerm() →
  rsUnary() (
    ( x = <AND> | x = <AND2> ) rsUnary()
    {
      // poppe v1, poppe v2
      // pushe Schnitt von v1 und v2
    }
  )*
```

```
rsUnary() →
  <NOT> rsElement()
  {
    // poppe v
    // pushe Komplement von v
  }
|
  rsElement()
```

```
rsElement() →
  ( <STRING>
    {
      rs = token.image; // speichere nur den String-Wert
    }
    ( <STRING>
      {
        rs += + token.image;
      }
    )*
```

```
| “ ” <STRING> // siehe Nutshell 1.1, S. 23
  {
    rs = token.image; // speichere nur den String-Wert
  }
```

```

    (<STRING>
    {
        rs += + token.image;
    }
    )*      " "
)
{
    pushSet(ls, ls2, rel, rs);
}

| "(" rs() ")" // Sprung in die Rekursion

```

“ls” steht für “left side”, “eqRel” steht für “equality relation”, “rs” steht für “right side”. Die Tokens <X>, <Y>, ..., <GTEQ> stehen für die Aktanten (“#” speziell für deren Anzahl) bzw. die Relationen (letztere “greater than or equal” >=).

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt und nur die angegebenen Quellen benutzt zu haben.

(Klaus Hildner)