Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Document
D-93-08

Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse

Thomas Kieninger, Rainer Hoch

Mai 1993

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
D-6750 Kaiserslautern
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, SEMA Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Computer Linguistics
- Programming Systems
- Deduction and Multiagent Systems
- Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Friedrich J. Wendl
Director

Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse

Thomas Kieninger, Rainer Hoch

DFKI-D-93-08

Diese Arbeit wurde durch das Bundesministerium für Forschung und Technologie (FKZ ITW-9003 0) finanziell unterstützt.

Zusammenfassung

Die vorliegende Arbeit befaßt sich mit der Konzeption eines strukturierten *Lexikons*, welches den Zugriff auf Wörter und deren Information auf zwei grundsätzlich verschiedene Weisen erlaubt. Zum einen erfolgt der Zugriff über ein *3-faches Hashing*, das in Abhängigkeit vom Grad der Erkennung der einzelnen Zeichen automatisch eine der drei verfügbaren Hashfunktionen wählt [1]. Dabei kann der Benutzer zusätzliches Wissen in Form sogenannter *Sichten* zur Beschleunigung der Anfrage bzw. zur Einschränkung der zurückgelieferten Wortmenge einfließen lassen. Zum anderen ist die Benennung einer Sicht möglich. Diese Art des Zugriffs liefert alle Einträge zurück, die unter dem entsprechenden *Sichtnamen* zusammengefaßt worden sind.

Die Realisierung solcher *Sichten* ist ein wesentlicher Bestandteil des Systems, da das somit implizierte Wissen von den Benutzern auf vielfältige Art genutzt werden kann: einerseits kann die Mitgliedschaft eines Eintrages zu einer Sicht in Erfahrung gebracht werden, andererseits hat man die Möglichkeit, Sichten und deren logische Verknüpfung zur direkten Abfrage bzw. zur Einschränkung des Suchraumes zu verwenden. Zum Aufbau der dafür benötigten komplexen Struktur des *virtuell partitionierten Lexikons* wird eine aufwendige Generierungsphase notwendig.

Die vorliegende Arbeit beschreibt das zugrundegelegte Konzept und seine Vorteile gegenüber anderen Lösungsansätzen sowie den benötigten Generator und ein Laufzeitsystem, das umfangreiche Zugriffe auf dem virtuell partitionierten Lexikon erlaubt.

[1] Hierzu kann ein regulärer Ausdruck als Eingabe spezifiziert werden. Die darin erlaubten Wildcards sind '*', '?' in ihrer üblichen Verwendung und in Brackets ('[' und ']') eingeschlossene Zeichenalternativen für eine Position.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Moderne Rechenanlagen haben in der heutigen Zeit Einzug in fast alle Branchenbereiche gefunden. Textverarbeitung mittels Computer ist bereits eine Selbstverständlichkeit geworden. Mit der Steigerung der Rechenleistung und der Vernetzung von Arbeitsplatzrechnern nahm auch die zu verarbeitende Information zu. Einige Fachleute prognostizierten Mitte der 70er Jahre das „papierlose Büro", in dem Dokumente ausschließlich elektronisch vorliegen und auf ergonomischen Displays dem Menschen präsentiert werden. Die Praxis zeigt jedoch, daß das Papier nicht aus dem Büroalltag zu verdrängen ist. Im Gegenteil, der Papierkonsum ist heute höher als je zuvor und die Produktion steigt um 10 bis 15% jährlich.

Eine Verbindung zwischen Computer und Papier besteht derzeit zum größten Teil in der Erzeugung von Dokumenten mit Hilfe des Rechners. Die Umkehrung dieses Prozesses, das Einlesen von Druckerzeugnissen und die anschließende Archivierung oder Weiterverarbeitung der gewonnenen Daten, ist heute ein weit verbreiteter Forschungsbereich.

Zielsetzung des ALV-Projektes ist es, eine Verbindung zwischen gedruckten Dokumenten und Computer zu bilden. Dabei möchte man bildhafte Informationen eines eingescannten Briefes zunächst in eine ASCII-Repräsentation überführen (Texterkennung), um aus den so gewonnenen Daten in einem weiteren Schritt (Textanalyse) einfache semantische Informationen zu extrahieren und das Dokument gegebenenfalls zu klassifizieren. Es ist auch eine automatische Postverteilung denkbar, indem das System den Empfänger identifiziert. Abbildung 1.1 zeigt grob die Architektur des ALV-Systems. Der hier verfolgte Ansatz beschränkt sich exemplarisch auf Geschäftsbriefe, deren Layout- und Logikstruktur mit Hilfe des ODA-Standards (Office Document Architecture) [ISO8613] modelliert werden kann. Die bildhafte Information eines eingescannten Briefes wird zunächst durch die „Layout-Extraktion" in zusammengehörige, geometrische Bereiche wie Blöcke, Zeilen und Buchstaben unterteilt (segmentiert). Die so gewonnenen Layoutblöcke werden dann in der nächsten Phase, dem sogenannten „Logical Labeling", in logische Einheiten wie Adreßteil, Betreff, Datum oder Rumpf gefaßt.

Durch Verschmutzungen oder minderwertige Kopien können jedoch Unsicherheiten bei der Erkennung von Buchstaben entstehen. Beispielsweise können mehrere Alternativen von Buchstaben für eine Stelle von der Zeichenklassifikation vorgeschlagen werden, Buchstaben können aufbrechen („m" wird zu „rn"), mit anderen verkleben oder unkenntlich sein.

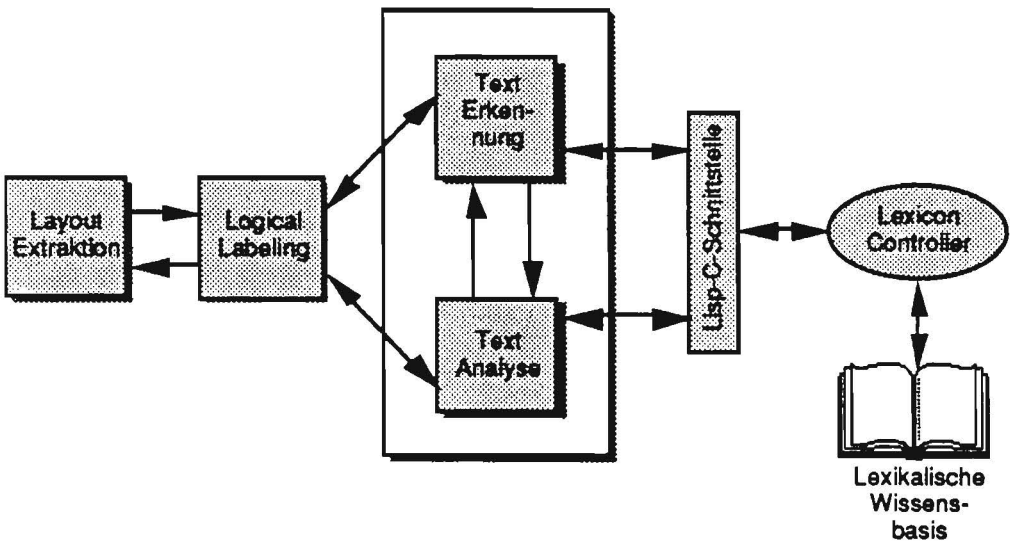

Abbildung 1.1: Architektur des ALV-Dokumentanalysesystems [Hoch92]

Der Lexikonabgleich hilft hier, unkorrekte Alternativen zu falsifizieren bzw. korrekte zu verifizieren (siehe Abbildung 1.2).

Da eine Verbesserung der Erkennungsergebnisse auf traditionellem Wege nach Ansicht einiger Experten gegenwärtig nicht mehr möglich ist [2], versucht man durch einen nachgeschalteten Wörterbuchabgleich die Resultate zu verbessern. Somit wird ein Lexikonsystem zu einer zentralen Komponente des Gesamtsystems.

Dieses Wörterbuch muß einerseits im Stande sein, vollständig erkannte Wörter als im Wortschatz vorkommend zu bestätigen und andererseits auch „passende" Vorschläge bei unvollständiger Erkennung zu machen. Bei vollständig erkannten, aber fehlerhaft geschriebenen Wörtern soll es in der Lage sein, Korrekturvorschläge zu finden.

Abbildung 1.3 zeigt die verschiedenen Stufen der Texterkennung, ausgehend von der bildhaften Darstellung des eingescannten und um eventuellen Drehwinkel der Vorlage korrigierten Briefes. Man erkennt die mit jedem Schritt der Segmentierung feiner werdende Struktur und die anschließende Texterkennung auf Wortebene. Das so entstandene Buchstaben-Hypothesennetz ist die Eingabe an das Lexikon, welches alle gültigen Wort-Alternativen zurückgibt.

In der Textanalyse werden Dokumente auf verschiedene Arten untersucht:

- Das System **INFOCLAS** sucht im Text nach bedeutungstragenden Wörtern und führt mittels statistischer Methoden eine Klassifikation des Briefes in verschiedene *Nachrichtentypen* gemäß des EDIFACT Standards [ISO9735] durch (siehe [Dittrich92]).

[2] Dies gilt insbesondere dann, wenn die Qualität der Eingabe schlecht ist (z.B. durch geringe Auflösung bei Faxen), oder wenn sehr kleine Schriften vorliegen. Probleme bereiten ferner Ligaturen, Verklebungen oder Schmutz.

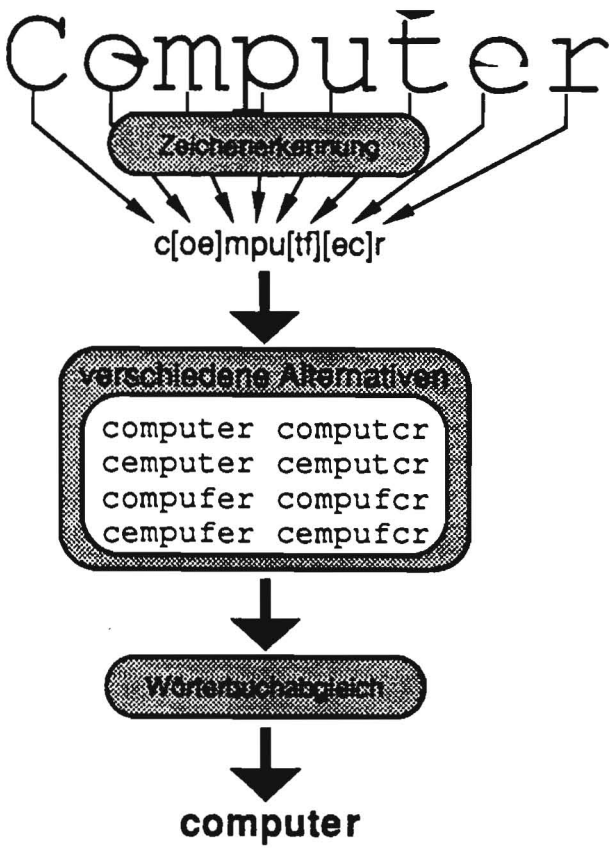

Abbildung 1.2: Verbesserung der Erkennungsergebnisse durch Lexikonabgleich

- Mit Hilfe von **Skimming-Techniken** kann ein Brief gewissermaßen „überflogen" werden, d.h. es werden nicht alle Wörter betrachtet, sondern nur eine Auswahl. Aus diesen Wörtern wird ebenfalls eine Nachricht extrahiert.
- Der **Insel-Parser** [Kirchmann93] analysiert Sätze auf ihre Syntax hin und kann so in Rückkoppelung mit anderen Systemen Aufschluß über die Art (Substantiv, Verb, Adjektiv etc.) bestimmter Wörter geben.

Jedes dieser Verfahren ist darauf angewiesen, daß zu den einzelnen Wörtern eine verwertbare Semantik oder syntaktische Information zur Verfügung steht, was wiederum die Existenz eines Lexikons voraussetzt.

Es ist zweckmäßig, für Texterkennung und Textanalyse (im folgenden *Analysesysteme* genannt) ein gemeinsames Wörterbuch zu konzipieren, da sie auf den gleichen Wortschatz zurückgreifen. Eine doppelte Datenhaltung kann dadurch vermieden werden. Unterschiede bestehen lediglich in der Art der Anfrage und der benötigten Information.

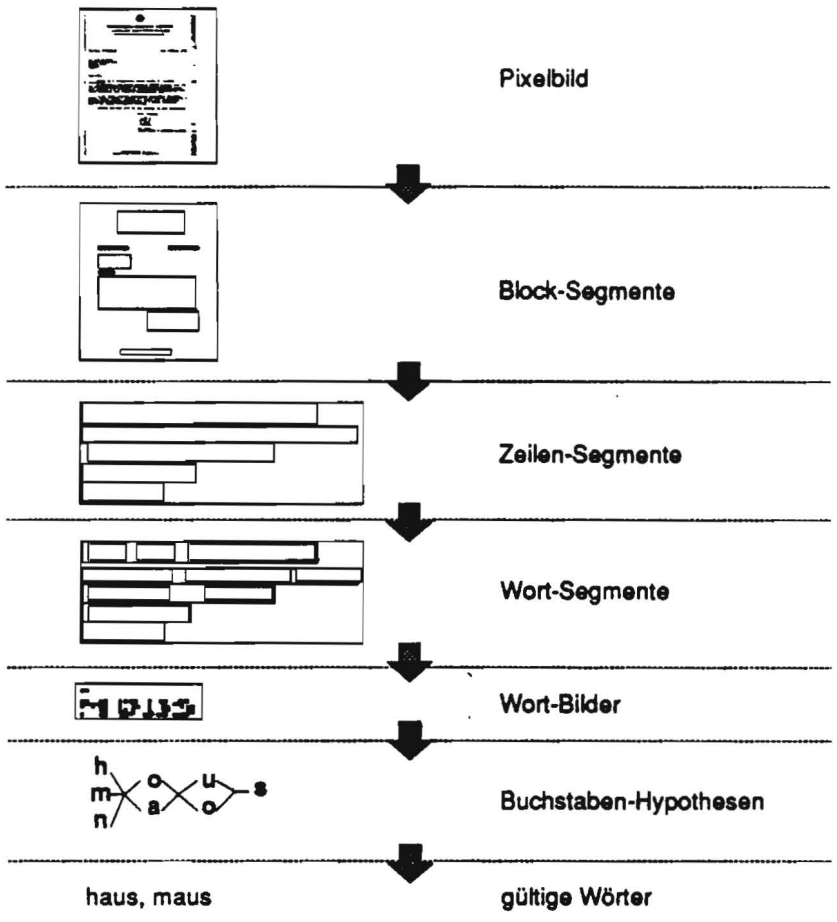

Abbildung 1.3: Analysephasen der Texterkennung [Dengel92]

Die vorliegende Diplomarbeit befaßt sich mit der Konzeption eines solchen Wörterbuches, welches sich neben den bereits erwähnten Anforderungen noch durch folgende Punkte auszeichnen soll:

- Schnelligkeit
- geringen Hauptspeicherbedarf
- leichte Wartbarkeit
- Modularität
- Erweiterbarkeit

Der Wunsch nach Schnelligkeit ist rasch erklärt: die Texterkennung greift für jedes Wort mindestens einmal auf das Lexikon zu, um erkannte Wörter zu verifizieren und macht das Lexikon so zu einer zeitkritischen Komponente. Die Textanalyse beschränkt sich unter Umständen auf bestimmte Wörter (z.B durch Text Skimming), wobei die Zugriffszeit

weniger kritisch ist. In diesem Fall ist das Bereitstellen lexikalischer Information wesentlich wichtiger. Dennoch hängt die Gesamtlaufzeit (Texterkennung plus Textanalyse) wesentlich von der Anfragedauer des Lexikons ab.

Die Forderung nach geringem Hauptspeicherbedarf resultiert aus der Tatsache, daß das Lexikon nur ein „Assistent" der Analysesysteme ist, die ihrerseits speicherresident sind. Würde man den Wortschatz ebenfalls im Arbeitsspeicher organisieren, so benötigte das System zusätzlich sehr viele Ressourcen. Als Folge wäre das Gesamtsystem entweder nur auf sehr großen Rechnern lauffähig, oder es würde die Performanz durch häufiges Auslagern speicherresidenter Information auf Dateien (Paging) erheblich leiden.

Leichte Wartbarkeit ist eine Anforderung, die ohnehin an alle Datenbanksysteme (und um ein solches handelt es sich beim Lexikon im weitesten Sinne) gestellt wird. Typisch für Wartungen sind z.B. Korrekturen oder Erweiterungen des Wortschatzes.

Die Modularität resultiert aus dem Wunsch, das Gesamtsystem in verschiedenen Domänen einsetzen zu wollen. Jede Branche und jeder Betrieb haben teilweise unterschiedliches Vokabular. Dies kann durch die Namen der Mitarbeiter, Kunden, Produkte, angebotenen Dienstleistungen usw. zustande kommen. Des weiteren ist es denkbar, einen speziellen Experten für die Analyse von Adreßteilen zu entwickeln, der mit einem verhältnismäßig kleinen Wortschatz auskommt. In jedem dieser Fälle muß lediglich ein Teil des Lexikons durch einen entsprechend anderen ersetzt (Datei mit Mitarbeiternamen) oder Teile hinzugefügt bzw. entfernt werden, um für eine spezielle Anwendung ein „maßgeschneidertes" Lexikon zu erhalten.

Das Lexikonsystem ist auch als Stand-Alone Anwendung oder in ein anderes System integriert denkbar. In diesem Fall kann selbstverständlich auch die zugrundeliegende Sprache (hier deutsch) variiert werden.

Unter Erweiterbarkeit versteht man die Möglichkeit, z.B. den Zugriff auf die Daten des Lexikons durch neue (Hash-) Funktionen zu ergänzen, (neue) Querverweise innerhalb der Einträge zu realisieren oder die möglichen Informationen, die das Lexikon zu einem Wort trägt, zu erweitern. Man möchte das System gewissermaßen „offen" halten für neue Erkenntnisse und Verfahren.

Der Aufbau dieser Dokumentation gliedert sich wie folgt:

- Zunächst werden verschiedene *Organisationsformen* und *Datenstrukturen* für Wörterbücher vorgestellt und ihre Vor- und Nachteile herausgestellt.
- Im Anschluß wird das daraus resultierende *Hashverfahren* genauer betrachtet und unterschiedliche Arten der Berechnung von Hashcodes und die damit verbundenen Probleme erläutert.

  Ferner werden *Qualitätsmerkmale* von Hashfunktionen besprochen und graphisch veranschaulicht.
- Danach wird der zentrale Begriff der *Sichten* und ihre Verwirklichung in Verbindung mit dem Hashverfahren vorgestellt.
- Der folgende Abschnitt befaßt sich mit dem *Generator* und seinen Ein-/ Ausgaben.
- Kapitel 6 beschreibt das *Laufzeitsystem* mit seinen unterschiedlichen *Anfragemechanismen* und dem *Filtersystem.*
- Schließlich werden einige Besonderheiten der *Implementierung* diskutiert und einige *Datenstrukturen* und *Algorithmen* vorgestellt.
- Ein Kapitel mit *Testergebnissen* beendet den Hauptteil der Dokumentation.
- Anhang A stellt die *Lisp-Schnittstelle* vor, welche es ermöglicht, die volle Funktionalität des Laufzeitsystems von Lisp-Programmen aus zu nutzen.
- Anhang B ist als Tutorial zur *Generierung eines Wörterbuches* zu verstehen. An einem kleinen Beispiel werden alle Einzelschritte erklärt.
- Abschließend zeigt Anhang C eine beispielhafte Lisp-Sitzung, die den Gebrauch des gerade generierten Wörterbuches verdeutlicht.

Kapitel 2

Organisationsformen lexikalischen Wissens

Wie bereits in der Einleitung erwähnt, werden von den Analysesystemen unterschiedliche Anforderungen an das Lexikon gestellt. Es gilt nun, eine geeignete Datenstruktur zu finden, mit deren Hilfe man den genannten Restriktionen gerecht wird.

Um für eine gegebene Zeichenkette zu entscheiden, ob er ein gültiges Wort repräsentiert, muß man überprüfen, ob der String in einer vorgegebenen Wortliste enthalten ist. Das Problem verlagert sich also auf die Darstellung einer Menge von Zeichenketten, unserem Wortschatz oder Lexikon.

2.1 Ansätze zur Realisierung eines Lexikons

Da es sich bei einem realistischen Lexikon um Datenmengen handelt, deren Mächtigkeit relativ groß ist (>10.000 Einträge), ist eine speicherresidente Repräsentation der einzelnen Wörter nicht vertretbar. Es gibt drei grundsätzlich verschiedene Weisen, diesem Problem zu begegnen [Eliman90]:

1. Mit Hilfe von **probabilistischen Modellen** wird der Wortschatz auf gewisse Eigenschaften (Features) hin analysiert. Auf diese Eigenschaften, die relativ kompakt darstellbar sind, werden potentielle Wörter untersucht. Der Vergleich dieser Merkmale des Kandidaten mit denen aus dem Lexikon (ggf. Verrechnung von Einzelwahrscheinlichkeiten) gibt mit mehr oder weniger großer Sicherheit Auskunft darüber, ob ein Wort Element des Lexikons ist oder nicht.

   Durch den relativ geringen Speicherplatz dieser Verfahren kann im Primärspeicher gearbeitet werden, was Plattenzugriffe überflüssig und die Berechnung der Wortmitgliedschaft schnell macht.

2. Das zweite Verfahren zur Ermittlung gültige Wörter, ist der **explizite Vergleich** mit einem zugrundegelegten Wortschatz. Diese Methode liefert nicht nur exakte Ergebnisse, indem sie jede Buchstabenfolge eindeutig dem Lexikon zuordnet oder

zurückweist, sondern sie bietet auch die Möglichkeit, einem Wort Informationen zuzuordnen. Der Relationencharakter des Lexikons wird hierbei sehr deutlich.

3. Als weiterer Punkt sei noch die Möglichkeit eines **hybriden Ansatzes** erwähnt. Dabei werden probabilistische Methoden und expliziter Wörterbuchabgleich miteinander kombiniert, um sich der Vorteile beider Verfahren zu bedienen.

   Für unsere Zwecke könnte dies z.B. bedeuten, daß bei Präsenz mehrerer Wortalternativen einige direkt durch binäre n-Gramme (siehe Abschnitt 2.2 - Punkt 2) verworfen werden und evtl. nur noch eine Alternative übrigbleibt. Die verbleibende (reduzierte) Anzahl potentieller Kandidaten wird schließlich durch den Wörterbuchabgleich verifiziert bzw. falsifiziert.

Da die probabilistischen Modelle in Tests unzulängliche Resultate lieferten (siehe Abschnitt 8.2.4 und [Riseman74]), und die Analysesysteme von unserem Lexikon die Bereitstellung wortspezifischer Daten fordern, sind wir bei der Konzeption unseres Systems an den expliziten Vergleich mit einem Wörterbuch gebunden. Einige Zugriffsformen hierfür sollen im folgenden Abschnitt kurz beschrieben werden.

2.2 Wahrscheinlichkeitsmodelle

Im folgenden werden einige probabilistische Ansätze vorgestellt:

1. Eines dieser Verfahren verwendet **n-Gramme** als charakteristische Eigenschaft eines Wortschatzes [Peterson80] [Takahashi90]. In einer Initialisierungsphase ermittelt das System anhand einer Wortliste die Häufigkeiten bestimmter Buchstabenkombinationen der Länge n (sogenannter n-Gramme). Ein potentielles Wort wird auf seine n-Gramme hin untersucht und aus den einzelnen Übergangswahrscheinlichkeiten wird eine Gesamtwahrscheinlichkeit des Kandidaten für seine Existenz im Wortschatz errechnet. Für $n = 2$ spricht man von Bigrammen; für $n = 3$ von Trigrammen. Wählt man n größer, so bekommt man genauere Ergebnisse, andererseits steigt die Anzahl möglicher Kombinationen exponentiell an, was die zu speichernde Datenmenge vergrößert. Ferner werden bei ungenauer Erkennung einzelner Zeichen weniger n-Gramme des Kandidaten verfügbar, was die Konfidenz der berechneten Gesamtwahrscheinlichkeit schwächt.

   Liefert die Texterkennung beispielsweise mehrere Alternativen $(w_1, w_2, \ldots, w_n)$ für ein Wort, so könnte dieses System aufgrund der Wahrscheinlichkeiten $(p(w_1), p(w_2), \ldots, p(w_n))$ eine nach Auftrittswahrscheinlichkeit sortierte Liste zurückliefern.

2. Anwendung von Trigrammen in Texterkennungssystemen ergaben jedoch keine signifikanten Verbesserungen der Ergebnisse [Riseman74]. Aus diesem Grund ging Riseman zu **binären n-Grammen** über. Diese können nur die Werte 0 und 1 annehmen: 1 falls ein n-Gramm gültig ist, 0 sonst. Dadurch nehmen die Gesamtwahrscheinlichkeiten für ein potentielles Wort ebenfalls Werte zwischen 0 und 1 an (1 falls alle n-Gramme des Wortes gültig sind, 0 sonst).

   Um über die Integration dieses Verfahrens in unser Konzept zu entscheiden, wurden im Rahmen dieser Diplomarbeit Tests mit binären Trigrammen und den nach Meier 8000 häufigsten deutschen Wörter [Meier78] als Basis durchgeführt. Die Ergebnisse waren jedoch sehr unbefriedigend, da gezielt fehlerhaft geschriebene Wörter (in den meisten Fällen) nicht zurückgewiesen werden konnten. Einige Beispiele aus diesen Tests sind im Anhang zu finden.

3. Eine Erweiterung obiger Methode auf **positionale, binäre n-Gramme** ist laut [Hull88] in der Lage, Fehler zu korrigieren. Hier werden Buchstabenübergänge zwischen beliebigen Positionen in einem Wort gesammelt und getrennt betrachtet. Ein kleines Beispiel mit Bigrammen soll dies verdeutlichen. Unser Lexikon bestünde aus den Wörtern { cat, cot, tot } und wird durch die drei positionsabhängigen Übergangsmatrizen $d_{1,2}$, $d_{1,3}$ und $d_{2,3}$ dargestellt. Bei Eingabe „coo“ geschieht folgendes: Die Matrizen werden auf die Eingaben hin untersucht: $d_{1,2}(c, o) = 1$, $d_{1,3}(c, o) = 0$ und $d_{2,3}(o, o) = 0$.

   Da $d_{1,3}$ und $d_{2,3}$ negative Ergebnisse lieferten wird die Schnittmenge der Indizies $\{1, 3\} \cap \{2, 3\} = \{3\}$ gebildet. Dies ist die fehlerhafte Position, die es nun zu korrigieren gilt. Wir suchen dazu alle Buchstaben α mit $d_{1,3}(c, \alpha) = 1$ und $d_{2,3}(o, \alpha) = 1$.

Wörterbuch = {cat, cot, tot}

	a	c	o	t
a	0	0	0	0
c	1	0	1	0
o	0	0	0	0
t	0	0	1	0

$d_{1,2}$

	a	c	o	t
a	0	0	0	0
c	0	0	0	1
o	0	0	0	0
t	0	0	0	1

$d_{1,3}$

	a	c	o	t
a	0	0	0	1
c	0	0	0	0
o	0	0	0	1
t	0	0	0	0

$d_{2,3}$

Abbildung 2.1: Übergangsmatrizen positionaler, binärer Bigramme

Dies ist in unserem Fall nur das 't'. Folglich ist „cot" der einzige Korrekturvorschlag aus dem Basislexikon.

Dieses Verfahren hat jedoch auch Schwachpunkte. So fallen zum einen doch recht große Datenmengen zur Speicherung der Features an, zweitens können durch einen sehr großen Wortschatz sehr viele Einzelwahrscheinlichkeiten gleich 1 sein und dadurch auch fehlerhafte Worthypothesen akzeptiert werden. Schließlich werden bei Verklebungen und Aufbrechungen die Positionen der nachfolgenden Buchstaben nicht korrekt bestimmt, was die Berechnung der Korrekturvorschläge verfälscht.

4. Eine weitere probabilistische Methode soll an folgendem Beispiel veranschaulicht werden: zehn **unabhängige Hashfunktionen** $(h_1, h_2, \ldots, h_{10})$ bilden Zeichenketten auf den Bereich 1 bis 20.000 ab. Ferner haben wir einen Bitvektor B der Länge 20.000, dessen Elemente zunächst alle gleich 0 sind. Nun werden die Wörter W_i des Lexikons (10.000 Einträge) eingefügt, indem jeweils die zehn Hashwerte $h_j(W_i)$ berechnet und die entsprechenden Bits im Vektor $B[h_j(W_i)]$ auf 1 gesetzt werden. Um ein Wort x auf Mitgliedschaft im Lexikon zu testen, werden die zehn Hashcodes $(h_j(x))$ berechnet und jeweils das entsprechende Bit des Vektors $B[h_j(x)]$ untersucht. Ist eines der Bits gleich 0, so ist das Wort definitiv nicht im Lexikon enthalten; sind alle Bits gleich 1, so kann man mit hoher Wahrscheinlichkeit (> 99.9%) sagen, daß das Wort im Lexikon enthalten ist [Nix81].

2.3 Wörterbuchkonzepte

In diesem Abschnitt werden einige beispielhafte Verfahren vorgestellt, die eine explizite Speicherung des Wortschatzes realisieren:

1. Die Wörter des Lexikons können in einer **sequentiellen Liste** organisiert sein. Im Falle unvollständiger Erkennung müßte diese Liste sequentiell durchlaufen und die Einträge mit der Eingabe verglichen werden [3]. Bei vollständiger Erkennung und lexikographisch sortierter Liste kann der Zugriff durch eine binäre Suche geschehen [4].

2. Die Einträge können auch in den Knoten eines **Binärbaumes** gespeichert werden. Als Ordnungsrelation bei der Suche wird die lexikographische Ordnung verwendet. Ein Spezialfall eines binären Suchbaums ist der **AVL-Baum**, der nach jeder Einfüge- bzw. Löschoperation ausbalanciert wird und dadurch bei einer Lexikongröße von N Einträgen höchstens $O(\log_2 N)$ Suchschritte (= maximale Tiefe) benötigt, um ein vollständig erkanntes Wort zu finden [Knuth73] [Comer79].

3. Bei sehr großen Datenmengen bietet sich der **B-Baum** an, da er einer Organisation der Daten auf Datei entgegenkommt. Im B-Baum hat ein innerer Knoten in der Regel weit mehr als zwei Söhne, was die maximale Höhe und somit die Anzahl der Suchschritte bei gegebener Lexikongröße reduziert [Comer79].

   Ein generelles Problem bei Bäumen ist die Suche nach unvollständigen Wörtern. Soweit der Wortanfang ausreichend erkannt wurde, um einen lexikographischen Unterschied beim Vergleich festzustellen, kann man bei der Suche im Baum auf Nachfolgerknoten verzweigen. Aber ab einem gewissen Grad muß der komplette Teilbaum nach Kandidaten durchsucht werden. Ist bereits der Wortanfang unbekannt, so müßte man folglich den gesamten Baum durchlaufen, was sich negativ auf die Performanz niederschlägt.

4. Bei der **mehrstufigen Indizierung** werden die ersten Buchstaben des Suchstrings sukzessive für eine Verzweigung auf eine weitere Stufe verwendet. Ab einer vorgegebenen Stufe wird der so „eingekreiste" Wortschatz linear durchsucht. In den einzelnen Verzweigungsstufen kann die Anzahl der Nachfolger variieren, so daß z.B. in der ersten Stufe jedem Buchstaben ein Verweis zukommt, in der zweiten aber nur noch für je zwei Zeichen ein Nachfolger vorgesehen ist [Harris85].

5. Als Weiterführung der mehrstufigen Indizierung kann der **Buchstabenbaum** (auch **Trie** genannt) angesehen werden. Dabei handelt es sich um eine Baumstruktur, in der, von einem Wurzelknoten ausgehend, für jeden Buchstaben des Alphabets ein Nachfolger möglich ist.

   Jeder Zeichenfolge ist hier eindeutig ein Pfad im Baum zugeordnet. Alle Knoten besitzen eine Wortende-Markierung, die, sofern sie gesetzt ist, ein gültiges Wort auf diesem Pfad signalisiert. Man erkennt leicht, daß mit jedem zusätzlich erkannten

[3] Die 'grep'-Familie von Unix-Kommandos bietet diese Möglichkeit der sequentiellen Suche eines Musters in einer Datei an. Dabei können reguläre Ausdrücke als Eingabe spezifiziert werden.

[4] Das Unix-Kommando 'look' durchschaut eine Datei alphabetisch sortierter Wörter mittels binärer Suche nach einem spezifizierten Eintrag.

Buchstaben, vom Wortanfang aus gesehen, der Suchraum sukzessive eingeschränkt wird und der Trie somit bei bruchstückhafter Information sehr gut geeignet ist, auf eine Eingabe passende Wörter zu finden. Ein Problem bereiten Unsicherheiten am Wortanfang. Hier ist der Trie in seiner „Urversion" ungeeignet. Im Rahmen des ALV-Projektes wurden jedoch Verbesserungen am Trie vorgenommen, so daß man über eine selektive Einstiegsmatrix auch dann an der richtigen Stelle mit der Suche ansetzen kann, wenn der Anfang des Wortes ungenau ist [Pleyer91].

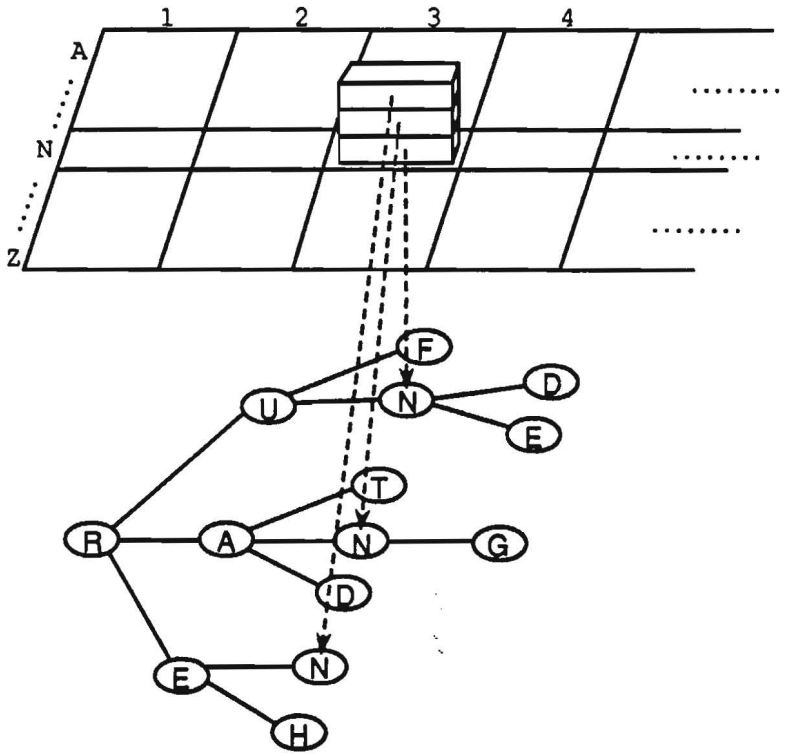

Abbildung 2.2: Trie mit selektiver Zugriffsmatrix [Dengel92]

6. Eine weitere Methode, große Datenmengen zu verwalten, ist die Verwendung einer (relationalen) **Datenbank.** Doch die Einträge in einem Wörterbuch sind nicht von homogener Gestalt. Da für alle Elemente einer Relation die gleichen Attribute gelten, wären bei den meisten Einträgen viele Informationszellen unbelegt, was eine ungewünschte Vergößerung der Dateien zur Folge hätte.

   Darüberhinaus sind Lexikoneinträge oftmals *rekursiv* definiert und damit von ihrer Länge unbegrenzt, was erhebliche Probleme bei der Modellierung eines Datenbankschemas aufwirft [Ide91]. Die Einträge des Lexikons lassen sich folglich nicht in geeigneter Weise in ein Relationenschema übertragen.

   Ein weiteres Problem stellt die Suche nach Wörtern bei unvollständiger Information dar. Hierfür müßten die charakteristischen Merkmale eines Wortes (z.B. Wortanfang oder Wortende) als weiterer Suchschlüssel definiert werden.

7. Schließlich sind noch **Hashverfahren** zu erwähnen. Bei diesem Verfahren wird aus dem Suchbegriff x eine Adresse $h(x)$ berechnet und über diese Adresse auf eine indizierte Tabelle T zugegriffen. Dort ist im Falle direkter Adressierung unmittelbar die gesuchte Information zu finden; bei indirekter Adressierung steht ein Verweis auf eine weitere Tabelle oder Liste, die die eigentlichen Daten trägt [Knuth73][Maurer75][Wagner92].

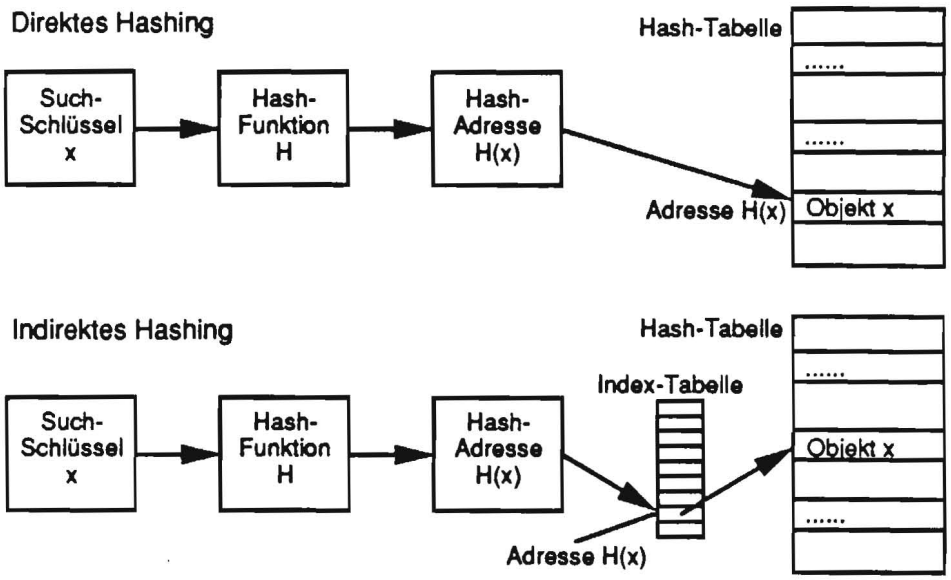

Abbildung 2.3: Direktes und indirektes Hashing

Die Berechnung der Adresse durch die *Hashfunktion* h kann auf unterschiedliche Weisen geschehen. Dabei sind für die Wahl der Funktion h nicht zuletzt die Merkmale (Features) des zu suchenden Objekts entscheidend. Nun kann es vorkommen, daß unterschiedliche Elemente in ihren Features ununterscheidbar sind und so durch die Hashfunktion auf dieselbe Adresse abgebildet werden. Wir sprechen hier von *Kollisionen.* Diese stellen ein Problem beim Hashverfahren dar, welches auf zwei Arten gelöst werden kann.

(a) Ist eine Adresse bereits durch ein anderes Objekt belegt, so kann man beim Einfügen (bzw. Suchen) auf benachbarte Adressen ausweichen. Dieses Ausweichen muß kontrolliert geschehen, man spricht auch von *Sondieren.* Beispielhaft hierfür sind *lineares* und *quadratisches* Sondieren [Maurer75].

(b) Bei der zweiten Lösung werden die Elemente, die auf die gleiche Adresse abgebildet werden, zu einer *Liste* zusammengeführt. Sucht man nun nach einem Objekt, so muß die Liste an der entsprechenden Hashadresse durchlaufen werden.

Entscheidend für die Häufigkeit von Kollisionen sind neben der Anzahl der Objekte, die auf die gleiche Adresse abgebildet werden, auch die Länge der Hashtabelle, da die Adressen unter Umständen erst durch eine weitere Transformation auf den In-

dexbereich der Tabelle abgebildet werden müssen. Diese Abbildung kann z.B. durch das *Divisionsrestverfahren* geschehen ($h'(x) = h(x) \bmod l_T$).

2.4 Organisation des Lexikon-Systems

Um die Tatsache zu berücksichtigen, daß der größte Teil der Wörter in einem Dokument aus einem relativ kleinen Spektrum der deutschen Sprache stammen, ist eine hybride Organisation des Wörterbuches denkbar. Dabei kann der häufiger frequentierte, kleinere Teil des Wortschatzes im Arbeitsspeicher organisiert sein, was sich in verkürzten Zugriffszeiten bei einem Großteil der Anfragen auswirkt. Schlägt die Suche in diesem Hochfrequenz-Wörterbuch fehl, so wird der seltener verwendete, aber sehr viel größere Teil des Lexikons, welcher auf Dateien liegt, durchsucht.

Für den hochfrequenten Teil des Lexikons bietet sich der Trie an. Speziell durch die Einstiegsmatrix wird er zu einem sehr hilfreichen Werkzeug bei der Suche nach unvollständig erkannten Wörtern und hat damit entscheidende Vorteile gegenüber binären Suchbäumen.

In der momentanen Implementierung sind wir jedoch von der Integration eines hochfrequenten Teilwörterbuches abgekommen, da das Lexikon nahezu ausschließlich mit Unvollständigkeiten konfrontiert wird und in diesem Fall ohnehin auch der niederfrequente Teil nach möglichen matchenden Wörtern durchsucht werden müßte. Ein Zeitvorteil käme also nicht zustande.

Den Kern unseres Systems bildet somit ein Verfahren, welches seine Daten auf Dateien organisiert, um den Arbeitsspeicherbedarf gering und vom Wortumfang unabhängig zu halten. Wir entschieden uns für einen hashbasierten Ansatz, bei dem über mehrere unabhängige Hashfunktionen (je nach Art der verfügbaren Information) auf die Daten zugegriffen werden kann.

Dadurch erhalten wir eine ideale Kombination aus minimaler Anzahl der Leseoperationen (Vorteil gegenüber sequentiellen Listen und Bäumen bzw. Tries) einem möglichen Zugriff auf die Daten trotz Unsicherheiten in der Eingabe dank speziell hierfür entwickelter Hashfunktionen (Vorteil gegenüber binärer Suchbäume und Datenbanken) und variable Datensatzlänge (Vorteil gegenüber Datenbanken).

Jeder dieser Hashfunktionen ist eine eigene Tabelle mit gleichzeitig eigener Nachfolgerverzeigerung zur Kollisionsbehandlung zugeordnet [Wagner92]. Diese Struktur behandelt Kollisionen also durch Listenbildung. Die einzelnen Hashfunktionen berechnen ihre Werte aus verschiedenen Merkmalen eines Wortes. Im Falle vollständig erkannter Wörter ist es die Zeichenkette selbst, bei fehlender Information ist es die erste bzw. zweite Hälfte des Wortes (bis zu einer vorgegebenen Länge). Dabei werden die Buchstabenfolgen als Zahl interpretiert. Die Lettern entsprechen den Ziffern, ihre Werte werden durch ihre Ordnung im Alphabet bestimmt.

Beispiel: Hashcode-Berechnung des Wortes „Hand".
Mächtigkeit des Alphabetes inclusive Umlaute und 'ß' = 30.

$$\begin{aligned} Ordnung(\text{'}H\text{'}) &= 7 \\ Ordnung(\text{'}A\text{'}) &= 0 \\ Ordnung(\text{'}N\text{'}) &= 13 \\ Ordnung(\text{'}D\text{'}) &= 3 \end{aligned}$$

Die einzelnen Funktionswerte berechnen sich wie folgt:

$$\begin{aligned} h_{complete}(\text{'HAND'}) &= 7*30^3+0*30^2+13*30^1+3*30^0 &&= 189.393 \\ h_{begin}(\text{'HAND'}) &= 7*30^1+0*30^0 &&= 210 \\ h_{end}(\text{'HAND'}) &= 13*30^1+3*30^0 &&= 393 \end{aligned}$$

Dabei bezeichnet $h_{complete}$ die Funktion für vollständig erkannte Wörter, h_{begin} für den Wortanfang und h_{end} für das Wortende. Weiter gilt:

$$\begin{aligned} h_{begin}(\text{'HAND'}) &= h_{complete}(\text{'HA'}) \\ h_{end}(\text{'HAND'}) &= h_{complete}(\text{'ND'}) \end{aligned}$$

Nähere Informationen hierzu finden sich in einer ebenfalls im Rahmen des ALV-Projektes durchgeführten Diplomarbeit [Wagner92]. Aus einem darin dokumentierten Test geht hervor, daß bei geeigneter Wahl von Hashfunktion und Hashtabellen-Größe mit einer durchschnittlichen Kollisionskettenlänge kleiner zwei bei vollständig erkannten Wörtern zu rechnen ist.

Kapitel 3

Hashfunktionen – Kriterien der Güte

Eine Hashfunktion ist eine Abbildung der Menge der Schlüssel auf den Bereich der Hashadressen. Wichtige Anforderungen bei Hashfunktionen sind:

1. Einfache und schnelle Berechnung des Hashcodes.
2. Weite Streuung der Adressen (Surjektivität) in einen gegebenen Adreßbereich, um eine hohe Auslastung der Tabelle zu gewährleisten (ideale Werte sind $> 90\%$).
3. Gleichmäßige Abbildung in den Adreßbereich, um Kollisionen möglichst zu vermeiden oder gering zu halten.

Oftmals lassen sich nicht alle drei Punkte gemeinsam befriedigen. Daher ist je nach Art der Anwendung zu entscheiden, welches der Kriterien am wichtigsten ist. Da wir bei der Konzeption immer einen schnellen Zugriff als Anforderung zu berücksichtigen haben, ist Punkt 1 für uns relativ wichtig. Andererseits fallen auch komplexe Berechnungen im Vergleich zu Leseoperationen auf Dateien relativ schwach ins Gewicht, was die Wichtigkeit dieses Punktes wieder mindert.

Zu Punkt zwei muß man sagen, daß geringe Auslastungen zu leeren Informationszellen in der Hashtabelle führen, d.h. daß evtl. viel Speicherplatz für diese Tabellen allokiert, aber nur ein geringer Teil tatsächlich genutzt wird. In unserer Entwicklungsumgebung werden pro Hashadresse vier Byte benötigt, was bei Tabellengrößen im Bereich von 10.000 bis 20.000 angesichts der zur Verfügung stehenden Speicherressourcen zu vertretbarem Platzbedarf führt und die Auslastung ebenfalls zu einem sekundären Entscheidungsmerkmal macht.

Wichtiger ist für uns viel mehr die Berücksichtigung von Punkt drei. Lange Kollisionsketten sind gleichbedeutend mit hohen mittleren Zugriffszeiten. Wir sprechen im folgenden von der *Anzahl der Gruppen* $G_l = |h(W)|$, in die eine Hashfunktion h den ihr eingegebenen Wortschatz W bei gegebener Tabellenlänge l abbildet. (In der obigen Gleichung bezeichnet $h(W)$ den Bildbereich von h für W.)

Die Kollisionsketten K haben eine mittlere Länge von $K_l = |W|/G_l$. Die Auslastung A_l der Tabelle berechnet sich als $A_l = G_l/l$. Man sieht sofort, daß mit wachsender Anzahl Gruppen G_l sowohl K_l fällt als auch A_l steigt. Unser Ziel ist also die Suche von Hashfunktionen, welche bei gegebenen Wortschatz möglichst viele Gruppen erzeugen.

Da Hashfunktionen im allgemeinen nicht sofort Werte innerhalb der gültigen Adreßbereiches erzeugen, muß dies durch eine zweite Abbildung geschehen. Dazu wählten wir in unserem System das *Divisionsrestverfahren* $h(x) = h_0(x) \bmod l$ (h_0 ist die „Ur"-Hashfunktion, h die auf den Adreßbereich 0 bis $l-1$ korrigierte), bei dem die Ausgangswerte auf einen beliebigen Bereich abgebildet werden können.

Ein weiteres Problem ist die Wahl einer geeigneten Tabellenlänge, da durch das Divisionsrestverfahren alle $h_0(x)$ im Abstand l auf die gleiche Adresse $h(x)$ abgebildet werden und somit zusätzliche, „künstliche" Kollisionen entstehen.

$$|h_0(x_1) - h_0(x_2)| = k * l \quad \rightarrow \quad h(x_1) = h(x_2) \qquad k \in I\!N$$

Demzufolge können geringe Änderungen der Tabellenlänge l überraschend starke Unterschiede der Auslastung A_l und der mittleren Kollisionskettenlänge K_l bewirken. Diese Problematik wird in Abschnitt 3.2 genauer betrachtet.

3.1 Hashfunktionen – eine Auswahl

Bei der Wahl der Hashfunktionen findet man in der Literatur eine Vielzahl von Ansätzen. Hier sollen nur einige genannt werden:

1. Beim Verfahren der **signifikantesten Buchstaben** [Takahashi90] werden aus elektronisch vorliegenden (repräsentativen) Texten die Häufigkeiten der Buchstaben ermittelt. Danach werden die Zeichen nach ihrer Häufigkeit fallend sortiert und man erhält die Reihenfolge ihrer Signifikanz. (Wir haben zu Tests die Reihenfolge aus der Deutschen Sprachstatistik von Meier [Meier78] entnommen.)

   Nun werden aus dem Eingangswort der Hashfunktion die ersten n signifikantesten Buchstaben ermittelt. (Bei unseren Tests war $n = 3$.) Diese Buchstaben konnten mit einer Abbildung h_{sig} auf $I\!N$ transformiert werden (gegebenenfalls auch Divisionsrestverfahren). Damit ist die die Berechnung beendet.

   Diese Funktion ist relativ schnell berechenbar, jedoch kann bei Nichterkennen eines der signifikanten Buchstaben eines Wortes der charakteristische String und somit auch die richtige Adresse nicht korrekt bestimmt werden.

2. Eine weitere Methode stellen die Auswertung von **Ober- und Unterlängen** eines Wortes dar. Man spricht hier auch vom *Word-Envelope*. Dieser Berechnung liegt die Idee zugrunde, die Buchstaben des Alphabetes danach zu klassifizieren, ob sie den sogenannten Kernbereich über- oder unterschreiten [Sinha90]. Man erhält danach drei Buchstabenklassen.

$$\begin{aligned} M_1 &= \{a, c, e, m, n, o, r, s, u, v, w, x, z\} \\ M_2 &= \{b, d, f, h, i, k, l, t, \ddot{a}, \ddot{o}, \ddot{u}, ß\} \\ M_3 &= \{g, j, p, q, y\} \end{aligned}$$

Die Berechnung des Hashcodes wollen wir anhand eines Beispiels betrachten. Wir betrachten dazu das Wort „synthetisch". Im ersten Schritt ersetzen wir jeden Buchstaben durch den Index seiner Klasse und erhalten „1312122112". Die zweite Stufe entfernt mehrfach aufeinanderfolgende gleiche Ziffern durch eine einzige, das Ergebnis lautet „13121212". Diese Ziffernfolge charakterisiert die optische Umrandung des Wortes, daher auch der Name „Word-Envelope". Durch eine nicht triviale, jedoch recht schnell durchführbare Berechnung kann jede dieser theoretisch möglichen Ziffernfolgen durch eine injektive Funktion auf $I\!N$ abgebildet werden. Diese injektive Funktion soll anhand von Abbildung 3.1 beschrieben werden: Die erste Ziffer kann aus der Menge $\{1, 2, 3\}$ stammen, entsprechend verzweigt man auf die erste Stufe im Baum der Graphik. Von nun an hat ein Knoten nur noch je zwei Nachfolger, da im Word-Envelope ja keine gleichen Ziffern aufeinanderfolgen. Für jede Ziffer im charakteristischen String wird eine Stufe weiter im Baum verzweigt. Die Durchnumerierung (Zahlen links der Knoten) entspricht dem Bild eines Word-Envelopes, der zu einem Knoten führt, durch diese Hashfunktion. Die Verbindung der Einzelschritte stellt unsere Funktion h_{env} dar.

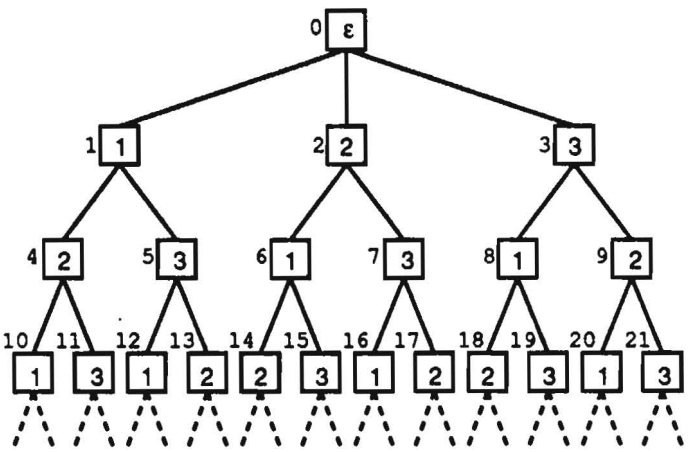

Abbildung 3.1: Transformation des Word-Envelope nach $I\!N$

Der Vorteil dieses Verfahrens liegt darin, derartig verfügbares Wissen der Texterkennung bei unvollständiger Erkennung ausnutzen zu können. Darüber hinaus hat man

eine gewisse Robustheit gegen Aufbrechungen oder Verklebungen von Buchstaben der gleichen Klasse.

Man muß diesem Verfahren auch eine gewisse Fontunabhängigkeit zugestehen, da in nahezu allen üblichen Zeichensätzen die gleiche Einteilung erfolgt. Doch hier verlassen wir rasch unser Diskussionsfeld, da die Problematik unterschiedlicher Fonts einem anderen Gebiet zuzuordnen ist.

Tests haben allerdings gezeigt, daß das Verfahren einen zugrundegelegten Wortschatz in verhältnismäßig wenige Gruppen einteilte, was lange Kollisionsketten zur Folge hat. Dies macht die Funktion für unseren Anwendungszweck eher ungeeignet.

Ein weiterer Nachteil ist die Handhabung von Großbuchstaben. Am Satzanfang werden bekanntlich alle Wörter groß geschrieben, wodurch ein völlig anderer Envelope entsteht. Das gleiche Problem tritt auf, wenn ein Wort zur HERVORHEBUNG durchgehend groß geschrieben wird. Solche Wörter könnten mit diesem Verfahren nicht gefunden werden – und gerade hier handelt es mit hoher Wahrscheinlichkeit um bedeutungstragende Begriffe.

Schließlich stellt der Buchstabe 'j' ein Problem dar, da er nicht eindeutig einer Klasse zuzuordnen ist. Für ihn müßte streng genommen eine weitere Klasse eingeführt werden, was in Bezug auf die Berechnung eine gleichmäßige Füllung der Hashtabellen unwahrscheinlicher werden läßt.

3. Eine weitere Hashfunktion h_{konv} berechnet ihren Wert, indem die Buchstaben als **Ziffern zur Basis 30** und das Eingangswort entsprechend als Zahl zu dieser Basis interpretiert werden. Diese Funktion ist schon in Abschnitt 2.4 näher beschrieben worden.

   Auf ihr aufbauend läßt sich gleich eine Gruppe von Hashfunktionen definieren. Man muß dazu lediglich die Eingabe vorverarbeiten. Alle drei Funktionen im Lexikon-System beruhen auf diesem Prinzip.

   - Funktion h_1 (oder $h_{complete}$) für vollständig erkannte Wörter ist identisch zu h_{konv}. (Der Index 'konv' steht für *konvertieren*, da es sich dabei prinzipiell nur um eine Konvertierung von einem Zahlensystem in ein anderes handelt.)
   - Die Werte für den Wortanfang errechnen wir mit h_2 (oder h_{begin}). Dazu wird die zweite Hälfte (bei ungerader Wortlänge die größere Hälfte) des Eingabewortes abgeschnitten und im Bedarfsfall auf die Länge vier weiter gekürzt. Die so entstandene Zeichenkette dient als Eingabe zu h_{konv}, die schließlich den Hashcode liefert.
   - Analog verfährt die Hashfunktion h_3 (oder h_{end}) für das Wortende. In diesem Fall wird lediglich die erste Hälfte der Eingabe abgeschnitten und ansonsten wie bei h_2 verfahren.

   Ein klarer Pluspunkt von h_{konv} ist die einfache Berechenbarkeit. Außerdem bilden unsere drei Funktionen h_1, h_2 und h_3 gewissermaßen eine Familie, in der relativ leicht ein Favorit für eine Anfrage zu berechnen ist.

   Weiter ist eine Umstellung unseres Systems auf eine andere maximale Worthälftenlänge sehr einfach möglich. Man erkauft sich dabei kürzere Kollisionsketten auf

Kosten mehrerer Alternativen (falls im Bereich unserer Teilinformation Unsicherheiten auftreten) oder umgekehrt. Der Wert vier als idealer Kompromiß geht aus Versuchen mit diesem und anderen Werten hervor, die von A. Wagner im Rahmen seiner Diplomarbeit durchgeführt wurden [Wagner92].

Die folgenden Abschnitte sollen die Güteeigenschaften der soeben vorgestellten Hashfunktionen $h_{complete}$, h_{begin}, h_{end}, h_{sig} und h_{env} im Hinblick auf die Verwendung im Lexikon herausstellen.

3.2 Hashtabellen Länge – auf der Suche nach dem Optimum

Die Länge der Hashtabellen ist ein interessanter Punkt, da sie aus Gründen des Speicherbedarfs möglichst klein gehalten werden soll, andererseits aber groß genug sein sollte, um die durch das Divisionsrestverfahren zusätzlich entstehenden Kollisionen gering zu halten. Beide Ziele lassen sich also nicht gemeinsam befriedigen und man ist gezwungen, einen Kompromiß zu finden.

Bei der Hashfunktion 1 ($h_{complete}$) ist dies kein Problem. Sie verteilt ihre Werte recht gut und bildet so viele Gruppen, daß man sich – übertrieben ausgedrückt – die durchschnittliche Kollisionskettenlänge K_l aussuchen kann. Dazu dividiert man die Mächtigkeit des Wortschatzes $|W|$ durch K_l. Eine benachbarte Primzahl des Ergebnisses kann als Tabellenlänge für diese Funktion herangezogen werden.

Etwas mehr Beachtung sollte man den übrigen Hashfunktionen schenken, da sie, wie man in Abschnitt 3.3 sieht, ihre Werte nicht sehr kontinuierlich auf die möglichen Adressen verteilen. Darüberhinaus bilden sie sehr viel weniger Gruppen, was auf die verwendeten Features zurückzuführen ist und zu vermehrten Kollisionen führt.

Um hier bei der Wahl der Länge ein Entscheidungskriterium zu erhalten, sind Auslastung A_l und Anzahl Gruppen in Abhängigkeit von der Tabellenlänge in den folgenden Abbildungen 3.2 bis 3.5 abgetragen. Bei den Diagrammen handelt es sich um diskrete Wertepaare, die durch Punkte repräsentiert sind. Die Werte der x-Achsen sind die unterschiedlichen Hashtabellenlängen [5], zu denen die Werte ermittelt wurden. Die Beträge der kleinsten und der größten Länge sind den einzelnen Diagrammen zu entnehmen. Die linke Leiste zeigt die Auslastung der Hashtabelle A_l in Prozent (fallendes Diagramm), die rechte gibt die Anzahl der Gruppen G_l der entsprechenden Tabellenlänge an.

Als Basis zur Ermittlung sämtlicher Werte wurde eine Datei mit 160.015 deutschen Wörtern [CELEX91] verwendet.

[5] Es wurden nur die Primzahlen des Bereichs als Hashlängen verwendet.

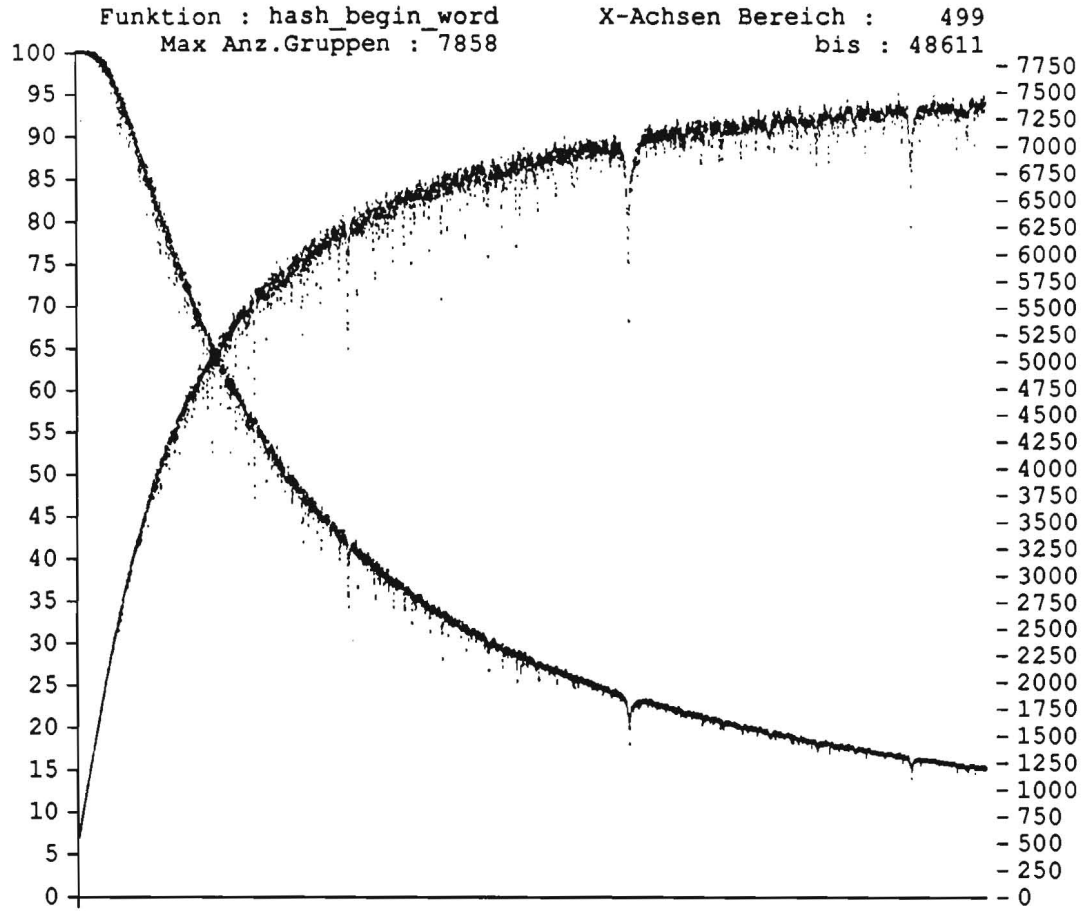

Abbildung 3.2: Auslastung und Gruppenzahl von h_{begin}

Abbildung 3.2 zeigt die Werte für h_{begin}. Man erkennt deutlich den schnellen Anstieg der Gruppenzahl G_l. Der anfangs starke Anstieg der Gruppenzahl flacht immer mehr ab, so daß ab einem bestimmten Punkt keine nennenswerte Verbesserung der mittleren Kollisionslängen zu erwarten ist. Bei der Wahl der Hashlänge gibt dieses Diagramm dem Benutzer groben Aufschluß darüber, wie groß er in etwa seine Hashtabelle wählen muß, um für sich einen optimalen Kompromiß zu erhalten.

Bei Betrachtung der Diagramme fallen auch Einschnitte nach unten sowohl bei A_l als auch bei G_l auf (beide Größen sind ja voneinander abhänging: $A_l = G_l/l$). Es handelt sich dabei um besonders ungünstige Werte für l, da hier offensichtlich sehr viele zusätzliche Kollisionen durch das Divisionsrestverfahren entstehen.

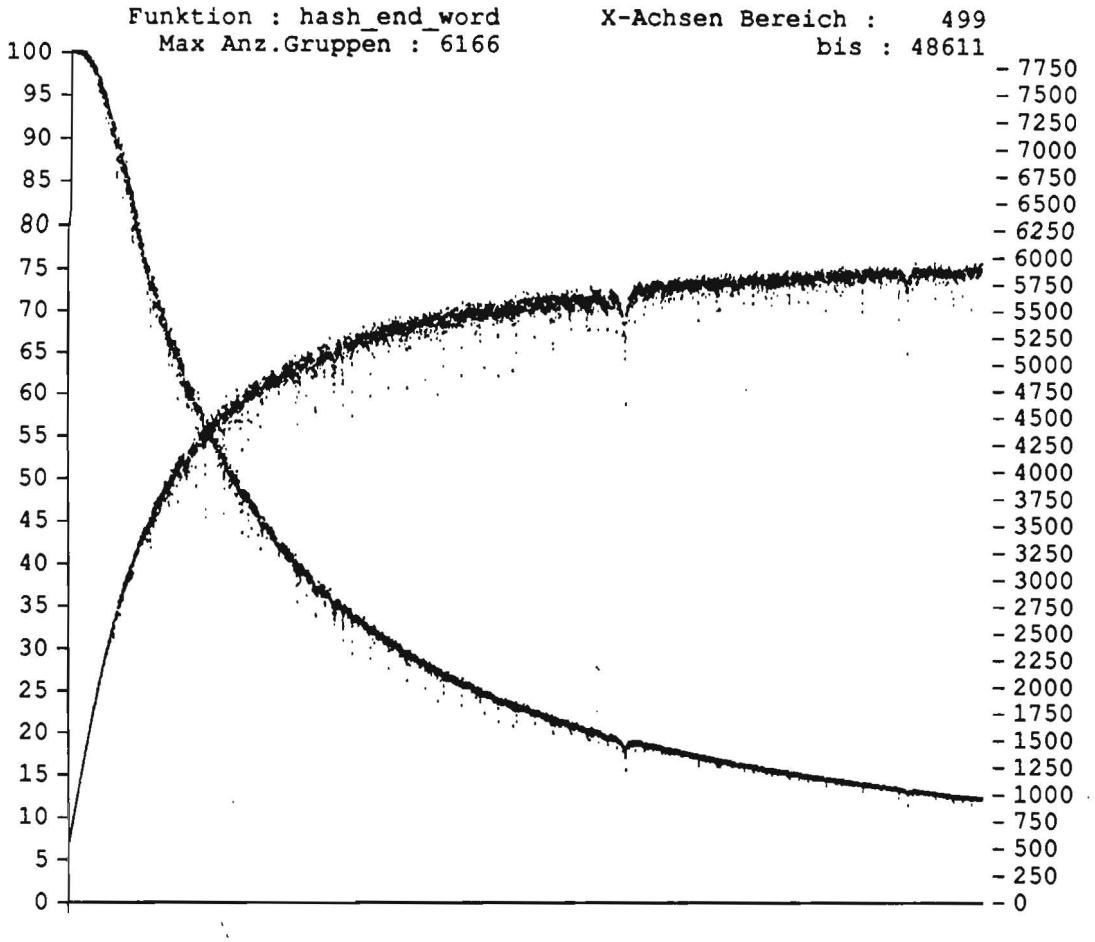

Abbildung 3.3: Auslastung und Gruppenzahl von h_{end}

Abbildung 3.3 zeigt die entsprechenden Werte für die Funktion h_{end}. In dieser Abbildung sind die gleichen Werte für den x-Achsenbereich gewählt. Außerdem hat die rechte Leiste der Gruppenzahl die gleiche Skalierung wie Abbildung 3.2. Dadurch wird der Qualitätsunterschied zwischen h_{begin} und h_{end} bezüglich der Anzahl der Gruppen G_l und somit natürlich auch bezüglich der mittleren Kollisionskettenlänge K_l deutlich.

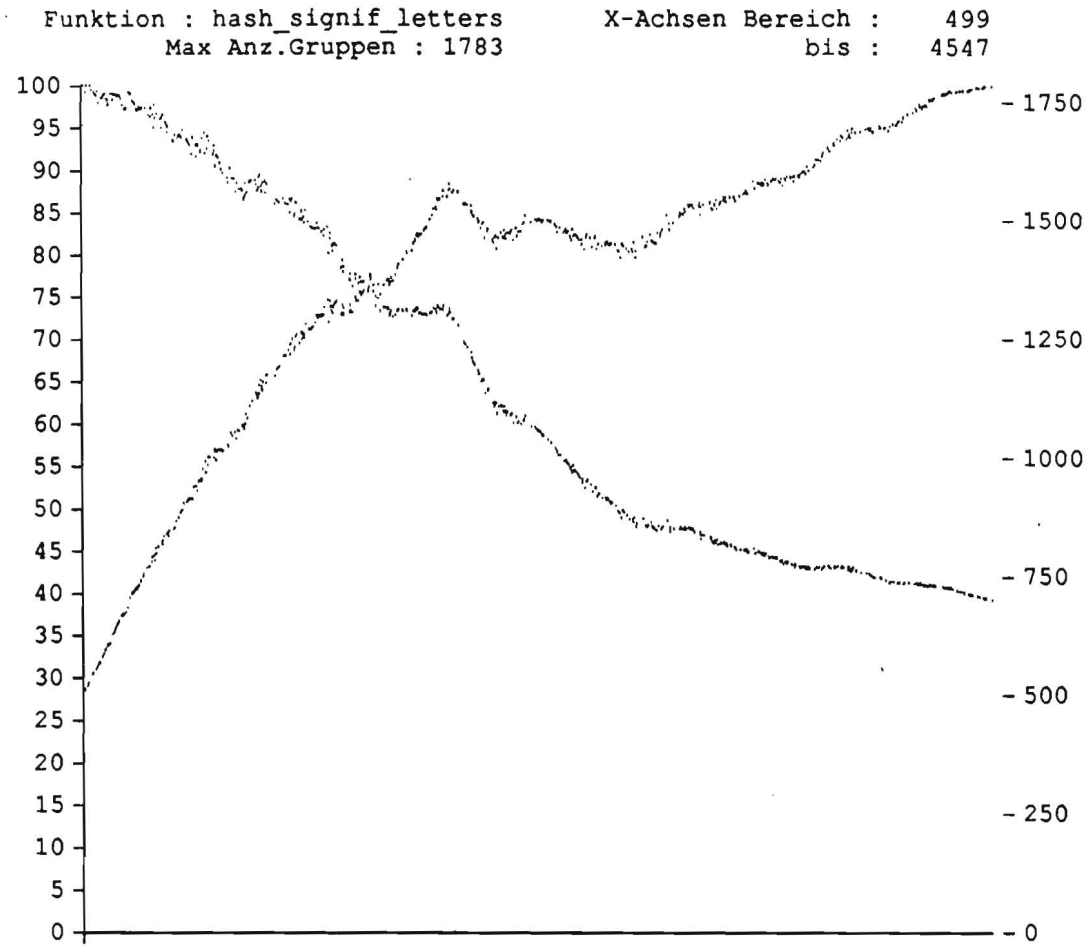

Abbildung 3.4: Auslastung und Gruppenzahl von h_{sig}

Die Werte für die Funktion h_{sig} in Abbildung 3.4 haben einen wesentlich geringeren x-Achsenbereich. Dies kommt daher, daß die Funktion auch ohne Divisionsrestverfahren nur Werte zwischen 0 und 4.547 liefert (vgl. Abschnitt 3.1). Das Diagramm würde also nach rechts verlängert (größeres l) ein konstant bleibendes G_l zeigen und A_l würde stetig fallen ($A_l = G_l/l$). Ein größerer Wert als 4.547 ist nicht nötig, er würde keins der beiden Ziele verbessern, die Auslastung hingegen verschlechtern.

Die maximale Gruppenzahl G_l von 1.783 führt jedoch zu unvertretbar langen Kollisionsketten und damit auch zu langen Anfragezeiten beim Lexikon. Daher ist h_{sig} keine sehr geeignete Funktion für das Lexikon.

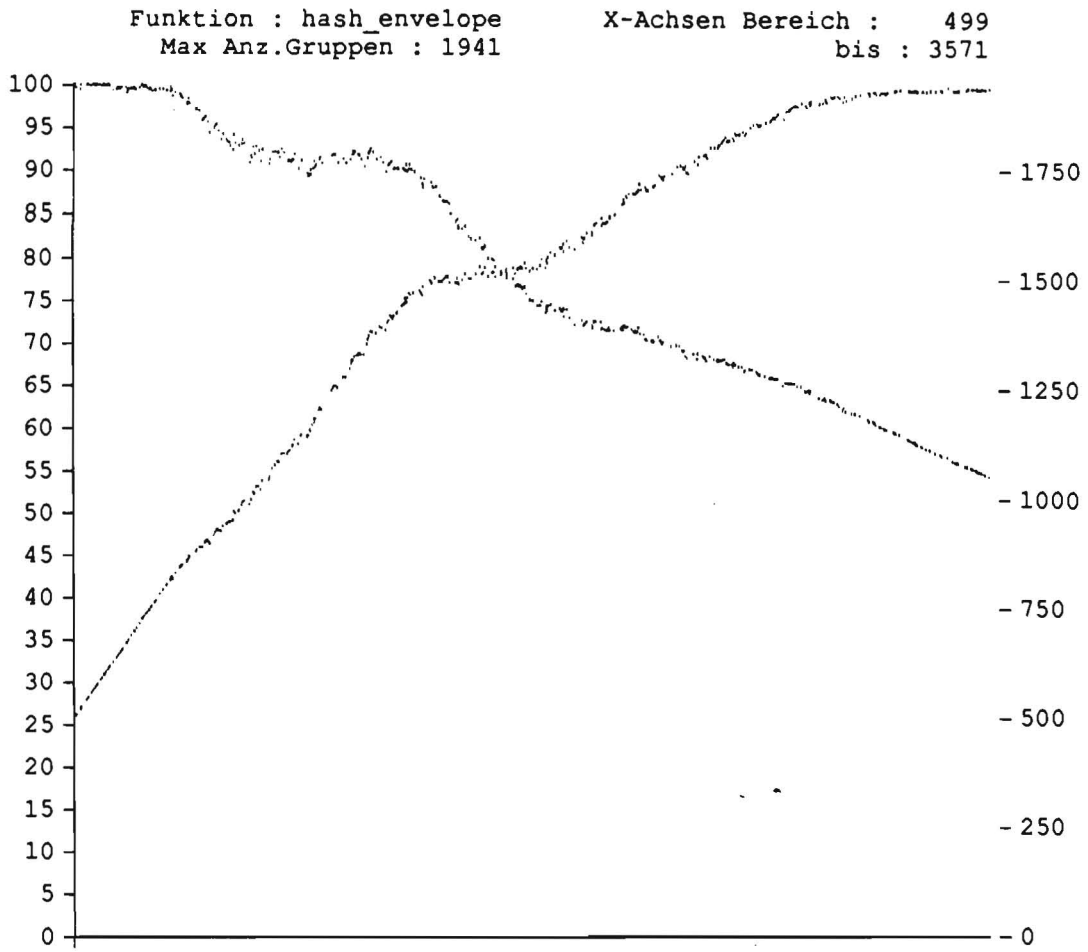

Abbildung 3.5: Auslastung und Gruppenzahl von h_{env}

Das Problem der kleinen Gruppenzahl G_l trifft auch auf die Funktion h_{env} zu (Abbildung 3.5). Genau wie bei h_{sig} würde auch hier eine Verlängerung des Diagramms nach rechts keine weitere Steigerung von G_l mehr zeigen – der maximale Wert von 1.941 ist schon bei einer Länge von 3.571 vorhanden. Daher ist auch diese Funktion nicht sehr gut für den Einsatz im Wörterbuchsystem geeignet.

3.3 Hashfunktionen und die Verteilung ihrer Werte

Nachdem wir uns mit möglichen Tabellenlängen der einzelnen Funktionen befaßt haben, soll dieser Abschnitt ein Bild der Verteilung der Werte durch die einzelnen Funktionen bei festen Hashlängen geben. Zu diesem Zweck verwenden wir die 160.015 Einträge der Datei `worteCELEX` [CELEX91]. Die Abbildungen 3.6 bis 3.10 zeigen die Verteilung der Einträge in den Bildbereich der jeweiligen Hashfunktionen.

Da die x-Achsen sehr große Bereiche überdecken, sind die Verteilungen in mehreren Zeilen dargestellt, um eine bessere Auflösung der Einzelwerte zu erhalten. Die einzelnen Zeilen müssen folglich als ein Diagramm aufgefaßt werden. Darüberhinaus ist auf Grund verschieden langer Tabellen die Skalierung der x-Achsen in den einzelnen Graphiken unterschiedlich.

Die y-Achsen zeigen die Längen der Kollisionsketten an den einzelnen Adressen. Ihre Skalierung ist logarithmisch, um das Bild nicht durch Ausreißer zu verzerren. Sie sind überdies der jeweils längsten Kollisionskette angepaßt, was zu vollkommen verschiedenen y-Achsenbereichen führt ($h_{complete}$:0–10; h_{env}:0–21.256).

Die x-Achsen überdecken den jeweiligen individuellen Adreßbereich der Funktionen. Für $h_{complete}$, h_{begin} und h_{end} haben wir die voreingestellten (Default-) Werte des Generators unseres Systems (siehe Kapitel 5) verwendet. Die Längen für h_{sig} und h_{env} sind die in Abschnitt 3.2 als maximal sinnvoll postulierten Werte.

Abbildung 3.6 zeigt das Diagramm für die Funktion $h_{complete}$. Man erkennt sehr deutlich, wie dicht und gleichmäßig die gesamte Tabelle gefüllt ist. Die maximale Kettenlänge beträgt 10 – das Mittel liegt bei 2 – die Auslastung A_l (im Diagramm als Füllung bezeichnet) ist 79,58%.

Im Diagramm zu h_{begin} (Abbildung 3.7) sieht man große Lücken im ersten Drittel des Adreßbereichs. Die durchschnittliche Kettenlänge ($K_l = 23$) ist aber immer noch vertretbar klein. Ist die Auslastung von 22,54% zu gering, so kann auf Kosten der Kollisionen eine kleinere Tabellenlänge als 29.989 gewählt werden.

Bei h_{end} zeigt sich fast das gleiche Bild (3.8). Nur ist hier die Tabelle nicht ganz so dicht gefüllt wie in 3.7. Die Anzahl der Gruppen ist etwas geringer, was sich auch negativ auf $K_l(= 28)$ und $A_l(= 18,74\%)$ auswirkt.

Abbildung 3.9 zeigt das Diagramm zu h_{sig}. Man sieht sehr deutlich die großen Lücken und die ungleichmäßige Verteilung der Werte auf den Adreßbereich, was bei der relativ kleinen Tabellengröße dennoch zu einer Auslastung von 39,21% führt. Jedoch machen die geringe Anzahl der Gruppen $G_l(= 1.783)$ und die damit verbundene große mittlere Kettenlänge $K_l(= 89)$ die Funktion für den Einsatz bei großen Lexika ungeeignet.

Schließlich zeigt Abbildung 3.10 die Graphik für die Funktion h_{env}. Die extrem lange maximale Kollisionskette (21.256) und die hohe mittlere Kettenlänge ($K_l = 82$) machen auch diese Funktion trotz der durch die geringe Tabellenlänge verursachte hohe Auslastung ($A_l = 54,21\%$) für eine Anwendung im Lexikonsystem ungeeignet.

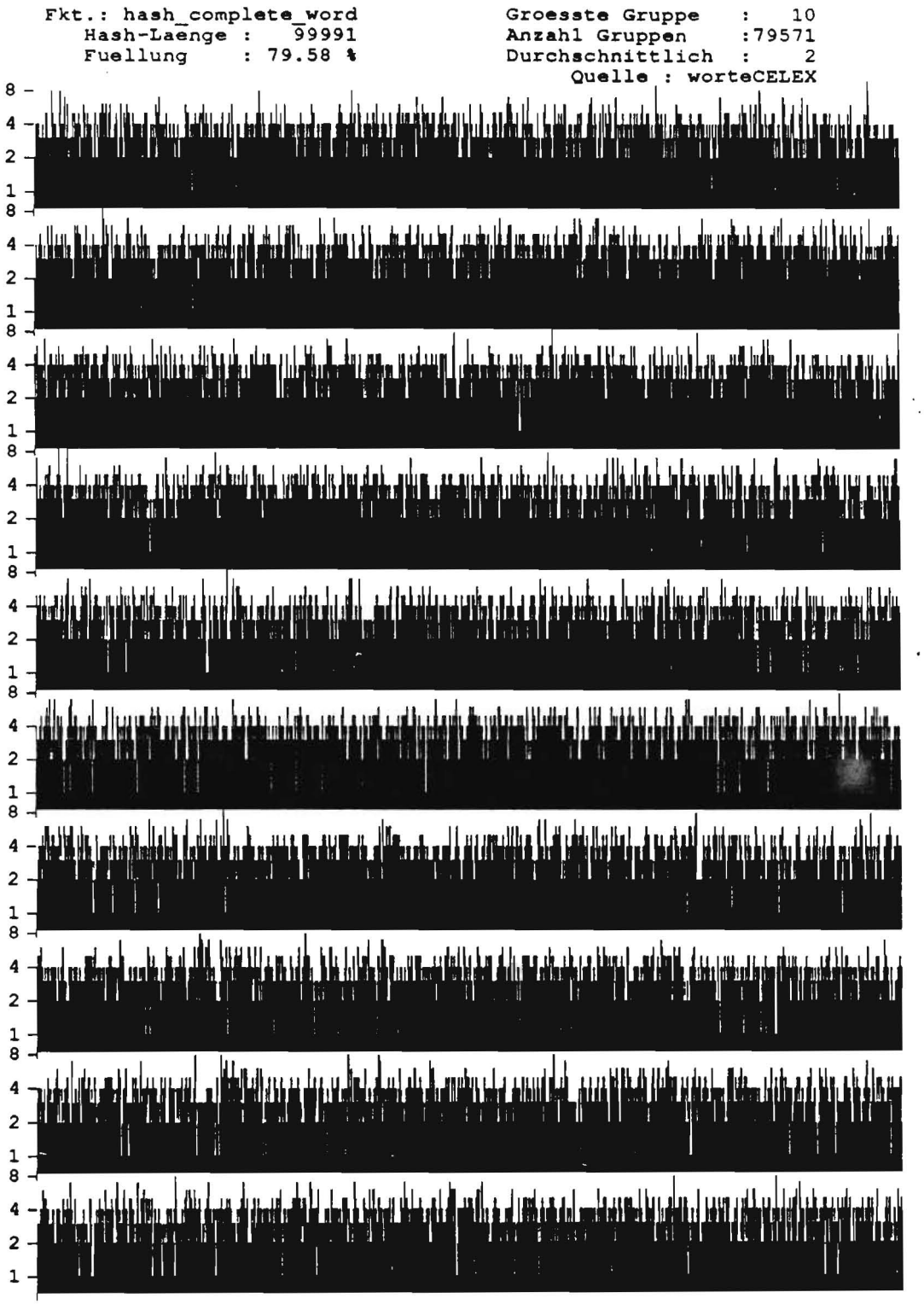

Abbildung 3.6: Verteilung der Wörter auf den Bildbereich von $h_{complete}$

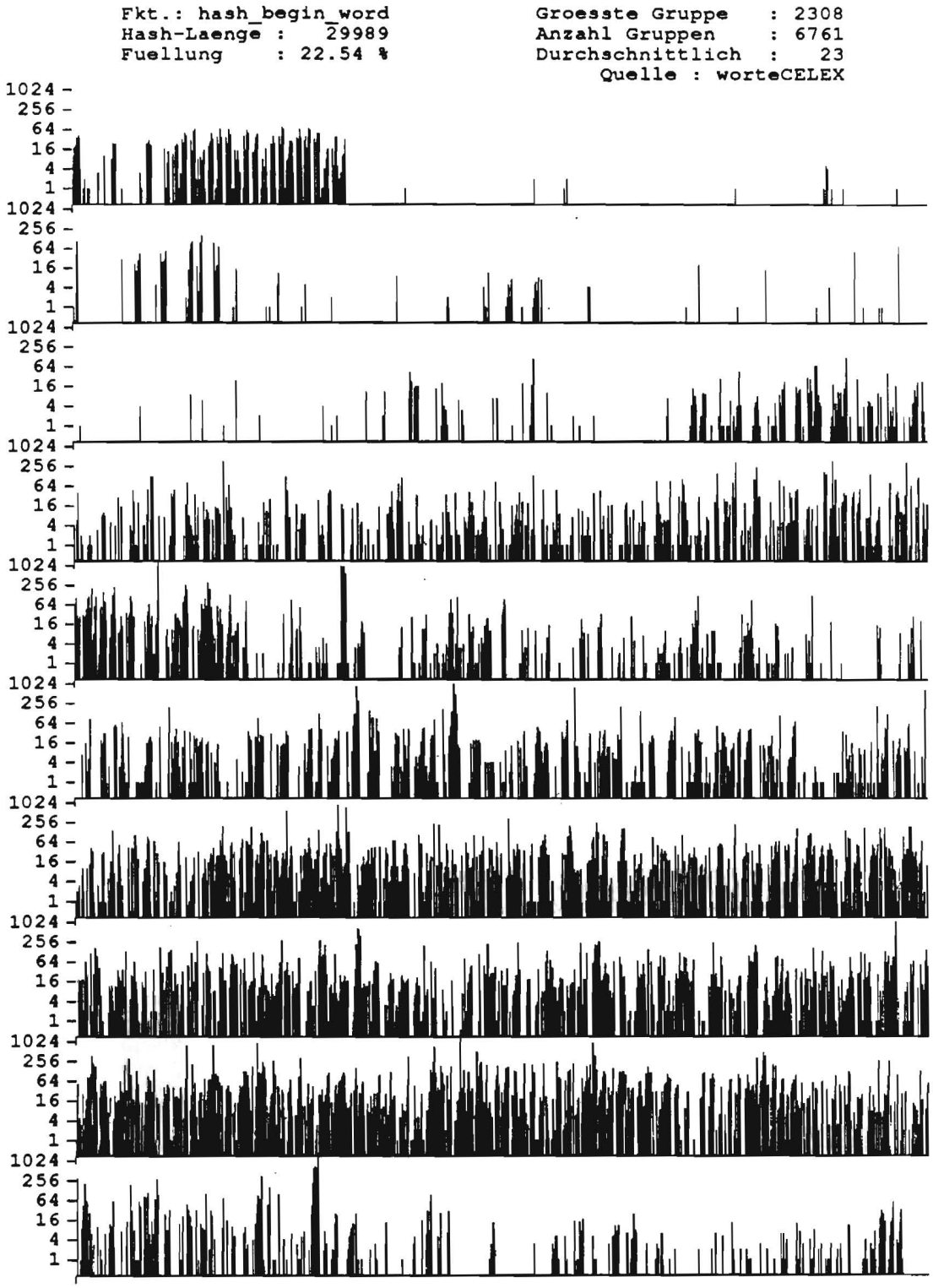

Abbildung 3.7: Verteilung der Wörter auf den Bildbereich von h_{begin}

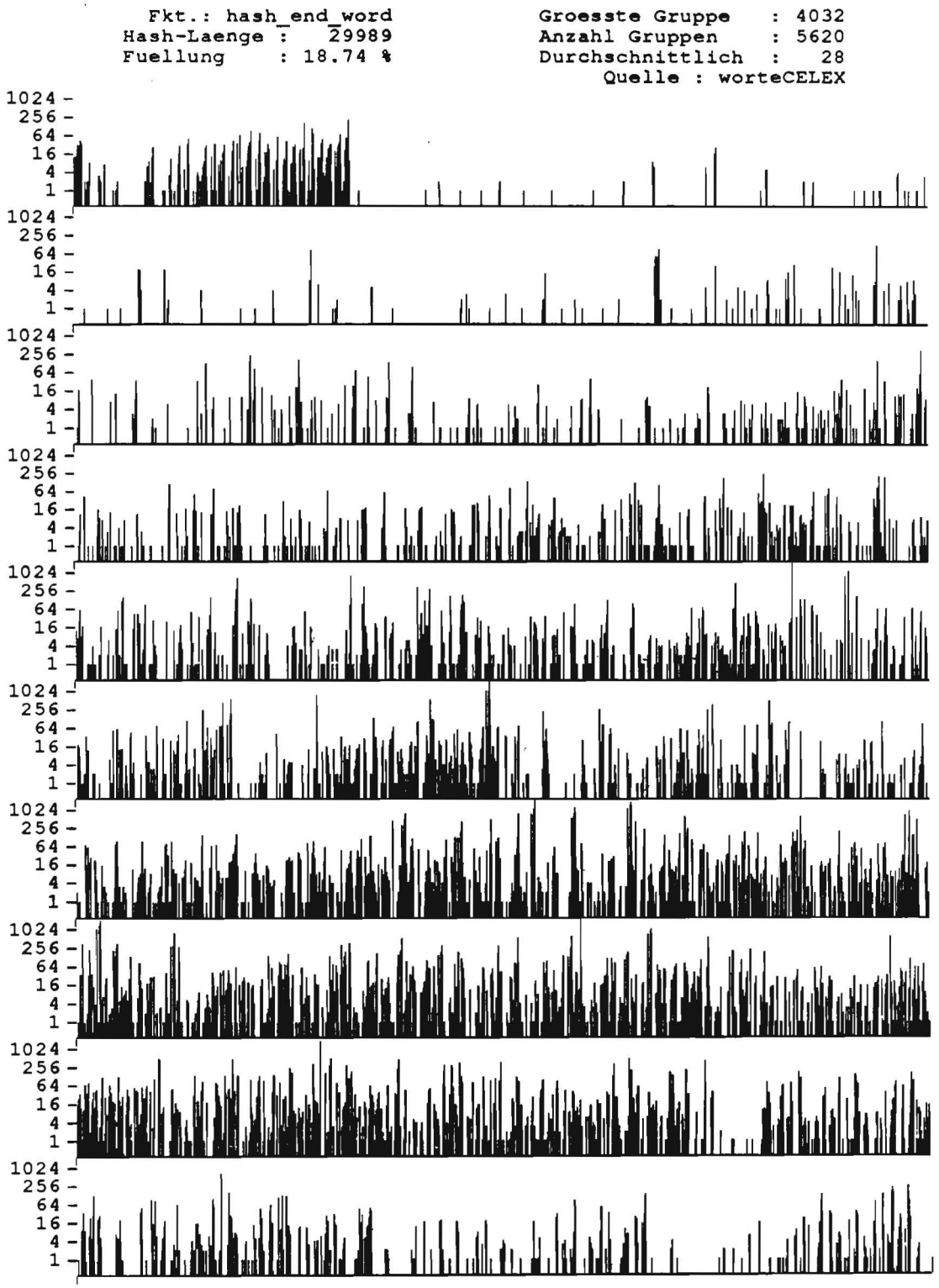

Abbildung 3.8: Verteilung der Wörter auf den Bildbereich von h_{end}

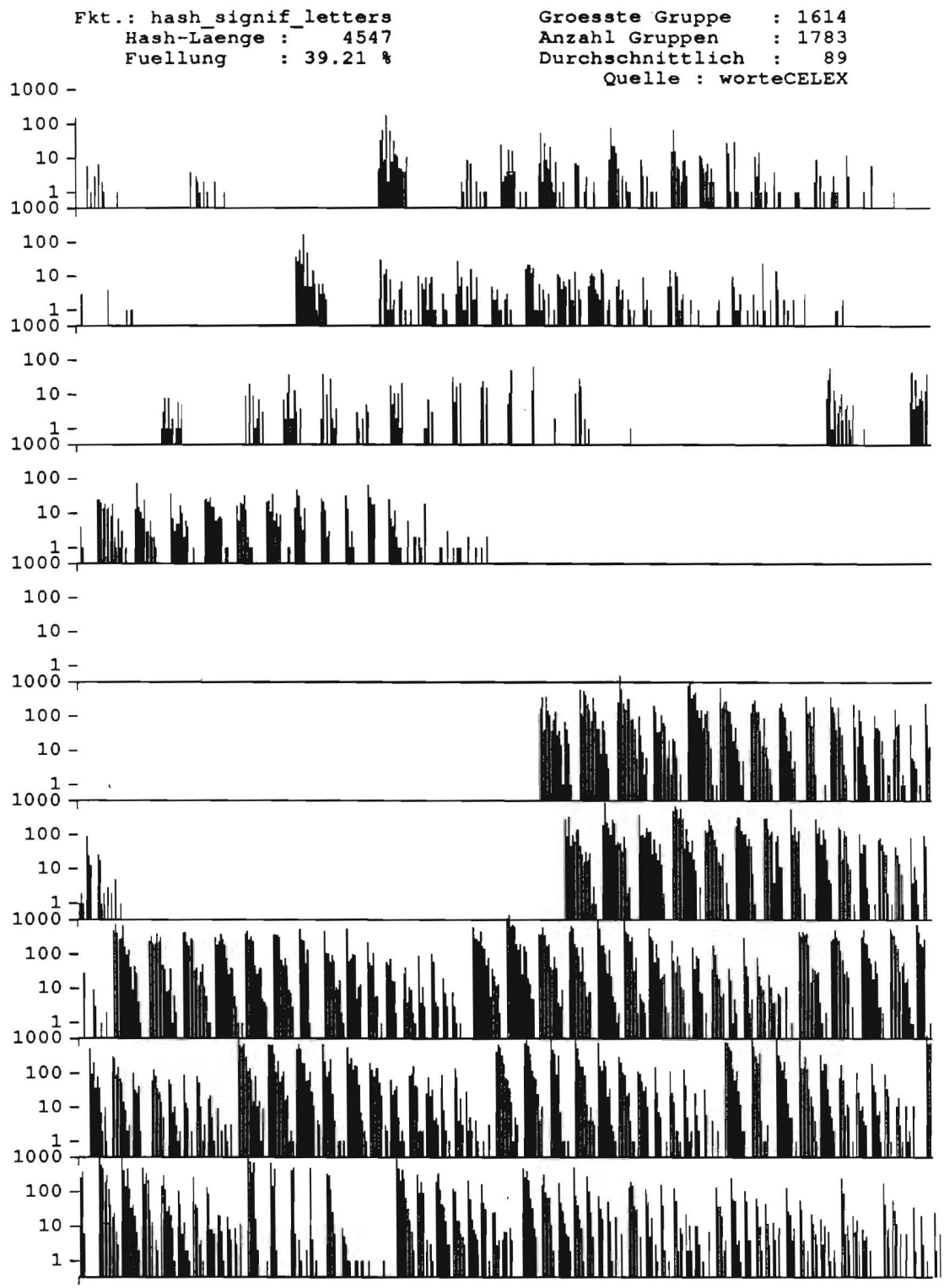

Abbildung 3.9: Verteilung der Wörter auf den Bildbereich von h_{sig}

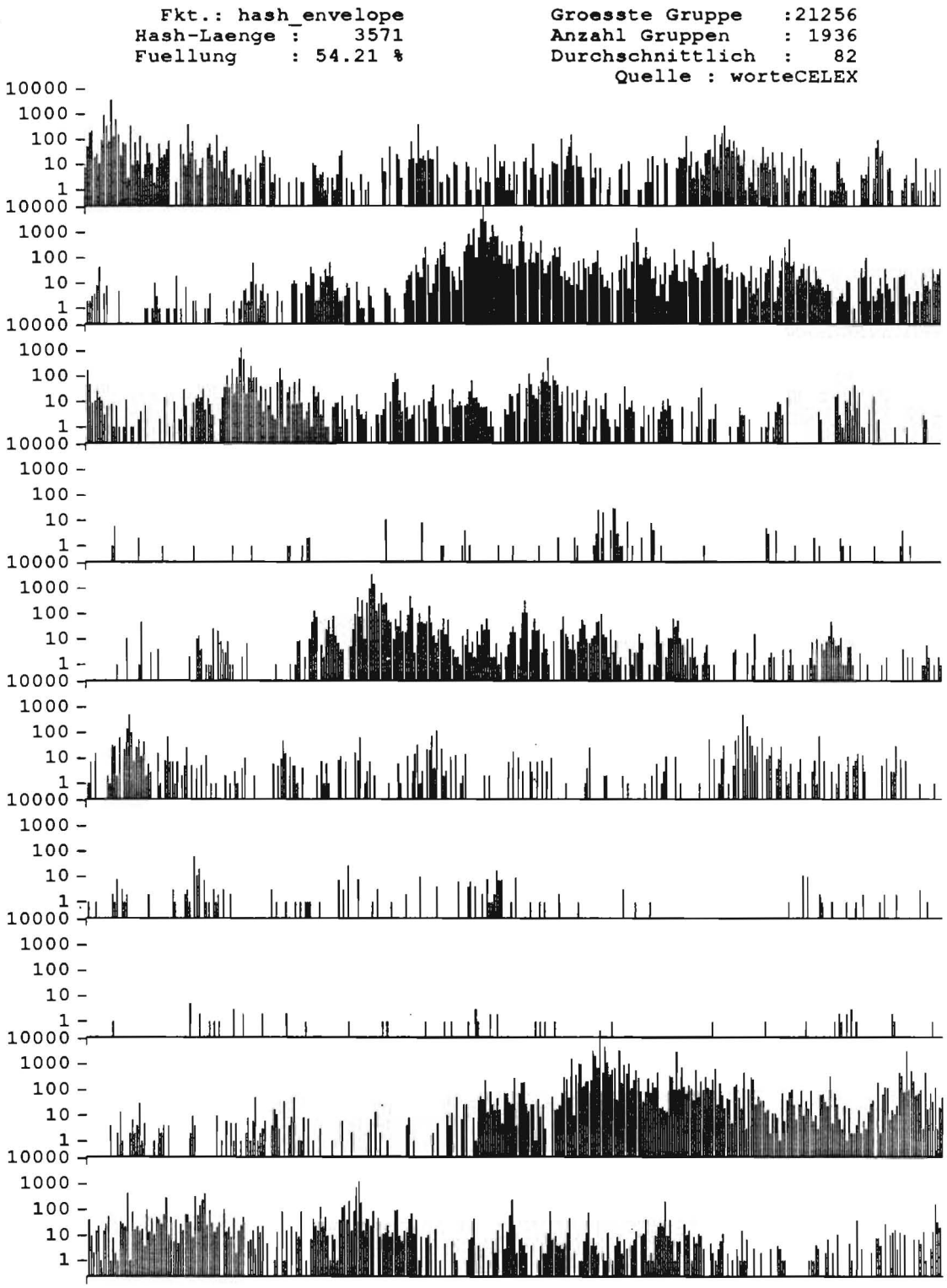

Abbildung 3.10: Verteilung der Wörter auf den Bildbereich von h_{env}

Kapitel 4

Hashing und Sichten

Wie bereits erwähnt, bildet der hashbasierte, im Sekundärspeicher liegende Teil des Lexikons den Kern unseres Systems. Wann immer ein Wort unvollständig erkannt wurde, wird er komplett nach möglichen Alternativen durchsucht, da der Benutzer alle matchenden Einträge erfahren will. Das System liefert bei vollständiger Erkennung sehr kurze Anfragezeiten.

Ganz anders sehen die Kollisionsketten aus, wenn wir uns auf **Teilinformationen** stützen. Da viele Wörter entweder gleich beginnen oder enden, werden sie auch auf dieselben Hashadressen abgebildet, und verursachen dadurch **lange Kollisionsketten**. Ein erster Versuch, die mittleren Zugriffszeiten zu optimieren, bestünde nun darin, Wörter innerhalb einer Kette in Reihenfolge fallender Zugriffshäufigkeit abzuspeichern, um so häufiger referenzierte Begriffe früher zu finden. Letztenendes müssen in diesem Fall doch die Überlaufketten vollständig durchsucht werden, da man nur so gewährleisten kann, alle matchenden Alternativen zu finden. Die Gesamtanfragezeit verkürzt sich folglich durch diese Anordnung nicht.

Eine andere Möglichkeit, zur Optimierung der Anfragezeit, ist eine Einschränkung des Suchraums und somit eine Verkürzung der Kollisionsketten. Diese Einschränkung kann einmal durch die Wahl eines möglichst kleinen Lexikons geschehen, das gerade den für die gegebene Anwendung (z.B. Adreßerkenner) benötigten Wortschatz beinhaltet. Hier wird zusätzliches Wissen über die zu erwartenden Begriffe bei der Gestaltung des Lexikons des Wortschatzes eingebracht [Srihari83].

Es ist aber auch möglich, dieses Wissen dynamisch, also zur Laufzeit, auf ein bestehendes (großes) Lexikon zu applizieren. Das heißt, daß die benötigte Zusatzinformation vom Anfragesystem bereitgestellt und vom Lexikonsystem sinnvoll genutzt werden muß.

Unser System soll das dynamische Einbringen von Wissen unterstützen. Zu diesem Zweck führen wir hier den Begriff der **Sicht** ein.

Definition: Unter einer **Sicht** versteht man die prinzipielle *Einteilung des Wortschatzes* bezüglich eines Merkmals. So stellen beispielsweise die *Kategorisierung*, *Wortlängen* oder *Zugehörigkeit zu den logischen Objekten eines Dokuments* verschiedene Sichten dar.
Unter den (Sicht-) **Ausprägungen** verstehen wir die in jeder Sicht vorkommenden unterschiedlichen Attribute wie z.B. *Substantive*, *Verben*, *Adjektive* und *Funktionswörter* in der Sicht der Kategorie oder *Adreßteil*, *Betreff*, *Anrede*, *Rumpf* in der Sicht der logischen Objekte.

Die Realisierung der Sichten hilft uns zum einen bei der Suche nach unvollständig erkannten Wörtern, indem gewissermaßen ein „Fenster" auf die potentiellen Einträge gelegt wird und kann weiterhin auch dazu genutzt werden, die unter einem *Sichtnamen* zusammengefaßten Begriffe vom Benutzer abzufragen. Letzteres kommt einer bestimmten Form semantischen Wissens gleich.

Beispiele für Sichten sind:

- **Wortkategorie** mit den Ausprägungen *Substantive, Verben, Adjektive, Funktionswörter usw.* Ein Experte der Worterkennung (Insel-Parser [Kirchmann93]) ist in der Lage, morphosyntaktische Informationen hierfür bereitzustellen. Man erkennt die Absicht, syntaktische Erwartungshaltungen (Valenzen) eines Wortes zur Suchraumeinschränkung heranzuziehen.
- **Wortlängen** mit den evtl. zu Gruppen zusammengefaßten Ausprägungen der Buchstabenanzahl wie *L1-2, L3, L4, ..., L12-99.* Auch diese Sicht ist von praktischen Nutzen, da die Wortlänge sehr genau, wenn nicht sogar exakt bestimmt werden kann.
- **Logische Sichten**, die das Vorkommen eines Wortes in einem der logischen Bestandteile des Dokumentes kennzeichnen. Die Ausprägungen wären hier z.B. *Adreßteil, Anrede, Datum, Rumpf, Schluß, etc.* Diese Sicht ist besonders interessant, da das anfragende System durch das *Logical Labeling* weiß, in welchem Dokumentbereich es sich befindet und das benötigte Wissen in der Tat zur Verfügung steht.
- Eine Einteilung bzgl. gewisser Features wie z.B. den **Word-Envelope**, der im Kapitel 3.1 als mögliche Hashfunktion vorgestellt wurde [6]. Hier wären die unterschiedlichen Hashcodes die zugehörigen Ausprägungen, die unter Umständen zu Gruppen zusammengefaßt werden könnten.

Die Entwicklung unseres Lexikons umfaßt u.a. die Realisierung solcher Sichten. Dabei stehen uns zwei Konzepte zur Verfügung, die im folgenden Abschnitt vorgestellt werden.

[6] Eine Hashfunktion unterteilt ihrerseits den Wortschatz in viele Teile und stellt somit selbst schon eine Sicht dar. Diese Problematik wird in Abschnitt 4.5 genauer betrachtet.

4.1 Statische Sichten

Es ist denkbar, daß das anfragende System zusätzlich zur Zeichenkette, die das unvollständig erkannte Wort in Form eines regulären Ausdruckes darstellt, zusätzliches Wissen in irgend einer Form mitliefern kann. Falls es gelingt, dieses Wissen zur Suche auszunutzen, kann die Anfragedauer wiederum reduziert werden. Sucht man z.B. gezielt nach einem Verb und bietet das Lexikon die Möglichkeit, die potentiellen Kandidaten auf Verben einzuschränken, so kann die Anfragedauer erheblich verkürzt werden. Wir sprechen bei derartigen a-priori Einschränkungen des Suchraums von **statischen Sichten.**

Ganz offenbar handelt es sich bei einer Sicht um eine Teilmenge unseres Gesamtwortschatzes. Es wäre denkbar, den Wortschatz in unterschiedlichen Sublexika zu verwalten, die je eine der Teilmengen beinhalten. Doch wie soll man konkurrierende Sichten behandeln? Man kann sich mehrere sinnvolle Einteilungen bzw. Partitionierungen des Wortschatzes vorstellen. Beispiele hierfür sind neben der Wortkategorie (Substantive, Verben, Adjektive, Funktionswörter) noch die Zugehörigkeiten der Wörter zu den einzelnen logischen Objekten eines Briefes (Namen, Straßen, Orte, Betreff, Anrede, Rumpf usw.) oder Aufspaltungen nach der Anzahl der Buchstaben (als Zusatzinformation seitens der Texterkennung denkbar).

Diesem Problem kann man auf mehrere Weisen begegnen:

1. **Mehrere Kopien des Gesamtwortschatzes** könnten jeweils bezüglich einer anderen Sicht aufgeteilt werden. Das Resultat wären Dateien mit z.B. nur Verben oder nur Substantiven bzw. in einer anderen Kopie Dateien mit Wörtern gleicher Länge. Je nach vorliegender Information verzweigt man in eine bestimmte Datei einer ausgewählten Kopie des Wortschatzes (siehe Abbildung 4.1).

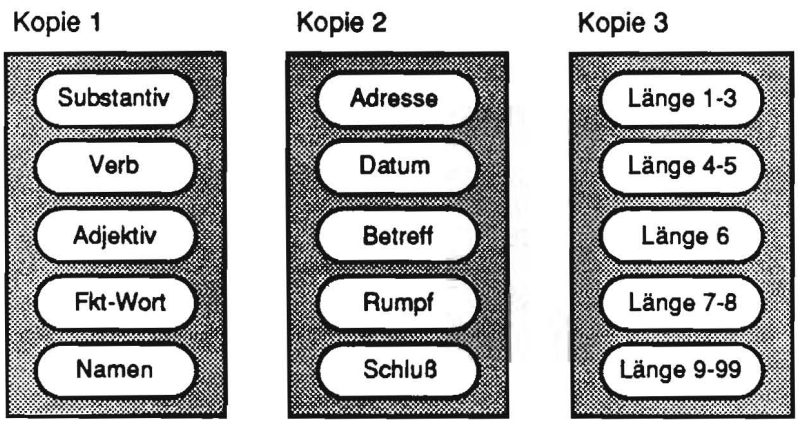

Abbildung 4.1: Physikalische Partitionierung in mehreren Kopien

Dieses Verfahren bewirkt jedoch enorme Mengen redundanter Daten, was nicht nur den Platzbedarf drastisch erhöht, sondern auch die Wartbarkeit erschwert, da eine Modifikation am Wortschatz immer gleich in jedem Teil des Wörterbuches vollzogen werden müßte.

2. Einteilung des Wortschatzes gemäß **mehrerer Sichten gleichzeitig**. Man erhält so das **Kreuzprodukt** der Ausprägungen aller Sichten als Gliederungsvorschrift, wodurch viele relativ kleine Sublexika entstünden. Letztere wären mit Wörtern gefüllt, die hinsichtlich unserer Sichten ununterscheidbar sind (siehe Abbildung 4.2).

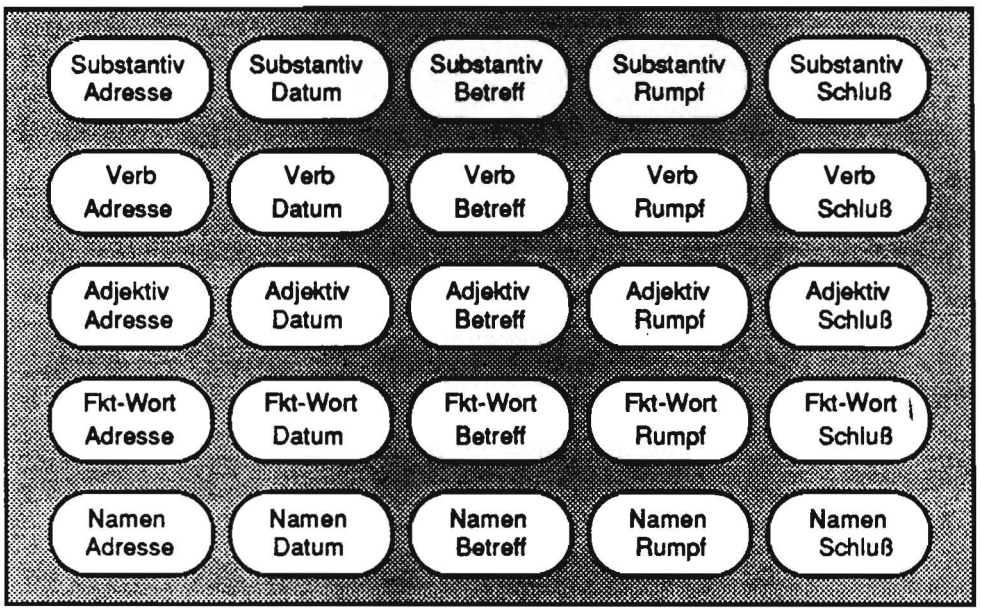

Abbildung 4.2: Partitionierung gemäß mehrerer Sichten gleichzeitig

Doch bei einer Suche ohne zusätzliche Sichtinformation erweist sich diese Vorgehensweise als nachteilhaft, da sämtliche Sublexika durchsucht werden müssen. Um eine falsche Worthypothese zurückzuweisen, müßte man alle Teile nach dem entsprechenden Wort durchsuchen und mindestens ebensoviele Dateizugriffe ausführen, wie Sublexika existieren. Somit kann auch diese Art der Sichten-Realisierung nicht unseren Anforderungen genügen.

3. Die beiden bisher angesprochenen Verfahren bewirken eine Aufsplittung der Daten in verschiedene Dateien. Wir sprechen hier auch von **physikalischer Partitionierung**. Demgegenüber stellt diese dritte Variante eine **virtuelle Partitionierung** dar, bei der der gesamte Wortschatz in einer **komplexen Struktur** untergebracht ist. Hier existieren unterschiedliche Verweiszeiger mit deren Hilfe die Wörter einer Sicht entsprechend ihrer Sicht-Ausprägung miteinander verbunden sind. Sucht man ein Wort mit ungenauer Information, so kann man je nach vorliegender Zusatzinformation entweder die normale Kollisionskette, die alle Wörter mit entsprechendem Hashcode verbindet, oder die einer ausgewählten Sicht verwenden.

   Abbildung 4.3 zeigt eine auf herkömmliche Weise verkettete Liste, bei der jedes Element konsultiert werden muß.
   Im Vergleich dazu erkennt man in Abbildung 4.4 zwei zusätzliche Verweiszeiger pro Eintrag, die jeweils nur eine Teilmenge der Elemente der normalen Kette miteinander verbinden. Diese Teilmengen sollen zwei Sichten darstellen.

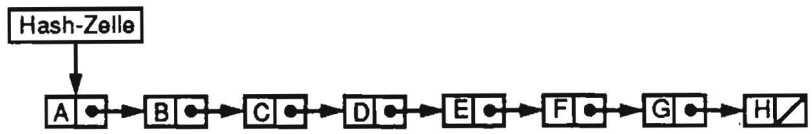

Abbildung 4.3: Kollisionskette ohne statische Sicht

Sucht man im Beispiel der Abbildung nach Objekt 'H', so mußsen bei normaler Verkettung acht lesende Zugriffe auf die Datenstruktur vorgenommen werden. Weiß das System aber, daß sich das Objekt in der Sicht 2 befindet, so kommt es über die entsprechende Sichtverkettung mit nur fünf Zugriffen aus. Da die Datenstruktur auf Sekundärspeicher organisiert ist, macht sich dieser Vorteil deutlich bemerkbar.

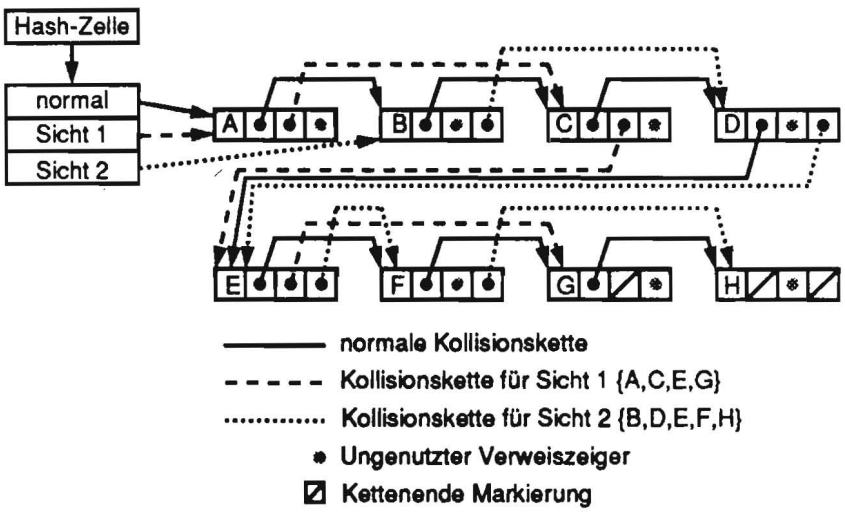

Abbildung 4.4: Kollisionskette mit zwei statischen Sichten

4.1.1 Virtuelle Partitionierung

Wir haben uns bei der Realisierung der Sichten für die dritte Lösung entschieden. Dabei kann es jedoch zu folgender Situation kommen: sucht man ein Wort und weiß, daß es sich dabei z.B. um ein Verb handelt, so möchte man die Verweiskette der Sicht Wortkategorie durchlaufen. Doch woher weiß man, daß die Verweiskette gerade alle Verben besucht und nicht etwa die Substantive? Und wieso ist es möglich, daß eine Kette sowohl für Substantive also auch für Verben oder Adjektive verwendet werden kann? Diese Fragen wollen wir im folgenden Abschnitt etwas genauer betrachten.

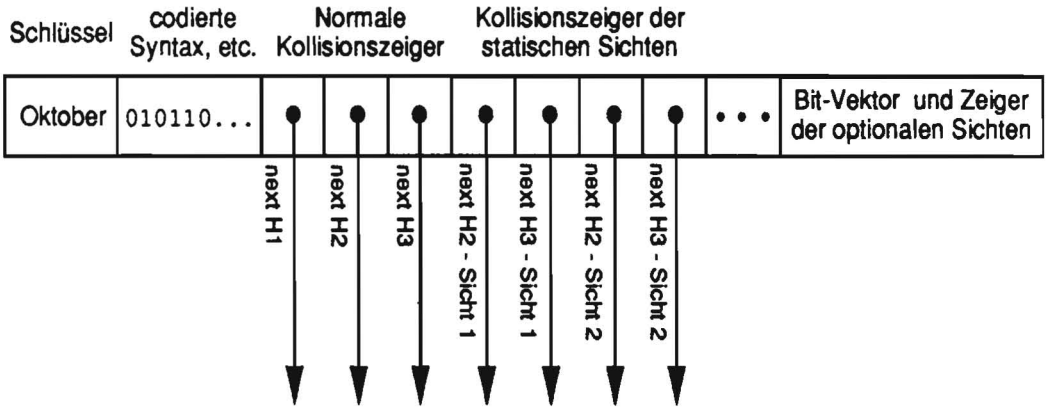

Abbildung 4.5: Mehrfachverzeigerung der Kollisionsketten

Abbildung 4.5 zeigt den Ausschnitt der Mikrostruktur, der die Kollisionszeiger der einzelnen statischen Sichten und Hashfunktionen und die der normalen Verweisketten trägt.

Mehrfache Verwendung von Kollisionszeigern

Zur Beantwortung der zweiten Frage muß man sagen, daß die Einteilung der Sichten nach Wörtern unterschiedlicher Ausprägung nur dann durch dieses Konzept befriedigt werden kann, wenn die Wortmengen der einzelnen Ausprägungen zueinander disjunkt (elementfremd) sind. Dies ist bei der Wortkategorie oder auch bei der Einteilung nach Wortlängen der Fall. Problematischer verhält sich die Sicht der logischen Objekte. Letztere unterteilt den Wortschatz in nicht disjunkte Mengen. Auf diese Problematik kommen wir in Abschnitt 4.1.2 zurück.

Wir wollen die Bedingung der Disjunktheit der Ausprägungen einer Sicht noch einmal verdeutlichen. In Abbildung 4.4 sieht man pro Eintrag zwei zusätzliche Verweiszeiger, von denen nur diejenigen tatsächlich genutzt werden, die an Objekte dieser Sicht(-ausprägung) [7] gebunden sind. So sind dort z.B. die Zeiger der Sicht 1 bei den Objekten 'B', 'D', 'F' und 'H' ungenutzt. Gäbe es nun eine Menge, die genau diese Elemente (oder eine

[7] Hier wird eine gewisse Begriffsverschiebung deutlich. Zunächst sprechen wir von einer Sicht – und plötzlich wieder von zwei Ausprägungen einer Sicht. Das kommt daher, daß wir ursprünglich im Sinne unserer Terminologie von einer (und zwar der einzigen) Ausprägung einer Sicht hätten sprechen müssen, zu der sich eine zweite Ausprägung gesellt.

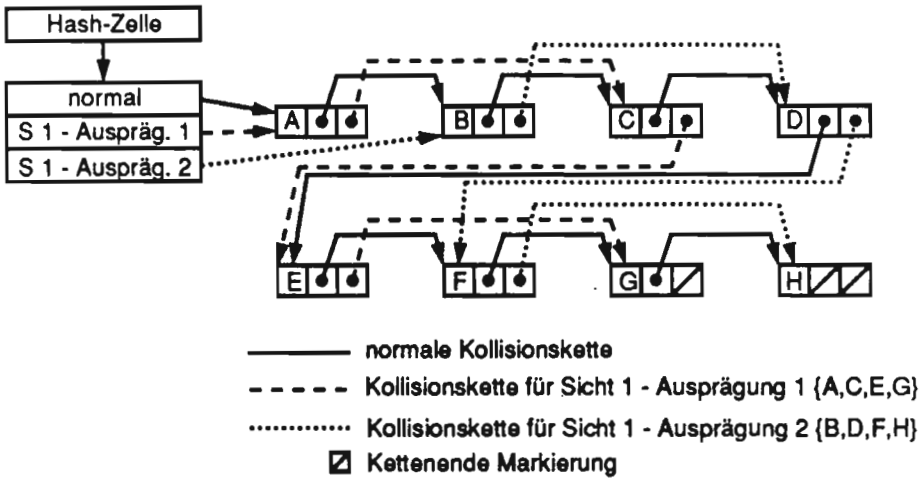

Abbildung 4.6: Kollisionskette einer statischen Sicht mit zwei Ausprägungen

Teilmenge davon) beinhaltet, so könnte man die bisher ungenutzten Zeiger für diese (neue) Sichtausprägung nutzen.

Abbildung 4.6 zeigt die beiden Ausprägungen, die durch die disjunkten Objektmengen $\{A, C, E, G\}$ und $\{B, D, F, H\}$ repräsentiert werden. Das Fehlen ungenutzter Verweiszeiger verbildlicht den effizienteren Umgang mit Speicherplatz.

Einstiegspunkte der Ausprägungen

Die erste Frage läßt sich anhand der Abbildungen 4.6 und 4.7 erklären.

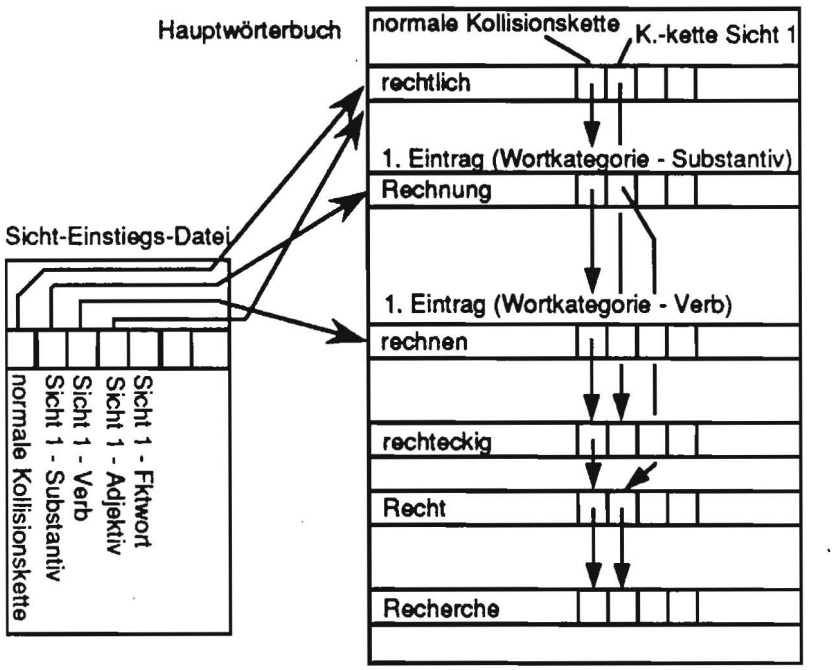

Abbildung 4.7: Verweise auf Sichtausprägungen

Zu jeder Ausprägung gibt es einen Verweis auf das jeweils erste Wort einer Kette. Dazu wird über den errechneten Hashcode (für Hashfunktion 2 oder 3) auf eine Hashzelle zugegriffen. Dort stehen die Adressen der Sichteinstiegsdatei, an der wiederum die Einstiegspunkte für alle Ausprägungen zu finden sind. An der so erhaltenen Stelle im virtuell partitionierten Wörterbuch werden dann die eigentlichen Daten, zu denen auch der Verweiszeiger der gewählten Sicht auf das nächste Wort der gleichen Ausprägung gehört, gelesen.

Wir erhalten somit die in Abbildung 4.8 dargestellte Struktur, die bei unvollständig erkannten Wörtern über mehrere Stufen hinweg auf die Daten im Lexikon zugreift und dabei auf bestimmte Sichten verzweigen kann.

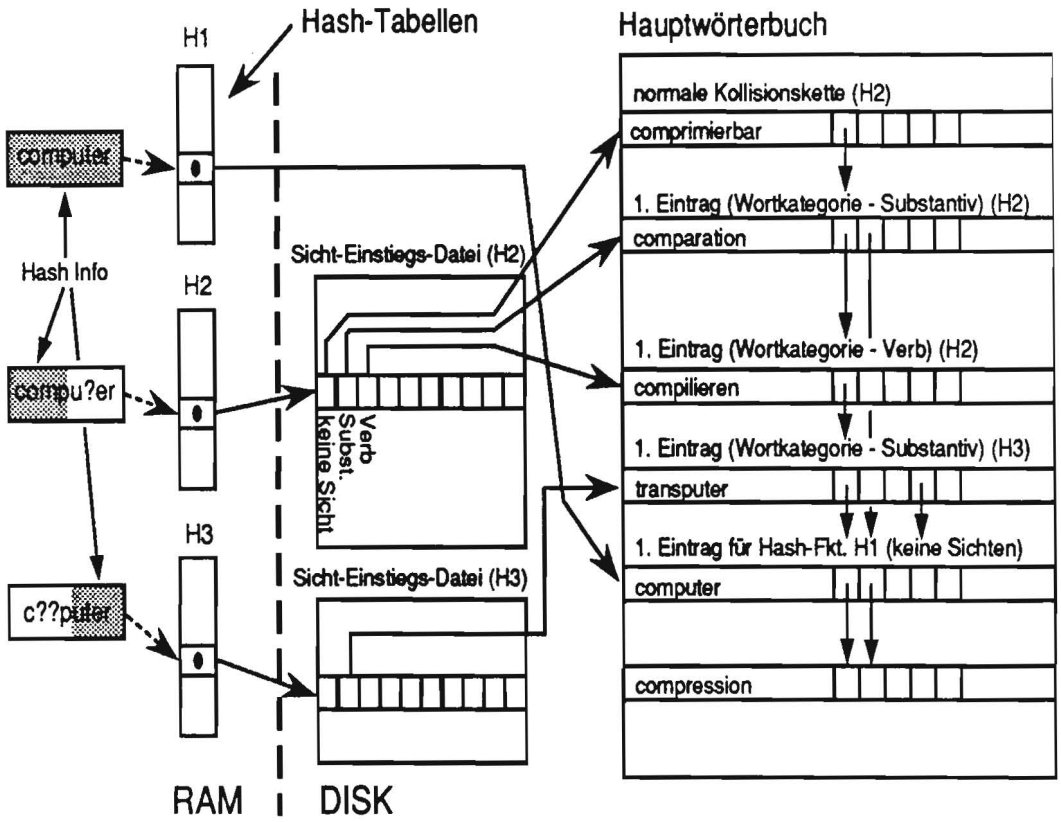

Abbildung 4.8: Architektur der statischen Sichten

4.1.2 Sichten nicht disjunkter Ausprägungen

Wie bereits erwähnt, zerfällt die Sicht der logischen Objekte durch ihre Ausprägungen nicht in Mengen disjunkter Wörter. Zur Erinnerung: unter den logischen Objekten verstehen wir Bereiche wie z.B. Adreßteil, Datum, Betreff, Anrede oder Rumpf eines Briefes. Daß die Wörter dieser Bereiche nicht elementfremd sind, wird sofort klar. Vielmehr ist eine Art Teilmengenbeziehung zu erkennen, in der der Rumpf die allumfassende Obermenge darstellt. Der Adreßteil stellt beispielsweise eine Teilmenge des Rumpfes dar. Durch feinere Untergliederung erhält man Namen, Straßen und Orte als Teile der Adresse und Vor- bzw. Nachnamen als Teile der Namen. Aus diesen Teilmengenbeziehungen kann man eine hierarchische Anordnung der Ausprägungen ableiten (siehe Abbildung 4.9).

Im Falle nicht vorhandener Disjunktheit benötigt jede Ausprägung (gleichsam Sichten) einen eigenen Kollisionszeiger (ein Wort kann schließlich in mehreren Ketten liegen). Streng genommen liegen hier im Sinne der Implementierung mehrere Sichten mit je einer Ausprägung vor, jedoch nehmen diese „Sichten“ für uns Menschen aufgrund ihrer gemeinsamen semantischen Ableitung eher die Position von Ausprägungen einer Sicht ein. In der Anwendung ist diese Begriffsverschiebung jedoch nicht mehr spürbar, da die Implementierung diesen Unterschied für den Benutzer – abgesehen vom Design des Profiles für die statischen Sichten – transparent macht.

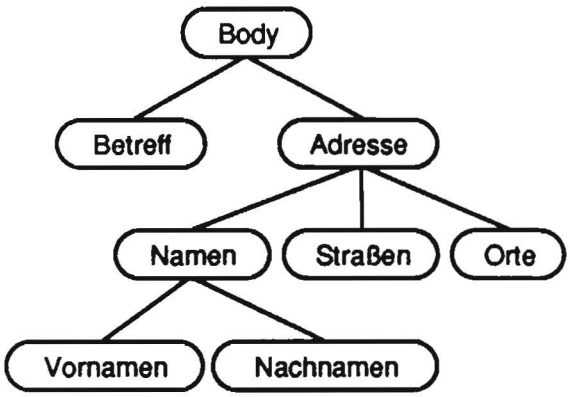

Abbildung 4.9: Hierarchie logischer Sichten

Durch die oben erwähnte Hierarchie erübrigt sich die Definition einer eigenen Sicht für jedes logische Objekt; es genügt vielmehr, dies für die „Blätter“ des Hierarchiebaumes zu tun. Auf diese Weise kann man die im Sinne des Speicherplatzbedarfs relativ teueren statischen Sichten deutlich reduzieren.

Ein Zugriff über eine hierarchische Sicht (innerer Knoten) muß bei diesem Vorgehen auf mehrere Zugriffe über die darunterliegenden „elementaren“ Sichten (Blätter) abgebildet werden.

4.2 Dynamische (optionale) Sichten

Die bisher behandelten statischen Sichten können immer dann eingesetzt werden, wenn über die Schreibweise plus Sichtinformation auf das Lexikon zugegriffen wird. In diesem Punkt unterscheiden sich die optionalen Sichten. Bei ihnen wird einer grundlegenden Anforderung der Textanalyse nachgekommen, beliebige Wortmengen (in der Definitionsphase) zu einer Einheit zusammenzufassen, um sie zur Anwendungszeit über einen der Menge zugeordneten (Sicht-) Namen vom Lexikon anfordern zu können. Die zugrundeliegende Intention der Textanalyse ist, im Text gezielt nach Wörtern einer Sicht zu suchen. Dabei ist anzunehmen, daß es sich um bedeutungstragende Begriffe handelt, die in einer solchen Sicht zusammengefaßt sind. So können beispielsweise Wörter, die typisch für eine Rechnung sind (in einer Sicht namens „Rechnung" liegend), dafür herangezogen werden, Wahrscheinlichkeiten für das Vorliegen eines Rechnungsschreibens zu evaluieren [Dittrich92].

Es ist möglich, daß einige Wörter sehr vielen optionalen Sichten zugehören, wohingegen andere keiner einzigen Sicht zugeordnet sind. Folglich wäre es in bezug auf den Plattenplatzbedarfs sehr teuer, für jede optionale Sicht bei allen Einträgen einen Verweiszeiger zu reservieren. Daher haben wir uns entschieden, nur dann einen Zeiger bereit zu stellen, wenn ein Wort auch tatsächlich in einer Sicht enthalten ist.

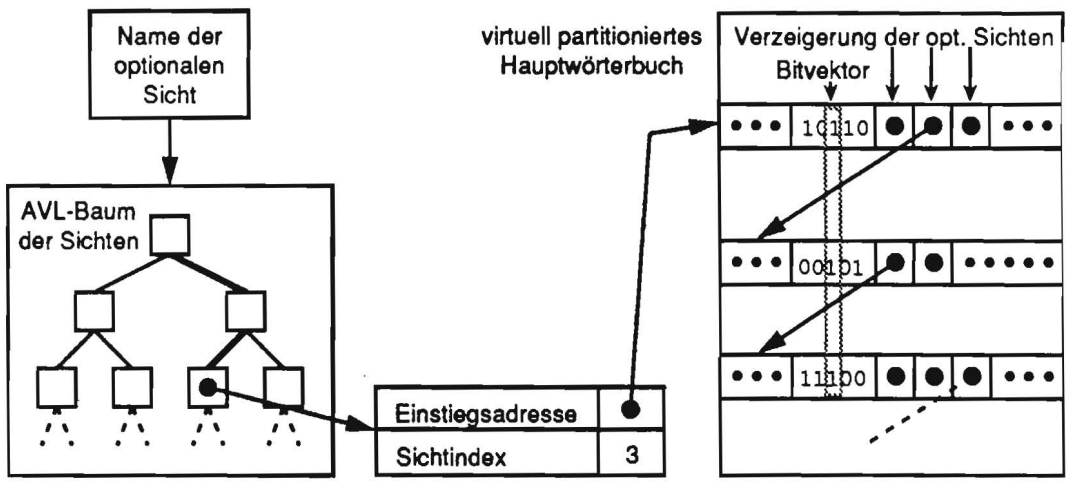

Abbildung 4.10: Architektur der optionalen Sichten

Deshalb kennzeichnet ein Bitvektor in jedem Eintrag die Zugehörigkeit zu den optionalen Sichten. Die Anzahl der gesetzten Bits ist folglich gleich der Anzahl der Verweise für optionale Sichten. Dieser Bitvektor dient zusätzlich der Ermittlung des richtigen Verweiszeigers einer Sicht. Ist beispielsweise das Bit für eine Sicht „X" das dritte gesetzte Bit im gesamten Vektor, so ist auch der dritte Verweiszeiger für Sicht „X" reserviert (siehe Abbildung 4.10).

Die Existenz eines Pointers ist also optional im Gegensatz zu den statischen Sichten, wo jeder Eintrag mit dem Raum für jede Sicht ausgestattet ist. Diesen Unterschied haben wir

zur Benennung der beiden unterschiedlichen Sicht-Konzepte herangezogen und sprechen somit von *statischen* bzw. *dynamischen* oder *optionalen Sichten.*

4.3 Suchraumeingrenzung durch Filter

Der für die optionalen Sichten bereitgestellte Bitvektor kann neben der Ermittlung des Nachfolgezeigers natürlich auch dafür genutzt werden, die Zugehörigkeit eines Eintrages zu einer bestimmten Sicht zu evaluieren. Das so vorhandene Wissen läßt sich weiter nutzen, indem man bei einer beliebigen Anfrage (sowohl über die Lesart als auch über eine dynamische Sicht) nur diejenigen Einträge zurückliefert, die durch ein Prädikat als einer bestimmten Sicht angehörend ausgewiesen werden.

Zwei (oder mehrere) verschiedene Prädikate lassen sich auch miteinander verknüpfen. Wir wollen diese Verknüpfung von Prädikaten im folgenden *Filter* oder *Filtersystem* nennen.

Das Lexikon stellt ein solches Filtersystem, bestehend aus mehreren Einzelfiltern, zur Verfügung. Es beruht auf den optionalen Sichten und „arbeitet" mit den entsprechenden Bitvektoren.

Die Funktionsweise und unterschiedlichen Verknüpfungen von Filtern werden in Abschnitt 6.3 detailliert betrachtet.

4.4 Mikrostruktur des Lexikons

Das virtuell partitionierte Wörterbuch hat die in Abbildung 4.11 dargestellte Mikrostruktur.

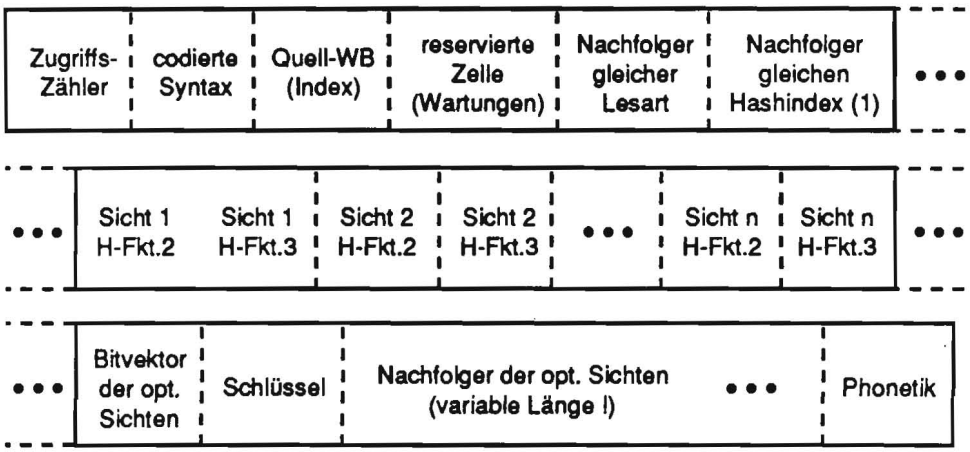

Abbildung 4.11: Mikrostruktur des virtuell partitionierten Lexikons

Die einzelnen Zellen haben folgende Bedeutung:

- Die erste Zelle beinhaltet einen 4 Byte großen *Zugriffszähler*, der die Anzahl der Zugriffe auf ein Wort protokolliert. Die gegenwärtige Implementierung sieht eine optionale Initialisierung dieses Wertes durch die Frequenzangabe in den Quellen vor. Dieser Wert kann dazu genutzt werden, hochfrequente Wörter (Schwellwert) in ein speziell dafür vorgesehenes, speicherresidentes Lexikon (z.B. Trie) zu laden.

  Problematisch wird die dynamische Inkrementierung dieses Zählers, da dies einer Bestätigung des Benutzers bedarf, sobald ein gesuchtes Wort gefunden wurde.

  Um ein *kompaktes Hauptwörterbuch* zu erhalten, bietet das Lexikonsystem eine bedingte Übersetzung [Kernighan78] an, bei der diese Zelle entfällt.

- Die *Syntax* wird in 4 weiteren Bytes codiert. Dies ist in den Abschnitten 5.4 und B.1.1 genau erklärt.

- Der nächste Eintrag ist ein *Verweis auf das Quellwörterbuch*, aus dem dieser Eintrag stammt. Diese Information kann bei Wartungsarbeiten an den Quellen interessant sein, da man so die Herkunft eines Eintrags bezüglich der Quellen ermitteln, und ihn anschließend ändern oder löschen kann. Im Falle der kompakten Version entfällt auch diese Zelle.

- Eine 8 Byte große ungenutzte Zelle kann bei Wartungen oder für kurzfristige Erweiterungen verwendet werden. Ihr genauer Gebrauch steht nicht fest – sie wird bei der erwähnten kompakten Version ebenfalls aus der Struktur entfernt werden.

- Der *Nachfolger für Einträge gleicher Lesart* zeigt auf gleichgeschriebene Wörter mit anderer Syntax (z.B. können alle Farben Substantiv und Adjektiv sein). Dadurch kann das Durchlaufen der Kollisionskette bei vollständig erkannten Wörtern nach Auffinden des Eintrages abgebrochen werden, es sei denn das Lexikon enthält noch ein weiteres Wort gleicher Lesart, welches über diesen Zeiger in einem weiteren Zugriff gefunden wird. Auch dieser Eintrag kann aus dem System eliminiert werden, da diese Verweise nur bei den ohnehin sehr schnellen Anfragen nach vollständig erkannten Wörtern wirksam werden.

- Der folgende Verweis ist für die normale *Kollisionskette der Hashfunktion 1.*

- Danach kommen sämtliche *Nachfolgezeiger der Hashfunktionen 2 und 3* für alle Sichten. Hierbei fällt auf, daß keine Zeiger für „normale" Anfragen ohne Sichtinformation vorhanden sind. Dies ist aber nur scheinbar der Fall, da auf Implementierungsebene Sicht 1 diese Aufgabe übernimmt. Dazu werden bei der Generierung alle Einträge automatisch in diese Sicht mit nur einer "Pseudo"-Ausprägung eingefügt. Wird bei der Suche keine andere Ausprägung spezifiziert, so wird diese "Pseudo"-Ausprägung als vorgegeben angenommen.

- Als nächster Eintrag folgt der *Bitvektor* für die optionalen Sichten und das Filtersystem.

- Ab der nächsten Zelle, dem *Schlüssel* für die Anfragen über die Lesart, kann die Länge der Zellen variieren, was schon durch die Möglichkeit unterschiedlich langer Wörter deutlich wird.

- Im Anschluß an den Schlüssel folgen die *Nachfolgezeiger der optionalen Sichten.* Ihre Anzahl ist gleich der Anzahl gesetzter Bits im Bitvektor.

- Schließlich exisitiert noch Raum für die optionale Speicherung der *phonetischen Information* eines Wortes.

4.5 Hashfunktionen und Sichten – zwei Wege ein Ziel

Bei genauer Betrachtung einer Hashfunktion handelt es sich dabei ebenso um eine Einteilung des Wortschatzes in Teilmengen, (die sogar disjunkt zueinander sind,) wie es bei den Sichten der Fall ist.

Worin liegen nun aber die Unterschiede? Warum ist eine Hashfunktion als Sicht nicht geeignet? Was bewirkt eine Sicht als primäres Selektionskriterium? Diese Fragen sollen im folgenden diskutiert werden:

1. Wichtigster Unterschied ist das Entscheidungsverfahren. Die Zugehörigkeit eines Wortes zu einer Sicht kann nicht immer algorithmisch bestimmt werden, da die Einteilung in Sichten, wie die *Wortkategorie* oder die der *logischen Objekte*, auf höheren, semantischen Eigenschaften beruhen und nicht wie bei den Hashfunktionen aus den Buchstabenfolgen *berechnet* werden können.

   Die Sichtzugehörigkeit kann im allgemeinen erst dann ermittelt werden, wenn die Lesart bereits bekannt ist, namlich durch Konsultieren wortspezifischer Information — zu einem Zeitpunkt also, zu dem das Wort bereits (über andere Zugriffstechniken) gefunden wurde.

2. Der Zugriff auf ein Wort mittels der Lesart geschieht in mehreren Stufen:

   (a) Suchraumeinschränkung durch eine ausgewählte Hashfunktion.

   (b) Verkürzung der Kollisionsketten durch eine evtl. angegebene statische Sicht bei unvollständig erkannten Wörtern.

   (c) Vergleich der Kandidaten mit dem Eingabestring [8].

   (d) Weitere Extraktion durch das Filtersystem.

   Dabei wird das Hashing als erstes verwandt, da es eine sehr starke Einschränkung des Wortschatzes ermöglicht und so alle nachgeschalteten Schritte auf geringen Datenmengen arbeiten. Außerdem ist die Berechnung der Hashadresse sehr schnell.

   Würde unsere Hashfunktion den Wortschatz in nur verhältnismäßig wenige Teilbereiche zerlegen, so wäre die primäre Eingrenzung nicht sehr stark und die für die Schritte 2 bis 4 verbleibenden Daten sehr groß. Aus diesem Grund sollte eine Hashfunktion gewählt werden, die den Wortschatz in viele Teile zerlegt. Da Sichten im allgemeinen weit weniger Ausprägungen haben als Hashfunktionen Bildpunkte, sind sie nicht als primäres Selektionskriterium geeignet.

3. Dennoch haben wir eine Sicht als primäres Selektionskriterium realisiert. Dies entspricht jedoch dem Zugriff auf die Daten über die Benennung einer optionalen Sicht und nicht dem mittels einer Hashfunktion. So könnte eine Anfrage beispielsweise „gib mir alle Mitarbeiter“ (natürlich in geeigneter Syntax) lauten, um die Namen

[8] Hier ist in der momentanen Implementierung ein exaktes Matching auf einen regulären Ausdruck möglich. Geplante künftige Erweiterungen sollen hier ein approximatives (fehlertolerantes) Matching ermöglichen, das auf der gewichteten Levenshteindistanz basiert (siehe Abschnitt 7.2.2).

der Beschäftigten abzufragen. Erstens ist die Intention dieser Art des Zugriffes verschieden (Abfragen von Begriffen, die in einer Sicht liegen) und zweitens ist hier kein Matching nachgeschaltet.

4. Eine Hashfunktion als „Quasi" Sicht ist aus mehreren Gründen nicht sehr sinnvoll:

   (a) die Verzweigung auf eine Sichtausprägung geschieht *nach* der Einschränkung durch eine (primäre) Hashfunktion. Nun wählt man für Schritt 1 diejenige Hashfunktion, die bei gegebener Eingabe die beste Vorauswahl verspricht. Hätte man nun eine Funktion, die in Stufe 2 (also als Sichteinteiler) eine weitere wesentliche Einschränkung ermöglicht, so ist zu überlegen, ob nicht diese Funktion, oder eine Kombination aus beiden Funktion erstere ersetzen soll.

   (b) Aufgrund der gewählten Architektur zur Realisierung statischer Sichten würden die vielen Ausprägungen einer Hashfunktion zu enorm großen Sichteinstiegsdateien führen, da deren Größe mit der Anzahl der Ausprägungen proportional wächst.

5. Da unser System auf einem dreifach-Hashing beruht, bei dem je nach Art der Eingabe eine andere Funktion gewählt wird und diese Entscheidung dem Benutzer abgenommen werden soll, muß der Controller des Laufzeitsystems in der Lage sein, selbständig eine optimale Wahl zu treffen. Entscheidungsgrundlage dieser Wahl kann nur der Eingabestring sein, folglich muß auch die Hashadresse aus der Eingabe berechnet werden, was die Verwendung einer üblichen Hashfunktion bedingt.

Als Fazit ist festzustellen, daß die Unterschiede und somit auch die Abgrenzung einer Partitionierung nach *Sicht* und *Hashfunktion* nicht in eine feste Definition gefaßt werden können. Es läßt sich kein eindeutiger Wert nennen, ab dem man von der Anzahl Ausprägunen einer Sicht oder der Anzahl Gruppen einer Hashfunktion sprechen muß. Ebenso sind Sichten möglich, die algorithmisch ermittelt werden können (z.B. Wortlängeneinteilung, die aufgrund weniger Gruppen nicht zur Primärauswahl geeignet ist).

Abschließend bleibt zu sagen, daß wir uns beim Zugriff über die Lesart eines Wortes zunächst einer Hashfunktion bedienen, um dann im zweiten Schritt auf eine evtl. verfügbare statische Sicht zu verzweigen.

4.6 Zusammenfassung

Zusammenfassend möchte ich hier die Besonderheiten des vorgestellten Konzepts noch einmal herausstellen.

(+) Geringer Arbeitsspeicherbedarf, da der Großteil der Daten auf Dateien organisiert ist.

(+) Sehr schneller Zugriff auf Wörter bei vollständiger Erkennung der Buchstaben (im Mittel weniger als zwei Leseoperationen auf Dateien).

(+) Behandlung unvollständiger Eingaben durch Zugriff über Teilinformationen und automatische Hypothesengenerierung bei extremen Unsicherheiten (hypothesize & test).

(+) Zugriff auf die Daten über unterschiedliche Hashfunktionen und automatische Auswahl der geeignetsten durch Analyse der Eingabe.

(+) A-priori Einschränkung des Suchraumes bei unvollständig erkannten Wörtern durch Ausnutzung evtl. vorliegender Sichtinformation (statische Sichten). Dadurch sind verbesserte Anfragezeiten zu erreichen.

(+) Effizienter Umgang mit Speicherressourcen, da sich alle Ausprägungen einer statischen Sicht einen Kollisionszeiger als Overhead-Information teilen (eine Ausnahme bildet die Sicht der logischen Objekte eines Briefes).

(+) Abspeicherung beliebig vieler Informationen zu den einzelnen Wörtern mit variabler Datensatzlänge.

(−) Die aufwendige Struktur macht einen Generierungsprozeß unumgänglich.

(+) Durch die Generierung bleiben die Quell-Wörterbucher modular, was eine Anpassung an andere Domänen bzw. Anwendungen vereinfacht.

(+) Definition beliebiger Wortmengen als optionale Sichten. Dabei sehr effizienter Umgang mit dem Speicherplatz.

(+) Zusätzliche Suchraumeinschränkung auf Verknüpfungen optionaler Sichten mittels des Filter-Konzepts.

(+) Mehrfache Nutzung des Bitvektors (optionale Sichten und Filter).

Kapitel 5

Generierungsprozeß

Die komplexe Struktur des virtuell unterteilten Wörterbuches macht einen Generierungsprozeß unumgänglich. Abbildung 5.1 zeigt den stark vereinfachten Ablauf der Generierung mit den Profiles und den Quell-Wörterbüchern, die dem Generator als Eingabe dienen, sowie dem virtuell partitionierten Gesamtwörterbuch mit den zughörigen Hashtabellen und Sichteinstiegsdateien als Ausgabe.

Dieses Vorgehen erinnert an einen Compilierungsvorgang, bei dem die (von Menschenhand erzeugten) Quellen eines Programms durch den Übersetzer (Compiler) in eine vom Computer ausführbare Form übergeführt wird. Kein Mensch käme auf die Idee, ein ausführbares Programm z.B. mit einem Binäreditor zu erstellen und genauso verhält es sich bei unserem Wörterbuch. Tatsächlich gibt der Generator auch Warnungen aus, wenn Informationen nicht interpretiert werden können oder die Disjunktheit der Sichtausprägungen nicht gewährleistet ist.

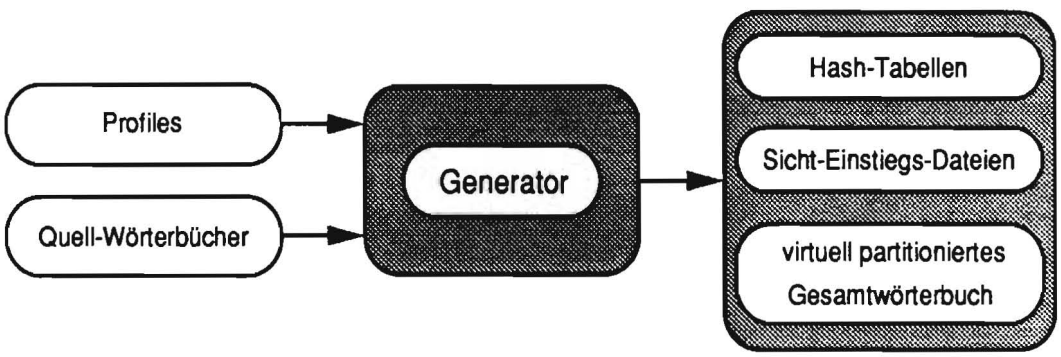

Abbildung 5.1: Der Generator und seine Ein-/ Ausgabe (vereinfacht)

Die Flußdiagramme der Abbildungen 5.2 und 5.3 zeigen den groben Ablauf des Generierungsprozesses. Dabei konkretisiert Abbildung 5.3 die Funktionalität der in Abbildung 5.2 schematisierten Box „*Quelle abarbeiten und in das virtuell partitionierte Wörterbuch einfügen*".

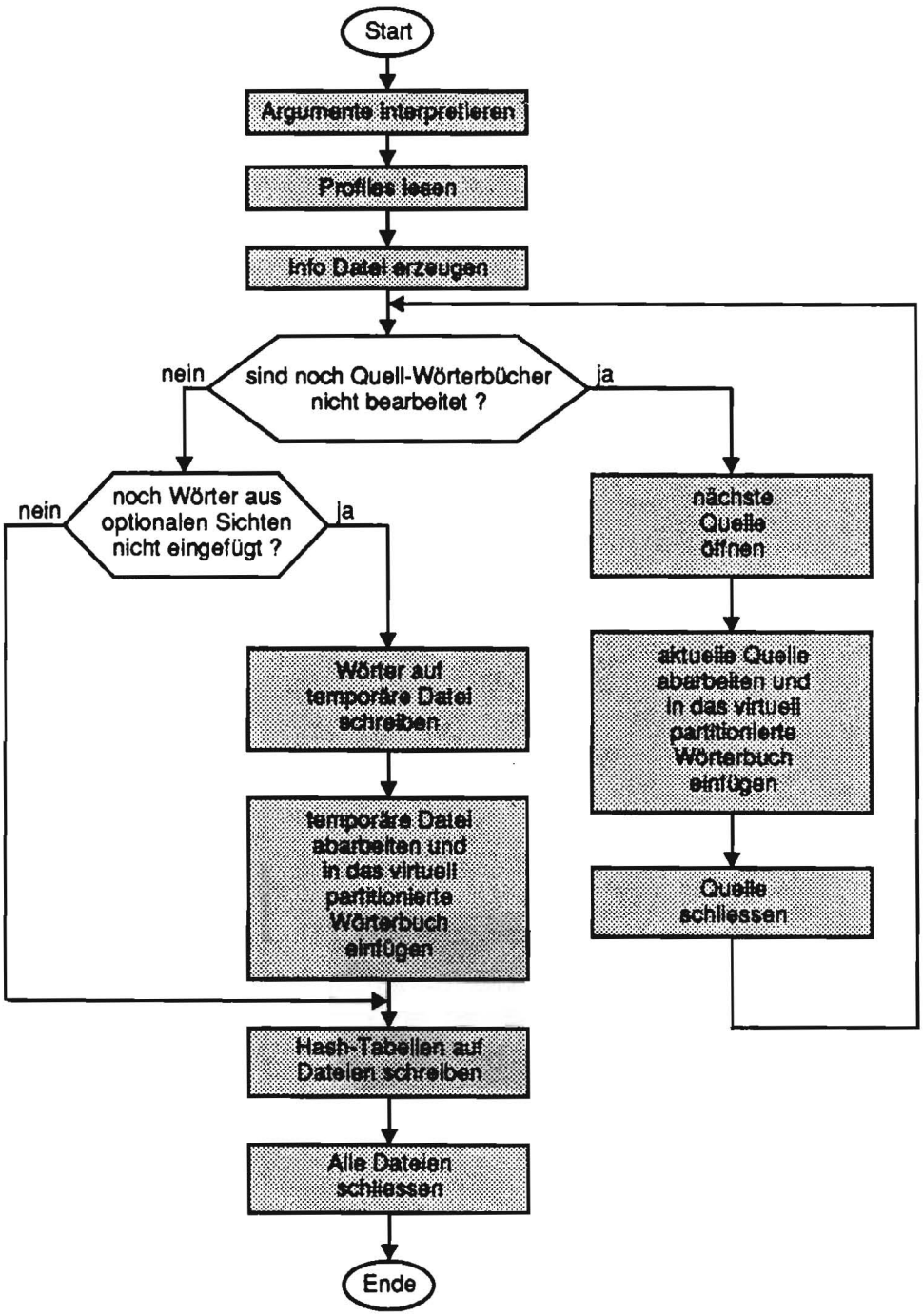

Abbildung 5.2: Flußdiagramm des Generators

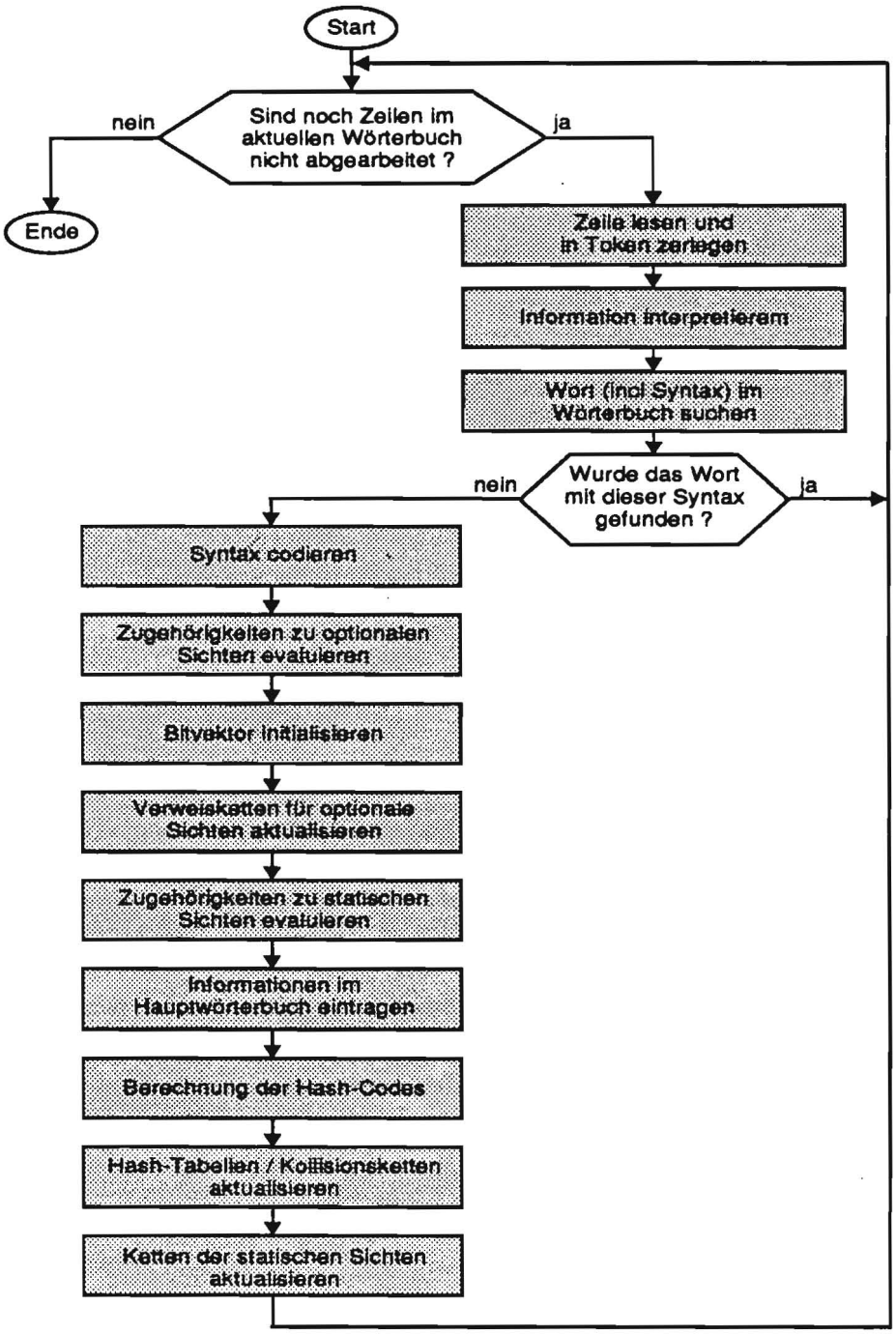

Abbildung 5.3: Flußdiagramm des Abarbeitens einer Quelle

5.1 Eingabedaten des Generators

5.1.1 Profiles

Momentan unterstützt das System vier Arten von Profiles, die wir im folgenden näher vorstellen werden:

1. Eine Datei mit Regeln zur Codierung / Decodierung der Syntax in ein 32-Bit Wort. Auf diese Weise bleibt das Lexikon flexibel. Zum Beispiel wird ein Wort im Quellwörterbuch durch das Token „SB“ oder durch „SUBST“ als Substantiv kenntlich gemacht. Ein entsprechender Eintrag in diesem Profile paßt das System daraufhin an.

2. Ein Profile zur Deklaration der statischen Sichten. Hier werden sowohl die Anzahl als auch die Ausprägungen der statischen Sichten definiert. Wie schon beim Syntax-Profile werden bestimmten Token im Quellwörterbuch jeweils eine spezielle Bedeutung zugeteilt, in diesem Falle die Zugehörigkeit zu einer bestimmten Sichtausprägung.

3. Ein weiteres Profile spezifiziert die optionalen Sichten. Diese können auf drei verschiedene Weisen angegeben werden.

   (a) Durch Auflistung von Wörtern, die einer Sicht zugehören.

   (b) Durch Angabe eines Tokens, welches ein Wort in einem Quell-Wörterbuch als einer Sicht zugehörend kennzeichnet.

   (c) Durch Angabe bereits definierter optionaler Sichten. Dies bewirkt eine Vereinigung dieser Sichten zu einer neuen Sicht. Man kann hierbei auch von einer Hierarchisierung sprechen.

   Alle drei Verfahren können miteinander verbunden werden, es besteht also keine Einschränkung, eine Sicht auf genau eine Weise zu definieren.

4. Schließlich werden durch eine spezielle Phrasendatei alle Phrasen (typische Redewendungen einer Domäne) angegeben, welche als solche referenzierbar sein sollen. Dadurch wird es möglich, zu einem bestimmten Wort diejenigen Phrasen zu erhalten, in denen dieses Wort vorkommt.

5.1.2 Quellen

Nachdem der Generator alle Profiles eingelesen hat, beginnt er mit dem Aufbau des Lexikons, indem er die Quellwörterbücher Zeile für Zeile einliest, die enthaltenen Informationen extrahiert und schließlich in das Hauptwörterbuch aufnimmt. Da der Generator mehrere Quellen zuläßt, hat der Benutzer die Möglichkeit, seinen Wortschatz modular zu organisieren. So können beispielsweise folgende Quellen zum Aufbau eines Lexikons herangezogen werden.

- Datei mit Substantiven.
- Datei mit Adjektiven.
- Datei mit Verben.
- Datei mit Funktionswörtern.
- Datei mit Mitarbeiternamen.
- Datei mit Kundennamen.
- Datei mit Straßennamen.
- Datei mit Ortsnamen.
- Datei mit Produktnamen.
- ...

Diese Modularisierung erlaubt eine sehr leichte Adaption des Systems an neue Aufgaben. So kann man sich beim Aufbau eines Lexikons für einen Adreßerkenner auf die Quellwörterbücher Mitarbeiternamen, Kundennamen, Straßennamen und Ortsnamen beschränken. Das resultierende System enthält demzufolge nur die relevanten Daten. Es besteht natürlich auch die Alternative, eine Sicht auf den Adreßteil zu definieren und den Gesamtwortschatz aufzunehmen – jedoch würde dies sehr viel größere Dateien zur Folge haben.

Um das Extrahieren der Informationen aus den Quellen schnell zu machen, werden alle potentiellen Token, die durch die Profiles mit entsprechenden Informationen versehen sind, in balancierten Suchbäumen (sogenannten AVL-Bäumen) [Knuth73] [Comer79] abgespeichert. Das System verwaltet mehrere unabhängige AVL-Bäume für die Token der Syntax und der statischen bzw. optionalen Sichten, sowie für die Einträge des Phrasenprofiles (siehe auch 7.2.1). Ein Token kann dabei in mehreren Profiles mit einer Bedeutung versehen werden und somit unterschiedliche Informationen gleichzeitig repräsentieren. So zum Beispiel „SUBST“: dies könnte sowohl die Syntax, als auch Zugehörigkeit zu einer (statischen und/oder optionalen) Sicht über Substantive kennzeichnen.

Sondertoken und ihre Bedeutung

Der Generator akzeptiert neben dem Wissen bezüglich Sichten und Syntax noch weitere Informationen aus den Quellen. Diese Informationseinheiten müssen jedoch zur Unterscheidung durch folgende *reservierte Wörter* dem System „angekündigt" werden:

- Das Token **PHONETIK** kündigt einen String an, der die Phonetik eines Wortes darstellt.
- Die Einträge, die in die Token **BEGSYN** und **ENDSYN** geschachtelt sind, werden als Synonyme aufgefaßt.
- Das **NOINSERT**-Token teilt dem Lexikon lediglich mit, daß dieses Wort nicht in das Hauptwörterbuch eingetragen wird – eine Zeile der Quelle, die mit diesem Token versehen ist, wird also ignoriert.
- Durch **FREQUENZ** kann der Eintrag mit einer Frequenzangabe versehen werden. Dieser Wert kann z.B. dazu benutzt werden, ein Hochfrequenzwörterbuch mit den am häufigsten referenzierten Wörtern (Schwellwert) zu füllen [John93].

5.2 Ausgabedaten des Generators

Wesentlich bei der Generierung sind die Verzeigerungen bezüglich statischer und optionaler Sichten. Wird ein neues Wort in das Lexikon aufgenommen, so müssen alle Sichten, denen dieser Eintrag angehört, aktualisiert werden. Dazu wird zu jeder beteiligten Sicht der jeweils letzte Eintrag aufgesucht und als entsprechender Nachfolger die Position des neuen Begriffs eingetragen.

Neben dem virtuell partitionierten Hauptwörterbuch werden noch die Sichteinstiegsdateien erzeugt. In ihnen werden die Positionen des jeweils ersten Eintrages einer Sichtausprägung der statischen Sichten abgespeichert. Da hier der Zugriff über eine der beiden Hashfunktionen für ungenaue Erkennung geschieht, müssen diese Verweise für jeden Hashcode separat verfügbar sein.

Des weiteren werden die entstandenen Hashtabellen gespeichert, da sie zur Laufzeit benötigt werden. Sie enthalten im Falle der Hashfunktion 1 die Positionen des jeweils ersten Eintrages im Hauptwörterbuch und im Falle der beiden anderen Hashfunktionen (für Wortanfang bzw. Wortende) die Positionen in den Sichteinstiegsdateien. Die Größen dieser Dateien sind identisch mit denen der Hashtabellen.

Die notwendigen Informationen zu den optionalen Sichten wie zugeteilter Index einer Sicht und Position des ersten Eintrages einer Sicht werden in einer weiteren Datei gesichert.

5.3 Aufruf des Generators

Die Steuerung des Generierungsprozesses geschieht ausschließlich über Parameter – Interaktionen mit dem Benutzer sind nicht vorgesehen. Neben den Namen der Quellen und Profiles können hier von den Defaultwerten abweichende Hashtabellenlängen festgelegt werden. Da die Namen der Profiles sowie die Namen der generierten Dateien und die Hashtabellenlängen zur Laufzeit bekannt sein müssen, werden diese Informationen in einem System-Profile abgespeichert. Die Angabe dieser Datei ist somit ausreichend, um dem Laufzeitsystem alle benötigten Daten mitzuteilen.

Abbildung 5.4 zeigt den Generator mit beispielhaften Quellen und detailliert aufgeführten Profiles als Eingabe. Ausgaben des Prozesses sind die System-Steuerdatei, das die zur Laufzeit benötigten Daten birgt, und das virtuell partitionierte Wörterbuch mit Sichteinstiegsdateien und Hashtabellen.

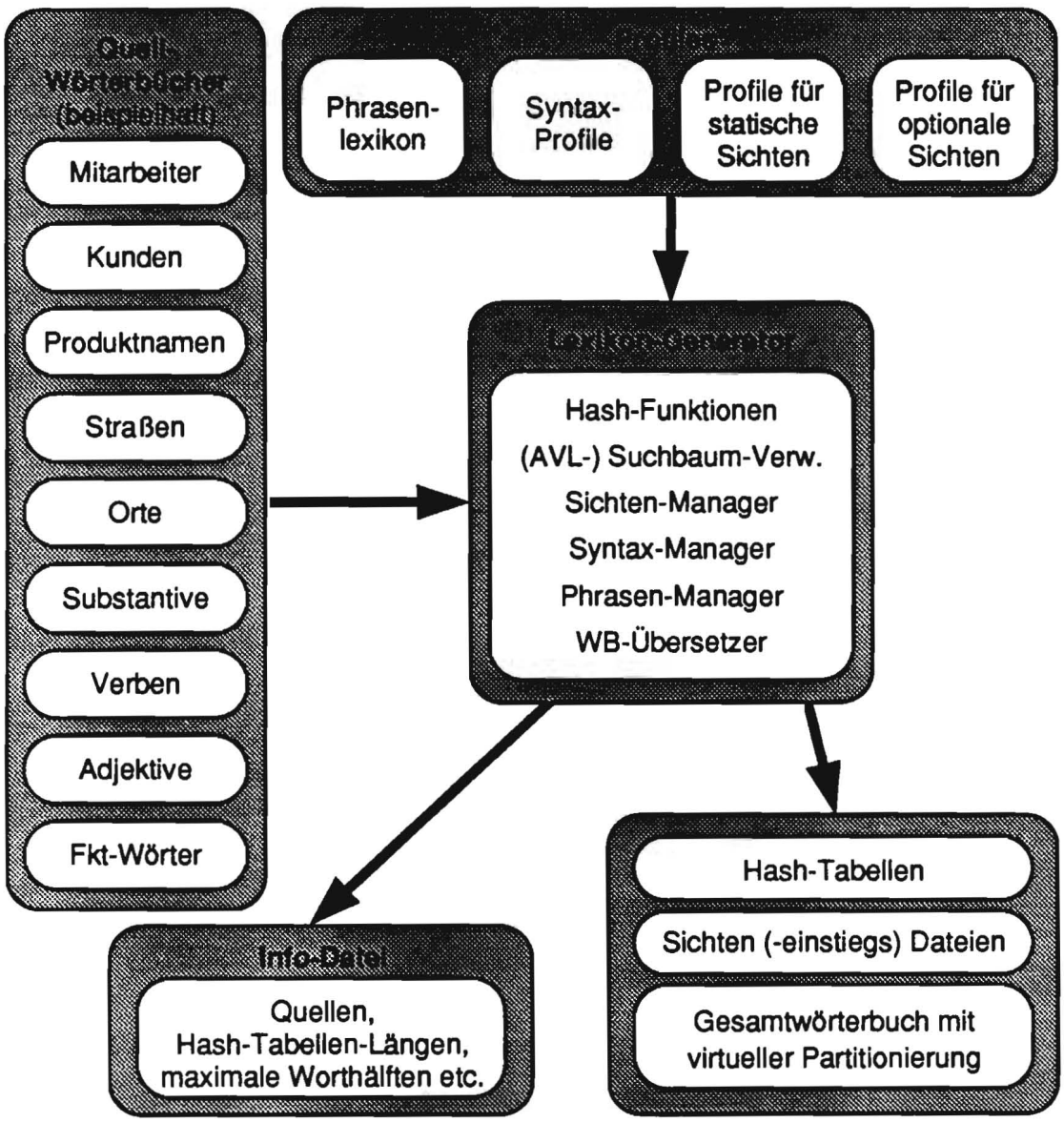

Abbildung 5.4: Der Generator und seine Ein-/ Ausgabe (detailliert)

5.4 Syntax-Profile: Überlegung spart Platz

Wie oben erläutert, steht zur Speicherung der Syntax ein 32-Bit-Wort zu Verfügung. Eine sehr einfache Art, Informationen hinein zu codieren, ist die Zuordnung je eines Tokens zu genau einem Bit. Allerdings wäre die Anzahl der unterschiedlichen Syntaxtoken auf 32 limitiert. Der Ausweg über mehr Informationszellen für die Syntax würde unsere Lexikon aufblähen, und widerstrebt somit unserem Ziel der kompakten Darstellung.

Wir wollen dabei ausnutzen, daß für Wörter unterschiedlicher Kategorie auch unterschiedliche syntaktische Informationen vorliegen, und daß diese Informationen gegebenenfalls mit den gleichen Bits des Syntax-Wortes codiert werden können. Dadurch erreichen wir eine effizientere Nutzung unserer Informationsträger.

Außerdem können verschiedene syntaktische Werte, die einem Eintrag exklusiv zugeordnet werden (z.B.: ein Wort ist entweder Substantiv, Verb, Adjektiv oder Funktionswort) zusammengefaßt werden, was die Codierung erneut kompaktiert.

Die Vorgehensweise soll an einem kleinen Beispiel veranschaulicht werden:
Will man für die Wortkategorie vier verschiedene Typen einräumen, so muß man eine Maske bereitstellen, die Raum für $4 + 1$ Werte in sich birgt (plus eins, da die Null für nicht-kategorisierte Wörter reserviert ist). Die Maske muß folglich 3 Bit breit sein und könnte dann $2^3 - 1 = 7$ verschiedene Typen aufnehmen. (Die oben erwähnte einfache Codierung kann mit 3 Bit nur 3 Typen aufnehmen.) Die folgenden Zeilen eines Syntax-Profiles sollen das Beispiel verdeutlichen. Die Werte von Muster und Maske werden als Binärzahlen angegeben, da sie so für den menschlichen Betrachter übersichtlicher sind und somit leichter erzeugt werden können. Im Beispiel beschränken wir uns auf eine 8 Bit Wortbreite.

	Token	Muster	Maske
SYNTAX	SUBST	00000001	00000111
SYNTAX	VERB	00000010	00000111
SYNTAX	ADJEKT	00000011	00000111
SYNTAX	FKTWRT	00000100	00000111

Üblicherweise will man in die verbleibenden 29 Bit (5 in unserem Beispiel) weitere Informationen aufnehmen, beispielsweise die Zeitform bei Verben, Geschlecht bei Substantiven oder Steigerungsform bei Adjektiven. Anstatt hier jeweils eigene Bereiche (Masken) zu definieren, kann mit einer gemeinsamen Maske gearbeitet werden. Diese muß groß genug sein um die größte der Gruppen aufzunehmen. Der Trick bei der Sache ist folgender: man erweitert die Maske auf die der Kategorie und fügt das jeweils richtige (Kategorie-) Muster an den entsprechenden Stellen hinzu. Auf diese Weise kann ein Bit-Bereich mehrfach genutzt werden. Unser Beispiel-Profile könnte so aussehen:

```
          Token      Muster       Maske
(Geschlecht bei Substantiven)
SYNTAX    MAENNL   00001001    00011111
SYNTAX     WEIBL   00010001    00011111
SYNTAX     NEUTR   00011001    00011111

(Zeitform bei Verben)
SYNTAX    GEGENW   00001010    00011111
SYNTAX    VERGAN   00010010    00011111
SYNTAX    ZUKNFT   00011010    00011111

(Steigerungsform bei Adjektiven)
SYNTAX   POSITIV   00001011    00011111
SYNTAX    COMPAR   00010011    00011111
SYNTAX     SUPER   00011011    00011111
```

Wird nun ein Syntaxtoken in ein Syntaxwort hineincodiert, so wird das Wort mit dem Muster des Tokens bitweise 'oder'-verknüpft. Bei der Decodierung wird das Syntaxwort zunächst mit der Maske eines Tokens bitweise 'und'-verknüpft. Stimmt das Ergebnis mit dem Muster des Tokens überein (sowohl Nullen als auch Einsen), so wissen wir, daß dieses Token zur Codierung herangezogen wurde.

Hier wird deutlich, daß der Designer des Profiles bei seiner Arbeit in hohem Maße für die Richtigkeit späterer Decodierergebnisse verantwortlich ist.

Kapitel 6

Laufzeitsystem

In diesem Kapitel soll nun auf das Laufzeitsystem und seine zahlreichen Zugriffsmöglichkeiten auf die Daten eingegangen werden. Die Architektur des Laufzeitsystems ist in Abbildung 6.1 zu sehen. Sie zeigt das Lisp-C-Interface, dessen Funktionalität fast ausschließlich in der Abbildung von Lisp-Funktionen auf die, des in C implementierten Lexikons besteht. Zur Initialisierung lädt der Controller das System-Profile und die übrigen vier Steuerdateien. Danach ist das System bereit, Anfragen zu bearbeiten, wobei es auf unterschiedliche Wörterbücher zugreift.

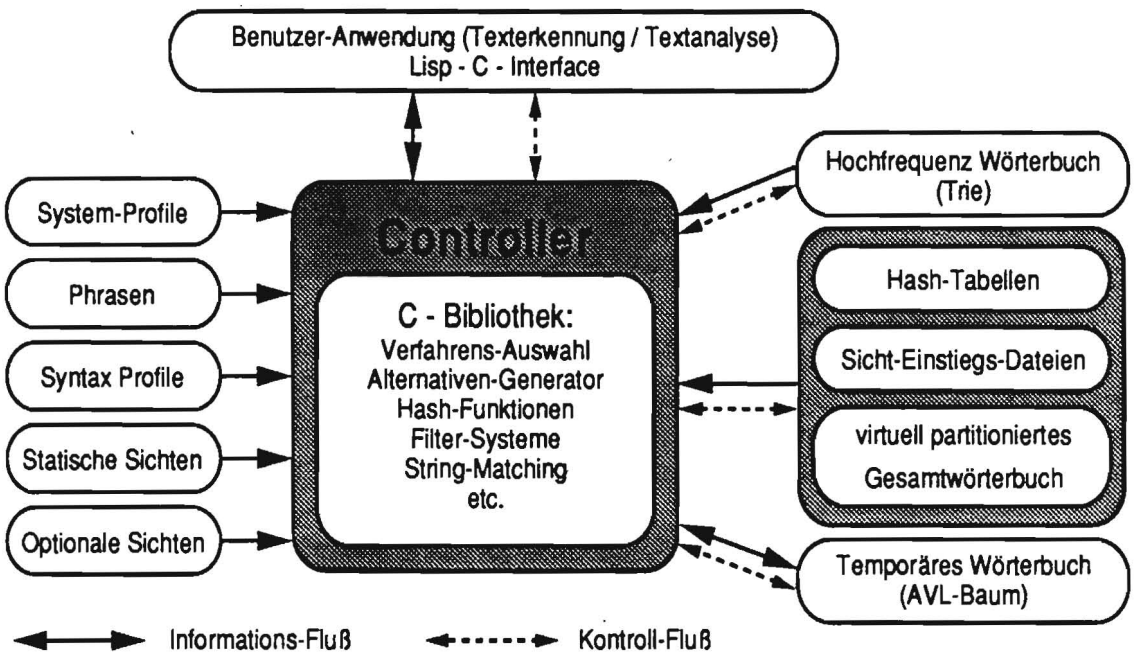

Abbildung 6.1: Laufzeitsystem des Lexikons

Zur Initialisierung des Systems muß das generierte System-Profile angegeben werden. In ihm stehen alle relevanten Informationen wie die Pfade zu den Profiles und Quellen, sowie

die Hashtabellen-Längen der einzelnen Hashfunktionen und die Pfade zu den generierten Dateien (siehe Abbildung 6.2).

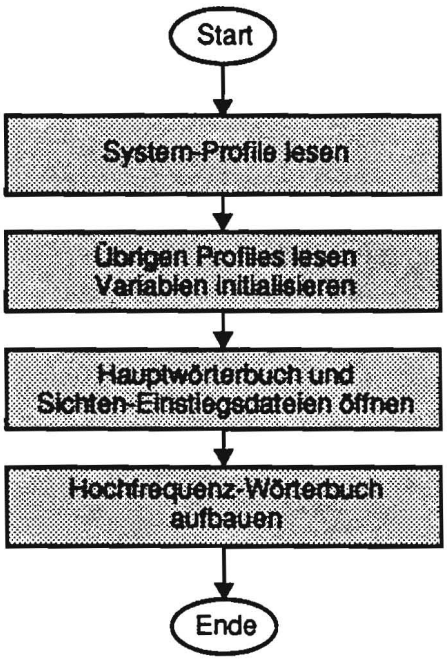

Abbildung 6.2: Flußdiagramm der Initialisierung des Laufzeitsystems

Die Indizes und Einstiegsadressen der optionalen Sichten werden in den Knoten eines AVL-Baumes abgespeichert, dessen Suchschlüssel die Namen der Sichten sind. Gleiches geschieht mit den Ausprägungen der statischen Sichten. Hier werden in den Knoten die vergebenen Indizes der Ausprägungen in den Datensätzen der Sichteinstiegsdateien sowie die Indizes der Sichten in den Datensätzen der Wörterbucheinträge gespeichert.

6.1 Anfrage mittels der Lesart

Eine typische Anfrage der Texterkennung ist die Spezifikation eines Strings als Suchmuster. Dabei können Unsicherheiten mit ins Spiel kommen, wie z.B. mehrere Alternativen für eine Position, bis hin zu Ungewißheiten über die Anzahl und Art der Buchstaben in einem Wort. Bei unserem System kann ein regulärer Ausdruck als Eingabe angegeben werden. Erlaubt sind Alternativen in eckigen Klammern „[eco]“, „?“ als Platzhalter für einen beliebigen Buchstaben und „*“ als Platzhalter für beliebig viele Buchstaben.

Das Flußdiagramm in Abbildung 6.3 zeigt die komplexe Arbeitsweise des Controllers bei der Vorbereitung des Systems auf diese Art der Anfrage. Dagegen sieht man in Abbildung 6.4 die Funktionalität beim zugehörigen Durchlaufen der Kollisionsketten.

Eine zulässige Eingabe wäre somit „c[oca]mpu[tf]?[rn]“, sie könnte beim Erkennen des Wortes „Computer“ entstanden sein. Dieser reguläre Ausdruck wird danach von einer Funktion (Controller) analysiert. Letzterer erkennt, ob es sich um ein vollständig erkanntes

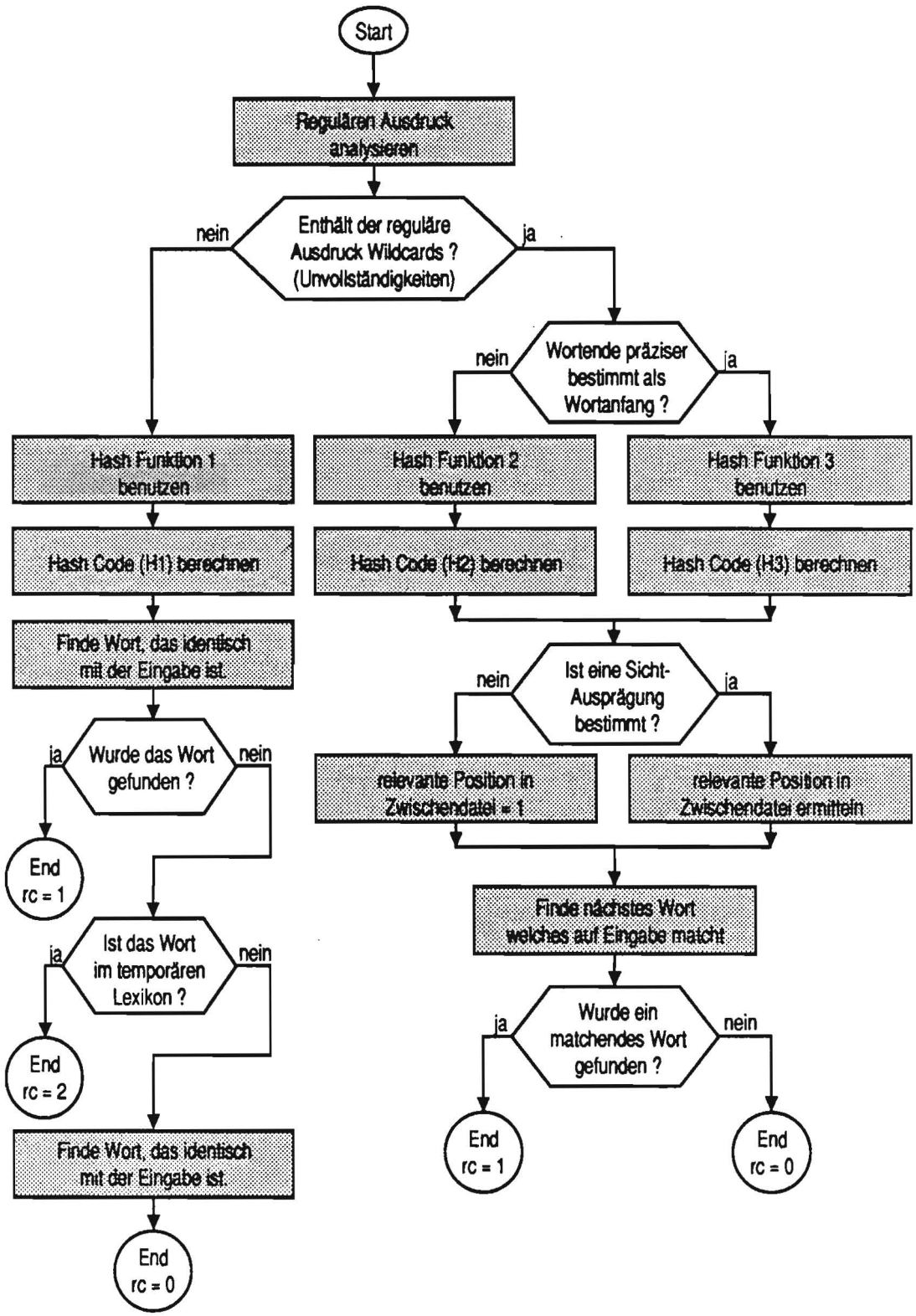

Abbildung 6.3: Vorbereitung der Anfrage mittels der Lesart

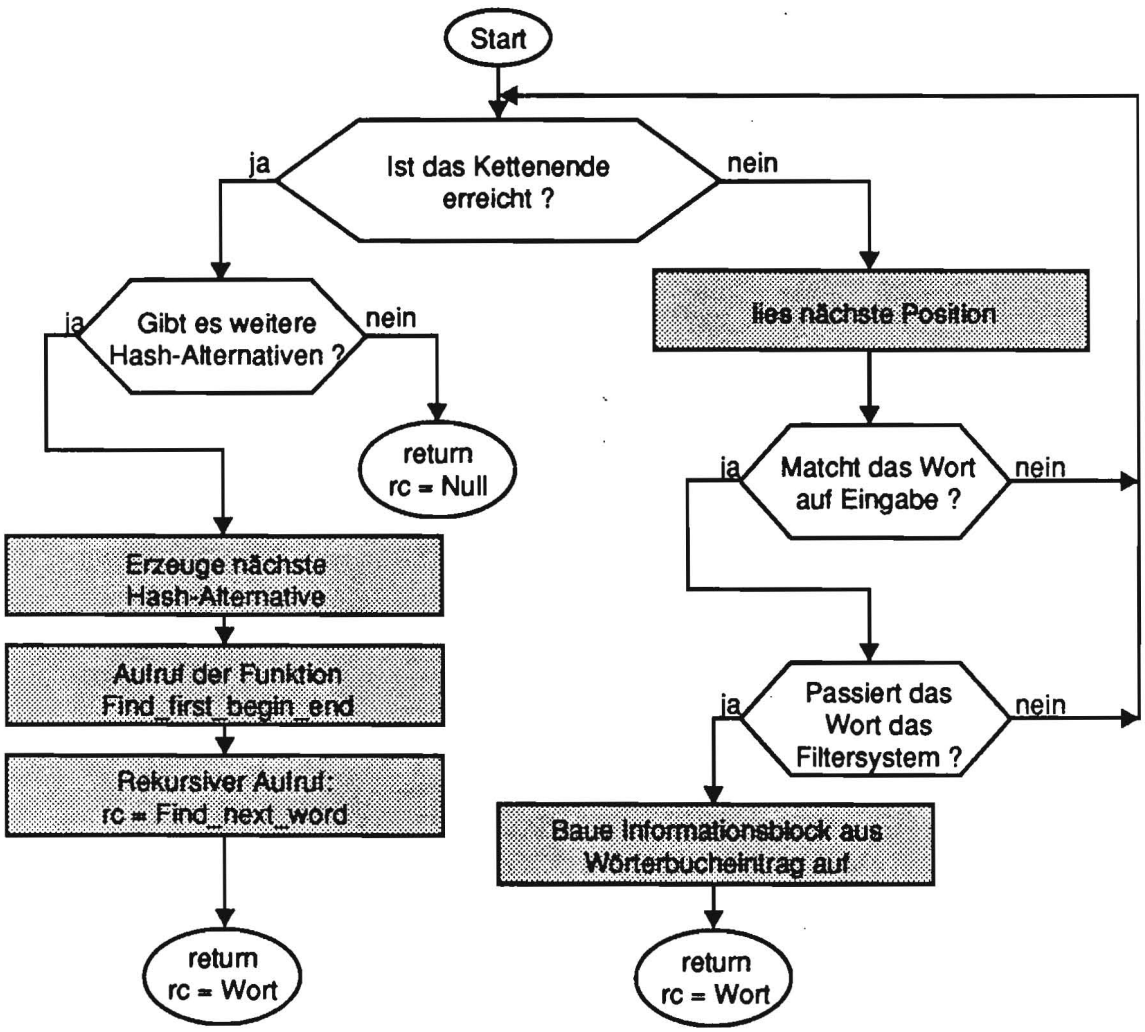

Abbildung 6.4: Durchlaufen der Ketten bei Anfragen mittels der Lesart

Wort handelt und der Zugriff über Hashfunktion 1 erfolgen kann. Falls dies nicht der Fall ist, werden die Anzahl der Alternativen für den Wortanfang bzw. für das Wortende ermittelt und daraufhin eine Entscheidung für eine der beiden Hashfunktionen 1 und 2 gefällt.

Wir wollen den Vorgang am obigen Beispiel „c[oca]mpu[tf]?[rn]" nachvollziehen. Die zweite Position enthält bereits Alternativen, so daß Hashfunktion 1 entfällt. Die Länge des Wortes ist genau bestimmt, da kein „*"-Wildcard enthalten ist, sie beträgt 8. Die Länge des charakteristischen Teilstrings ist das Minimum aus halber Wortlänge und einer Konstanten, die in unserer momentanen Implementierung den Wert 4 hat[9]. Die beiden Teilstrings für Wortanfang und Wortende lauten somit „c[oca]mp" und „u[tf]?[rn]". Der String „c[oca]mp" verursacht drei verschiedenen Anfragen an das Lexikon: „comp", „ccmp" und „camp". Das Wortende „u[tf]?[rn]" hingegen führt zu 120 verschiedenen Anfragen (**2** *Alternativen an Position 2* ***30** Alternativen an Position 3* ***2** Alternativen an Position 4* **= 120**). Der Controller entscheidet sich daraufhin für Hashfunktion 2. Für jede der drei Alternativen wird die entsprechende Kollisionskette im Hauptwörterbuch durchlaufen und die gefundenen Wörter werden mit dem Eingabestring verglichen. Wenn ein Match zustandekommt, wird die Anfrage beendet, der Returnwert ist 1. Wenn in keiner der Ketten ein passendes Wort gefunden wird, so liefert die Anfrage den Wert 0 zurück. Gleichzeitig wird eine Art Cursor verwaltet, der Informationen über die Anfrage und die Position des zuletzt gefundenen Eintrages hält. Dadurch ist es dem Benutzer möglich, die Suche dort fortzusetzen, wo sie gerade endete.

Im Erfolgsfall (Returnwert = 1) liefert daraufhin eine weitere Funktion das Wort zurück, welches auf die Eingabe matcht. Jeder erneute Aufruf dieser Funktion bewirkt ein Weitersuchen in der aktuellen Kette (bzw. einen Sprung auf die Kette einer nächsten Alternative) bis ein weiteres „passendes" Wort gefunden wurde und gibt bei positivem Ergebnis das entsprechende Wort zurück. Sind alle Ketten durchlaufen, so liefert die Funktion einen Nullpointer (in LISP: *nil*) zurück.

6.1.1 Verwendung einer statischen Sicht

Bei dieser Art der Anfrage hat der Benutzer die Möglichkeit sein zusätzliches Wissen hinsichtlich statischer Sichten einzubeziehen. Es wird jedoch nur dann genutzt, wenn es sich um ein *unvollständig erkanntes Wort* handelt, da sonst ohnehin mit Kollisionskettenlängen kleiner 2 gerechnet werden kann [Wagner92].

Über den AVL-Baum der Sichtausprägungen werden mittels des Ausprägungsnamens die benötigten Informationen (Index der Ausprägung I_A; Index der Sicht I_S) extrahiert und über den Hashcode aus der Sichteinstiegsdatei an Stelle I_A die Position des entsprechend ersten Eintrages dieser Ausprägung im Hauptwörterbuch gelesen. Von dort aus wird nicht über dem normalen Kollisionszeiger dieser Hashfunktion weitergesucht, sondern über den der entsprechenden Sicht I_S. Dadurch erspart man viele Leseoperationen auf dem Hauptwörterbuch und erhält somit kürzere Anfragezeiten.

[9] Dieser Wert (4) ergab sich nach [Wagner92] als sinnvoller Kompromiß aus den beiden Anforderungen, einerseits auf wenig Information zurückzugreifen und andererseits kurze Kollisionskettenlängen zu erhalten.

6.1.2 Temporäres Lexikon

Das in diesem Abschnitt vorgestellte temporäre Lexikon wird nur im Falle der Suche nach vollständig erkannten Wörtern aktiv und auch nur dann, wenn kein entsprechender Eintrag im Hauptwörterbuch vorliegt. Dies könnte durch neue Produktnamen, Personennamen oder falsch geschriebene Wörter geschehen. Die Aufgabe des temporären Lexikons liegt darin, solche Wörter aufzunehmen bzw. bei wiederholter Anfrage nach dem gleichen Wort den Benutzer darüber zu informieren.

Findet nun der Controller keinen passenden Eintrag im Hauptwörterbuch, so konsultiert er dieses temporäre Lexikon. Ist er dort fündig geworden, so wird ein Zugriffszähler, der zu jedem Wort verwaltet wird, inkrementiert und die Anfrage wird mit Returnwert 2 beendet. Ist die Suche negativ, so wird das Wort in das temporäre Lexikon aufgenommen und der Zugriffszähler auf 1 initialisiert. Der Returnwert ist in diesem Falle 0.

Wenn das Lexikonsystem beendet wird, so werden die Inhalte dieses Lexikons auf eine Datei gerettet. Zusätzlich zu den Begriffen werden auch die Zugriffshäufigkeiten auf die Datei geschrieben. Der Sinn dieser Vorgehensweise besteht darin, neue und häufig referenzierte Wörter zu erkennen und sie evtl. bei einer nächsten Generierung mit in das Hauptwörterbuch aufzunehmen. Zusätzlich können konsequent falsch geschriebene Wörter erkannt werden.

Solange kein neuer Generierungsprozeß gestartet wird, kann diese Datei (aber auch jede andere Wortliste) in das temporäre Lexikon geladen werden. Auf diese Weise können neue Wörter zumindest bei vollständiger Erkennung gefunden werden.

Das temporäre Lexikon kann jedoch nur dann durchsucht bzw. ergänzt werden, wenn ein Begriff vollständig erkannt wurde, da es nicht sehr zweckmäßig ist einen regulären Ausdruck als neues Wort aufzunehmen.

Als Datenstruktur wird in der momentanen Implementierung ein AVL-Baum verwendet. Dieser binäre Suchbaum zeichnet sich durch schnellen Zugriff aus, da er nach jeder Einfügeoperation ausbalanciert wird, wodurch keine zu tiefen Pfade entstehen. Eine weitere geeignete Struktur wäre der Trie, der bei Anfragen besser mit Unsicherheiten zurechtkommt. Die Ersetzung des AVL-Baumes durch den Trie ist eine mögliche Erweiterung bzw. Verbesserung des Systems[10].

[10] Im Rahmen einer im ALV-Projekt vergebenen Arbeit wird eine C-Bibliothek zur Verwaltung von Tries implementiert, die, sobald fertiggestellt, in unser System integriert wird.

6.2 Anfrage mittels einer optionalen Sicht

Das Konzept der optionalen Sichten entstand aus einer Forderung der Textanalyse heraus und wird demzufolge auch vorzugsweise von dieser benutzt. Dabei ist eine Menge von Einträgen im Lexikon unter einem Sichtnamen zusammengefaßt. Durch Angabe dieses Namens soll es möglich sein, die zugrundegelegten Wörter abzufragen. Wird nun eine solche Anfrage an das Lexikon gestellt, so wird ein AVL-Baum nach dem Sichtnamen durchsucht. Im Erfolgsfalle kennt das System jetzt die Position des ersten Eintrags dieser Sicht und den Index dieser Sicht, den man zur Ermittlung des Nachfolgezeigers benötigt. Die Anfrage ist danach beendet und hat den Returnwert 1. Falls eine Sicht mit entsprechendem Namen nicht gefunden wird, ist der Returnwert 0.

Eine weitere Funktion erlaubt das Durchlaufen der Sichtkette. Sie liefert das jeweils gefundene Wort zurück bzw. einen Nullpointer (in Lisp: *nil*), falls das Ende der Sicht erreicht ist.
Damit haben die Anfragen mittels der Lesart und die mittels einer optionalen Sicht eine Gemeinsamkeit: eine erste Funktion initialisiert das Lexikon auf eine bestimmte Art der Suche und eine weitere durchläuft das Lexikon und liefert die gefundenen Einträge zurück. Möchte man auf Lisp-Seite komplette Listen aller Wörter erzeugen, die eine Suche liefert, so muß man dort nur eine kleine Schleife programmieren, die sukzessive alle Anfrageergebnisse in einer Liste sammelt. Beispielfunktionen solcher Schleifen finden sich in der Datei der Schnittstellenfunktionen (siehe Anhang A.1).

6.2.1 Nachfolgerverzeigerung

Die Mikrostruktur (Abbildung 4.11) des Hauptwörterbuches hat einen ersten homogenen Teil, in welchem u.a. die normalen Kollisionspointer bzw. die der statischen Sichten untergebracht sind. Dadurch ist es sehr einfach, den richtigen Kollisionszeiger zu finden. Anders sieht die Situation bei den optionalen Sichten aus. Ihre Verzeigerung steht in

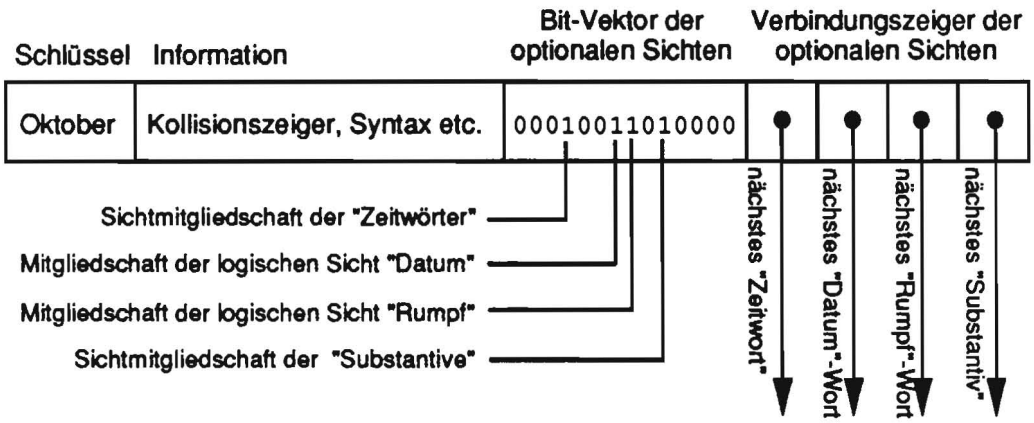

Abbildung 6.5: Realisierung optionaler Sichten

einem zweiten Teil, der sich durch unterschiedliche Längen auszeichnet. In ihm sind z.B. die Zeichenkette des Wortes selbst sowie optionale Informationen wie Phonetik, Synonyme und eben die Verzeigerungen der optionalen Sichten kodiert. Abbilding 6.5 zeigt einen Ausschnitt der Mikrostruktur, der den Bitvektor und die Verzeigerung der optionalen Sichten trägt.

Zur Berechnung des Nachfolgezeigers benötigt das System den Bitvektor, der die Zugehörigkeiten zu optionalen Sichten kennzeichnet, den Index der Sicht, die man gerade abläuft und die Stelle des ersten Zeigers der optionalen Sichten. Der Index ist aus dem AVL-Baum bekannt, der Bitvektor ist dem homogenen Teil der Daten zu entnehmen und die Stelle des ersten Zeigers ist direkt nach der Zeichenkette des Eintrages. Für jede optionale Sicht, der ein Eintrag angehört, muß ein Nachfolgezeiger bereitstehen und dies ist gleich der Anzahl gesetzter Bits im Bitvektor. Zur Sicht des ersten gesetzten Bits gehört der erste Nachfolgezeiger, zu der des zweiten gesetzten Bits der zweite und so fort. Kennt man also den Index der Sicht, so muß man ermitteln, wieviele Bits mit kleinerem Index gesetzt sind und bekommt damit die Stelle des entsprechenden Nachfolgezeigers.

6.3 Filtersystem

Prinzipielle Aufgabe eines Filters ist die Selektion von Objekten. Dies erreichen auch die in unserem System integrierten Filter. Die Information, die das Passieren eines Filters entscheidet, steckt in dem Bitvektor und ist so mit den optionalen Sichten verknüpft. Folglich werden auch die Namen dieser Sichten herangezogen, um einzelne Filter zu aktivieren.

Ist mindestens einer der Filter gesetzt, so werden nur diejenigen Wörter an den Benutzer des Lexikons zurückgeliefert, die einen der Filter passieren. Diese gilt sowohl bei Anfragen über optionale Sichten als auch bei Anfragen über die Lesart eines Wortes. Man kann so die Menge der zurückgelieferten Wörter einschränken, was den Datenfluß über die Lisp-C-Schnittstelle reduziert.

Setzt man nun einen Filter auf Sicht „X", so werden bei *jeder* Anfrage nur diejenigen Einträge zurückgeliefert, welche Elemente der Sicht „X" sind. Zudem können logische Verknüpfungen dieser Randbedingungen realisiert werden, indem ein Filter mehrfach gesetzt wird (Konjunktionen) bzw. mehrere Filter parallel gesetzt werden (Disjunktionen).

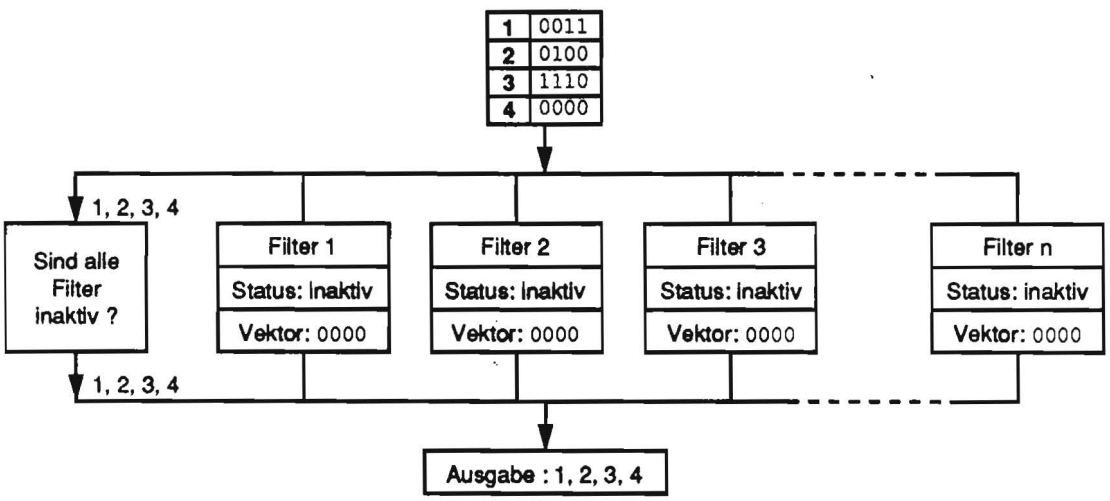

Abbildung 6.6: Funktionsweise eines *inaktiven* Filtersystems

Abbildung 6.6 zeigt ein Filtersystem mit vier beispielhaften Eingaben und deren 4-Bit breiten Vektoren. Sämtliche Eingaben durchlaufen die linke Box, da alle Filter den Status *„inaktiv"* haben.

6.3.1 Verknüpfung von Filterbedingungen

Das Lexikon gibt alle Wörter zurück, die mindestens einen der Filter passieren. Setzt man also einen Filter auf Sicht „X" und einen anderen auf Sicht „Y", so akzeptiert unser Filtersystem alle Einträge, die entweder in *„X" oder „Y"* liegen. Dies bewirkt also eine *Disjunktion.*

Demgegenüber wird eine *Konjunktion* realisiert, indem ein Filter sowohl auf Sicht „X" als auch auf „Y" gesetzt wird. Das so aktivierte Filtersystem läßt nur solche Wörter passieren, die in beiden Sichten (*„X" und „Y"*) enthalten sind.

Dies verdeutlicht Abbildung 6.7, die zwei aktive Filter (1 und 2) zeigt. Ihr Status ist *„aktiv"* und ihr Bitvektor gibt an, an welchen Stellen im Vektor der Eingabe eine 1 stehen muß, damit sie den Filter passieren kann.

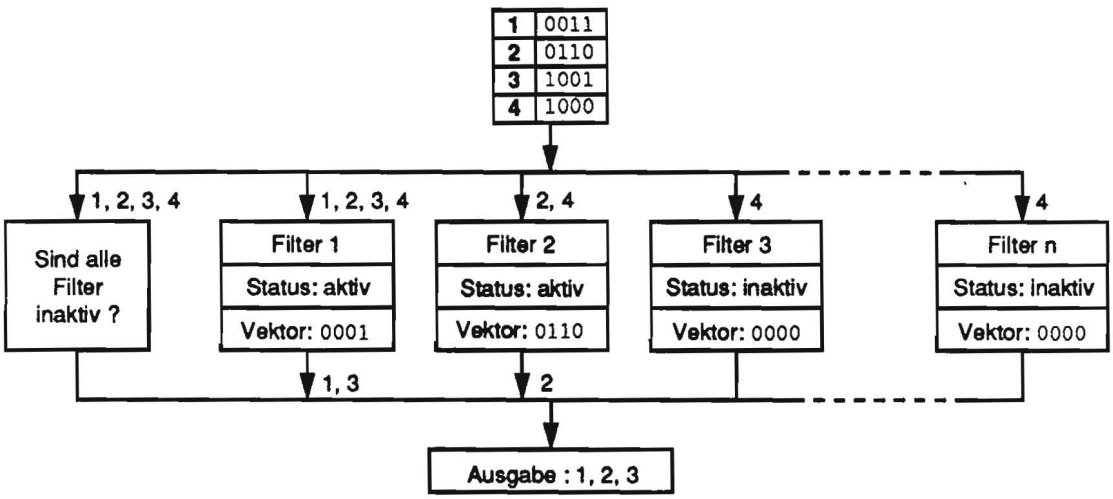

Abbildung 6.7: Funktionsweise bei *zwei aktiven* Filtern

Der Benutzer hat darüberhinaus die Möglichkeit, die *Arbeitsweise* eines Filters zu *negieren.* Wird ein Filter auf Sicht „X" gesetzt und anschließend seine Funktion invertiert, so läßt der Filter nur diejenigen Einträge hindurch, die *nicht in Sicht „X"* enthalten sind.
Wurde der negierte Filter zuvor auf *mehrere Sichten* gesetzt, so akzeptiert er in seiner invertierten Funktion nur diejenigen Wörter, die *weder in „X" noch in „Y"* liegen.

Dies wird in Abbildung 6.8 bei Filter 2 deutlich. Seine Statusvariable hat den Wert *„invers"*. Er läßt somit nur solche Einträge passieren, deren Vektor an allen gesetzten Stellen im Filtervektor eine Null hat.

Ein kleines Beispiel soll dies veranschaulichen: man setzt Filter 1 auf die Sichten „Substantive" und „Adreßteil". Filter 1 würde dann nur diejenigen Wörter passieren lassen, die sowohl in der Sicht „Substantive" als auch in der Sicht „Adreßteil" vorkommen.

Setzt man zusätzlich Filter 2 auf „Straßen", so werden alle Wörter zurückgegeben, die entweder in der Sicht „Straßen" oder in den Sichten „Substantive" und „Adreßteil" enthalten sind.

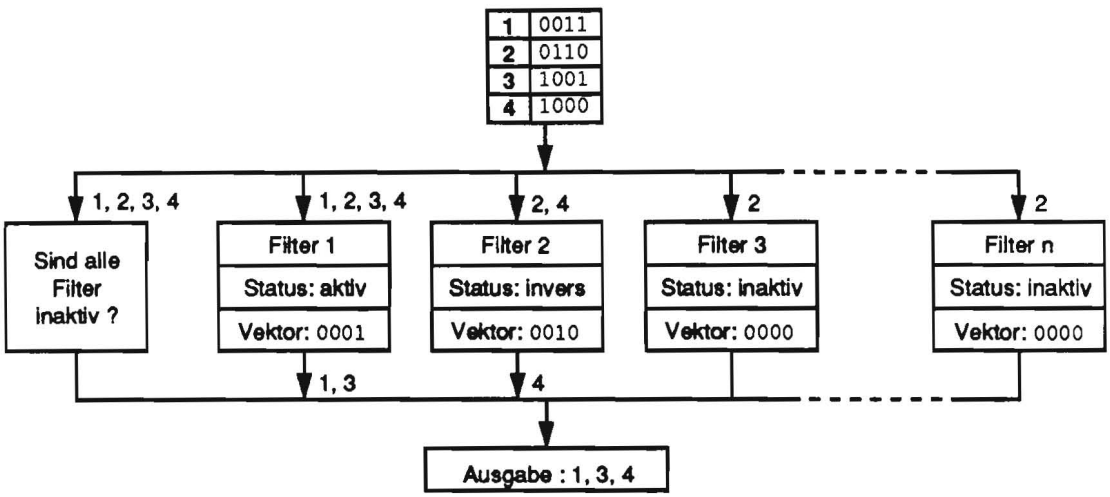

Abbildung 6.8: Funktionsweise bei einem *aktiven* und einem *inversen* Filter

Wird nun auch noch Filter 3 auf „Funktionswörter" gesetzt und anschließend negiert, so läßt er nur Kandidaten passieren, die nicht in dieser Sicht enthalten sind. Das komplette Filtersystem akzeptiert nun alle Wörter, die entweder in der Sicht „Straßen" oder in den Sichten „Substantive" und „Adreßteil" oder aber *nicht* in der Sicht „Funktionswörter" beinhaltet sind.

Die momentane Implementierung verfügt über acht Filter mit den Indizes 0 bis 7. Diese Filter sind identisch und es besteht keine Restriktion, die Filter in Reihenfolge ihrer Indizes zu aktivieren. Eine Erweiterung der Anzahl kann durch Ändern des Wertes **MAXFILTER** in der Datei `constant.h` und anschließende Neuübersetzung erfolgen.

6.3.2 Arbeitsweise des Filtersystems

Das Passieren eines Filters geschieht wie folgt: Ein Filter wird durch eine Statusvariable S_F und einen Bitvektor B_F repräsentiert. Letzterer ist typgleich mit dem Bitvektor, der den Einträgen zugeordnet ist (B_E). Ist S_F in einem Initialzustand (*„inaktiv"*), so ist dieser Filter nicht aktiviert. Hat S_F den Wert *„aktiv"*, so arbeitet der Filter auf normale Weise. Dabei wird für jedes gesetzte Bit in B_F überprüft, ob auch das entsprechende Bit in B_E gesetzt ist [11]. Nur wenn alle Tests positiv sind, läßt dieser Filter das Wort passieren.

Wird ein Filter negiert, so hat S_F den Wert *„invers"* und der Vergleich von B_F und B_E arbeitet folgendermaßen: Für jedes gesetzte Bit in B_F wird überprüft, ob sein Pendant in B_E nicht gesetzt ist. Falls doch, so läßt der Filter die Eingabe nicht hindurch.

[11] Die hierzu benötigten Bitoperationen sind in 'C' sehr schnell realisierbar

6.4 Abfragen lexikalischer Information

Bislang wissen wir, welche Möglichkeiten unser Lexikon bietet, um bestimmte Wörter aufzusuchen. Aber wie kann man die zusätzlichen Informationen zu einem Wort abfragen?

Hierzu existieren bereits einige Funktionen, die jedoch noch beliebig erweitert werden können.

1. Eine Funktion decodiert aus dem 32-Bit-Wort S_E der Syntax die Token, welche schon zur Codierung vorhanden waren. Die Informationen, die den Token im Syntax-Profile zugeordnet sind, sind ein charakteristisches Muster S_T und eine Maske M_T. Diese Maske M_T wird nun über S_E gelegt, d.h. beide Werte werden bitweise „und"-verknüpft. Sind Ergebnis und S_T identisch, so wird das entsprechende Token zurückgegeben.

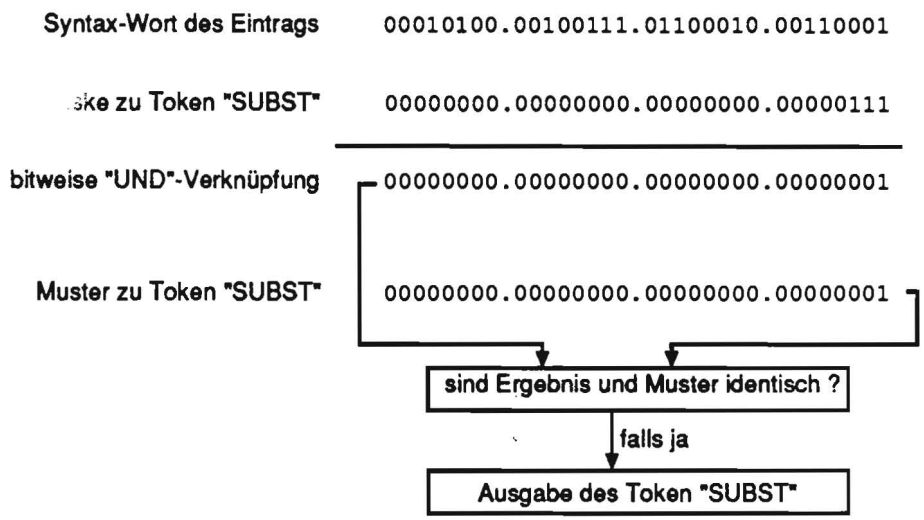

Abbildung 6.9: Decodierung der Syntax mittels des Syntax-Profiles

So werden nach um nach all die Syntax-Token an den Benutzer zurückgegeben, die zur Codierung angegeben wurden. Dabei kommt dem Designer des Syntax-Profiles eine hohe Verantwortung zu. Gruppen von Token, die exklusiv auftreten (Bsp. „Subst", „Verb" oder „Adjekt") müssen die gleiche Maske haben. Diese Maske muß breit genug sein, um alle verschiedenen, ihr untergeordneten Token codieren zu können. Darüberhinaus muß S_T mit M_T bitweise „und"-verknüpft identisch zu S_T sein.

Die Vorgehensweise bei der Übermittlung der Daten zwischen Lexikon-System und Lisp-Funktion geschieht durch wiederholten Aufruf der Funktion. Das System liefert mit jedem Aufruf das jeweils nächste Token (oder *nil*) zurück. Hier läßt sich eine einfache Schleife programmieren, die sämtliche Token in einer Liste sammelt.

2. Eine andere Funktion liefert die Namen der optionalen Sichten zurück, denen ein Wort angehört. Zu diesem Zweck wird vom System eine indizierte Liste verwaltet, mit der man über den Index einer optionalen Sicht I_S auf deren Namen kommt. Die Indizes der Sichten erhält man über den Bitvektor, man sucht in ihm nach gesetzten Bits — deren Index im Vektor ist gleich dem der Sicht I_S. Der Datenaustausch zwischen Lexikon und Lisp-System geschieht wie oben.

3. Die Phrasen, in denen ein gegebenes Wort enthalten ist, können ebenfalls ermittelt werden. Hierfür liegt im inhomogenen Teil der Mikrostruktur eine Verweisliste auf sämtliche Positionen im Phrasenlexikon. Ist diese Liste nicht leer, so können nach um nach alle betreffenden Phrasen ausgelesen und an den Benutzer zurückgegeben werden.

4. Falls zu einem Eintrag Synonyme angegeben wurden, so können diese von einer weiteren Funktion abgefragt werden. Auch hier werden die Daten auf die obige Weise sukzessive zurückgegeben.

Die Information, die das Lexikon zu einem Eintrag speichert, kann beliebig ergänzt werden. Das Lexikon-System ist offen für Erweiterungen, die allerdings nur dann notwendig werden, wenn entsprechendes Wissen zur Verfügung steht und von den Benutzern (Texterkennung bzw. Textanalyse) tatsächlich genutzt werden kann.

Kapitel 7

Implementierung

7.1 Die Entwicklungsumgebung

Als Entwicklungsumgebung stehen im ALV-Projekt SUN-SPARCstations mit dem Betriebssystem Unix zur Verfügung. Die Prototypen der Texterkennung und Textanalyse sind in Common Lisp implementiert, was die Verwendung der gleichen Programmiersprache zur Entwicklung des Lexikons nahelegt. Es wurde jedoch 'C' als Programmiersprache gewählt, da sie sehr viel schnelleren Code erzeugt und die Verwaltung der Speicherressourcen dem Benutzer überläßt. Die Performanz ist letztendlich ein wichtiges Kriterium bei der Konzeption des Systems. Die Quellen der momentanen Implementierung umfassen 5.010 Zeilen C-Code und 294 Zeilen Lisp-Code [12].

Zur Verbindung beider Systeme werden dabei Schnittstellenfunktionen benötigt, die es den Lisp-Programmen erlauben, auf das Lexikon zuzugreifen. Das LUCID-Common Lisp stellt hierfür ein *foreign function interface* bereit, mit dessen Hilfe die übersetzten Objekte der C-Quellen sehr komfortabel eingebunden und fremde Funktionen aufgerufen werden können [Lucid]. Die Übergabe von Parametern und Ergebnissen von Lisp nach C und von C nach Lisp geschieht nur mittels Werteparametern. Möchte man beispielsweise einen String übergeben, so genügt es nicht, die Adresse des Strings mitzuteilen, da der Garbage Collector des Lisp Systems den String evtl. an eine andere Stelle kopiert. Ferner ist die interne Darstellung von Zeichenketten und Zahlen nicht identisch. Aus diesem Grund muß folgendermaßen verfahren werden (siehe Abbildung 7.1):

1. Allokieren eines Speicherbereiches außerhalb des Einflußbereiches des Garbage Collectors.

2. Konvertieren des Strings von der Lisp- in die C-Darstellung.

[12] Die Werte beziehen sich auf Generator und Laufzeitsystem. Dabei werden sehr viele Funktionen (AVL-Baum Verwaltung, Leseroutinen der Profiles, Tokenisierungsfunktion und Bitvektor Operationen) in beiden Teilen verwendet. Der Lisp-Code enthält im wesentlichen nur die nötigen Deklarationen zum Aufruf der C-Funktionen.

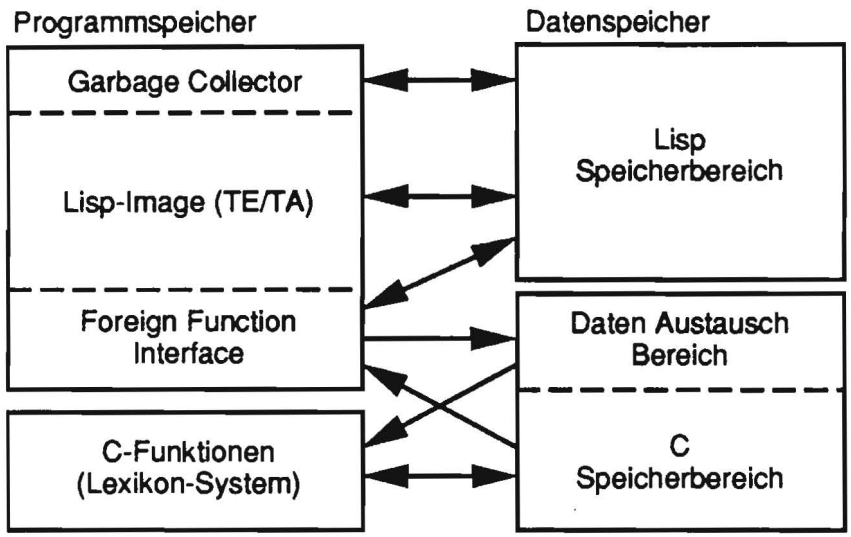

Abbildung 7.1: Datenaustausch zwischen Lisp- und C-Funktionen

Ähnlich wird bei der Übergabe von Zahlen sowie beim Zurückliefern von Ergebnissen verfahren. Dies macht die Schnittstelle etwas langsam und veranlaßte uns, zur Minimierung des Datenflusses die nahezu komplette Funktionalität auf die C-Seite zu verlagern.

7.2 Algorithmen und Datenstrukturen

Dieser Abschnitt soll nur eine Auswahl der wichtigsten Algorithmen und Datenstrukturen vorstellen.

7.2.1 AVL-Bäume

Da das vorgestellte System mehrfach über Zeichenketten auf Informationen zugreifen muß, ist eine geeignete Datenstruktur notwendig. Diese kann bei den zu erwartenden, relativ geringen Datenmengen im Arbeitsspeicher organisiert sein. Beispiele für die Anwendungsbereiche sind die Ermittlung der an die Syntax-Token gebundenen Muster und Masken, oder der Einstiegsadressen und Indizes der einzelnen optionalen Sichten sowie der Informationen der zur Definition von statischen und optionalen Sichten spezifizierten Token. Selbst das temporäre Wörterbuch bedient sich dieser Struktur.

In der gegenwärtigen Implementierung werden diese Aufgaben von *ausgeglichenen binären Suchbäumen* erledigt. Es wird jedoch keine völlig ausgeglichene Struktur realisiert, da die dafür notwendigen Einfügeoperationen sehr komplex und zeitaufwendig sind. Eine etwas schwächere Definition der Ausgeglichenheit stammt von Adelson-Velskii und Landis [Wirth75]:

> Ein Baum ist genau dann *ausgeglichen*, wenn sich für jeden Knoten die *Höhen* der zugehörigen Teilbäume um höchstens 1 unterscheiden.

Bäume, die diese Eigenschaften erfüllen, werden nach ihren Schöpfern *AVL-Bäume* genannt. Die Ausgleichs-Prozeduren für AVL-Bäume sind relativ einfach und schnell. Es gibt zwei unterschiedliche Arten der Umstrukturierung – je nachdem in welcher Form der Baum aus der Balance gerät. Abbildung 7.2 verdeutlicht die *einfache Rotation* nach rechts, zu ihr gibt es auch das spiegelbildliche Pendant, die *Linksrotation.*

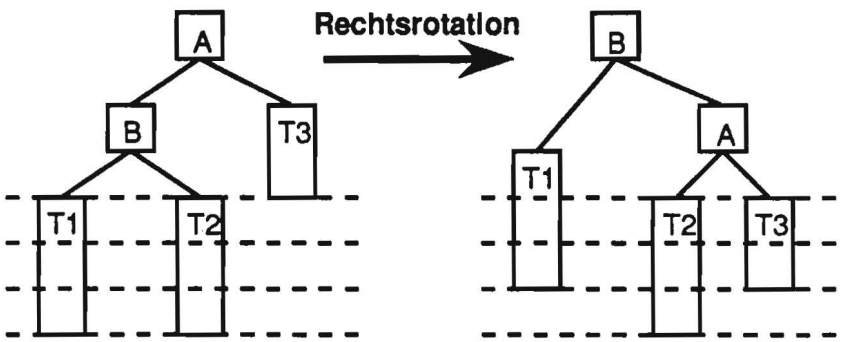

Abbildung 7.2: Operation „RR“ zur Wiederherstellung der AVL-Ausgeglichenheit

Die zweite Art der Rekonstruktion, die *Doppelrotation*, wird in Abbildung 7.3 gezeigt. Auch hier gibt es natürlich eine Doppelrotation nach links.

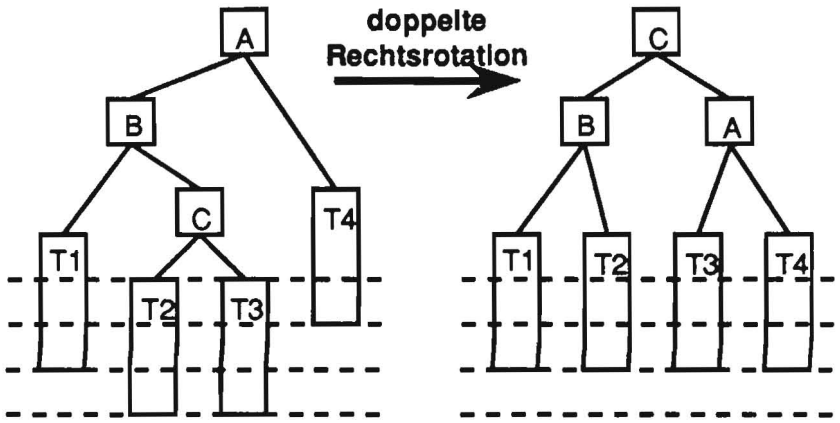

Abbildung 7.3: Operation „DRR“ zur Wiederherstellung der AVL-Ausgeglichenheit

Sowohl Einfügen und Löschen mit anschließender Rekonstruktion als auch Suchen im AVL-Baum mit n Knoten lassen sich im ungünstigsten Fall mit einer Zeitkomplexität von $O(logn)$ erledigen [Wirth75].

Informationsstruktur der Baumknoten

Ein Modul der Lexikonsystems beinhaltet die zur Verwaltung von AVL-Bäumen notwendigen Funktionen zum Einfügen, Löschen und Suchen. Die Daten, die an die Knoten gebunden werden können, sind je nach Anwendung unterschiedlich. Die folgende C-Typdefinition zeigt die Struktur der Knoteninformation:

```
typedef struct {
    char                *key;
    union {
        PD_Next_phrase  *next_phrase;
        int              temp_occurance;
        struct {
            MD_Syntax_type   syntax_code;
            MD_Syntax_type   bitfieldrange;
        } syntax;
        struct {
            int              sicht_index;
            int              auspr_index;
        } sicht;
        struct {
            long             first_entry;
            long             last_entry;
            int              view_index;
            MD_Bitvektor     match_vektor;
        } log_view;
        struct {
            MD_Bitvektor     vektor;
        } view_merkmal;
        struct {
            MD_Bitvektor    vektor;
            char            is_written;
        } view_entries;
    } data;
} SB_Info_type;
```

Der C-kundige Leser erkennt den Suchschlüssel *„key“* und die unterschiedlichen, alternativ verwendeten *union*-Untertypen des *„data“*-Bereichs (next_phrase, temp_occurance, syntax, sicht, log_view, view_merkmal und view_entries) für die einzelnen Einsatzbereiche der AVL-Bäume.

7.2.2 Matching

Exaktes Matching

Bei Anfragen mittels der Lesart werden die Wörter in den Kollisionsketten mit dem Eingabestring verglichen. Da die Eingabe ein regulärer Ausdruck sein kann, ist eine herkömmliche Vergleichoperation, wie die Standard C-Funktion *strcpy* nicht brauchbar.

Um diese Operation möglichst schnell zu machen, wird der reguläre Ausdruck gewissermaßen vorübersetzt. Dadurch werden '?'-Wildcards und in Brackets '[eco]' eingeschlossene Buchstabenalternativen ebensoschnell behandelt, wie eindeutige Zeichen an einer Position. Problematisch wird nur der '*'-Wildcard, da er ein Platzhalter für beliebig viele Buchstaben ist. In diesem Fall muß für jeden verbleibenden Teilstring überprüft werden, ob er mit dem Rest des regulären Ausdrucks matcht.

Approximatives Matching

Interessanter wird der Vergleich, wenn wir parametergesteuerte Fehler beim Vergleich zulassen. Hierzu eignet sich die sogenannte *gewichtete Levenshteindistanz*, die die kostengünstigsten Edieroperationen zur Überführung eines Strings in einen anderen ermittelt. Dabei werden den drei häufigsten Fehlern, *Einfügen*, *Löschen* oder *Ändern* eines Buchstabens, jeweils eigene Kosten (k_e, k_l, $k_{\bar{a}}$) zugeteilt, die durch die entsprechende Operation entstehen [Bunke90]. Zeichenketten, deren Levenshteindistanz *ld* einen vorgegebenen Wert d_{max} nicht überschreiten, werden durch diese approximative Matchoperation als „gleich" ausgewiesen.

Die Zuordnung von k_e, k_l, $k_{\bar{a}}$ und d_{max} kann das Verhalten der Vergleichsoperation vollkommen verändern. Setzt man alle Kosten auf 1 und d_{max} auf 2, so würden alle Wörter als gleich erkannt, die durch zwei beliebige der drei Edieroperationen ineinander übergehen. Ist dagegen $k_e = 0$ und $d_{max} = 0$, $k_l = 1$ und $k_{\bar{a}} = 1$, so akzeptiert die Match-Funktion alle Wörter, die durch beliebiges Einfügen von Zeichen in die andere Zeichenkette übergehen.

Die Integration eines solchen approximativen Matchens ist Bestandteil weiterführender Verbesserungen und Erweiterungen an der momentanen Implementierung. Zu beachten ist jedoch, daß bei Fehlern innerhalb des zur Hashcodeberechnung verwendeten Teilstrings das gesuchte Wort nicht gefunden werden kann. In diesem Fall müßte der Controller einen weiteren Zugriff über eine andere Hashfunktion initiieren.

7.2.3 Alternativengenerator

Der Alternativengenerator hat die Aufgabe, die Eingabe in Form eines regulären Ausdrucks „auszumultiplizieren", d.h. er erzeugt sukzessive alle möglichen Kombinationen aus den Einzelalternativen einiger Positionen (hypothesize & test strategie). Dazu wird er zunächst durch Angabe des regulären Ausdrucks initialisiert. Danach liefert er nach und nach alle Worthypothesen zurück bzw. einen Nullpointer, wenn es keine weiteren Alternativen mehr gibt.

Dabei ist auch zu beachten, daß bei Eingaben, deren Länge nicht genau bestimmt werden kann, auch die Längen des signifikanten Hashstrings variieren. Tabelle 7.1 zeigt alle zu erzeugenden Alternativen bei Eingabe von „[oca]o[nm]p*r". Der Controller erkennt in diesem Fall, daß bei Anfrage über den Wortanfang weniger viele Alternativen zustandekommen als beim Wortende, da das vorletzte Zeichen ein '*'-Wildcard ist. Bedingt durch diesen Wildcard ist auch die Gesamtlänge der Wortes nicht exakt bestimmt, sie beträgt jedoch mindestens 5. Für den Alternativengenerator bedeutet das, daß der signifikante Hashstring eine Länge zwischen 2 und 4 hat.

Alternativen bei Eingabe „[oca]o[nm]p*r"		
Länge 2	Länge 3	Länge 4
oo	oom	oomp
co	oon	oonp
ao	com	comp
	con	conp
	aom	aomp
	aon	aonp

Tabelle 7.1: Ausgaben des Alternativengenerators

Man erkennt, daß diese Eingabe zu 15 verschiedenen Anfragen an das Lexikon führt[13].

[13] Wenn zwei oder mehr verschiedene Alternativen auf die gleiche Hashadresse abgebildet werden, so wird dies vom System erkannt und entsprechend weniger Ketten durchlaufen.

7.3 Sprachspezifische Probleme

Bei Anfragen an das Lexikonsystem treten speziell im Deutschen einige Probleme und Fragen auf. Diese resultieren aus der unterschiedlichen Groß-/ Kleinschreibung, den Umlauten (ä, ö, ü) und dem scharfen 's' (ß) sowie den Bindestrichkomposita:

1. Die Fernschreiber vergangener Tage waren nicht in der Lage *Großbuchstaben* zu drucken, da man damals aus Kostengründen auf diese Unterscheidung verzichtete. Ferner verwenden manche Autoren große Lettern zum HERVORHEBEN von Texten, falls das vorhandene Schreibsystem keine andere Möglichkeit wie etwa **Fettdruck**, *Kursivschrift* oder KAPITÄLCHEN bietet. Schließlich werden am Beginn eines Satzes ohnehin alle Wörter groß geschrieben.

   Um diesem Problem zu entgehen, werden alle Einträge im Lexikon klein geschrieben. Zur Laufzeit werden Anfragen ins Kleingeschriebene überführt. Man wird dadurch unabhängig von der Schreibweise, was zwar auf Kosten eines gewissen Informationsverlustes geschieht, der aber durch die vorhandene Syntax (Substantive und Namen werden ja groß geschrieben) teilweise kompensiert wird.

2. Allzu oft sind Tastaturen (überwiegend bei älteren Systemen) nicht mit den *deutschen Sonderzeichen* ausgestattet. Hier weichen die Verfasser von Dokumenten üblicherweise auf Ersatzdarstellungen aus (ä → ae, ö → oe, ü → ue, ß → ss).

   Diesem Problem kann man entgehen, indem sowohl bei der Generierung als auch bei der Anfrage die gleichen Ersetzungen vorgenommen werden. Aber der hier eintretende Informationsverlust wird nicht wieder aufgehoben wie im Falle der Groß-/ Kleinschreibung.

   Eine andere Idee packt dieses Wissen möglicher Transformationen in die Matching-Komponente. Der Vergleich zweier Strings wird dadurch fehlertolerant gegenüber solchen Transformationen. Tritt jedoch eine solche Transformation im Bereich des zur Hashcodeberechnung relevanten Teilstrings auf, so kann dieses Verfahren Einträge nur dann finden, wenn sie diesbezüglich korrekt geschrieben sind.

3. Das Problem der Bindestrichkomposita ist eher der Segmentierung und Texterkennung zuzuordnen. Sie müssen Komposita erkennen und als Einheit zusammenfügen, bevor sie damit Anfragen an das Lexikon stellen [John93].

Kapitel 8

Testergebnisse

8.1 Generator

8.1.1 Laufzeiten des Generators

Die Laufzeiten des Generators haben in der momentanen Implementierung eine Zeitkomplexität von $O(n^2)$ (n = Anzahl der Lexikoneinträge). Da diese Zeiten für Lexikongrößen von etwa 100.000 Einträgen etwa 4 bis 5 Stunden beträgt, wollen wir uns hauptsächlich auf kleinere Wortschätze beschränken. Ein Lexikon mit 107.427 Einträgen wurde nur einmal mit einer statischen Sicht generiert. Die dazu benötigten Quellen werden aus bestehenden Wortlisten extrahiert. Sie haben dadurch zwar keinen Anspruch darauf, einen repäsentativen Querschnitt des Deutschen zu bilden, garantieren aber vorgegebene Längen, was die für die Vergleiche wesentlichere Eigenschaft ist. Außerdem sind sie gleichmäßig über die lexikographische Bandbreite gestreut (also von 'A' bis 'Z'), um keine unverhältnismäßig langen Kollisionsketten bei h_{begin} zu bekommen.

Generierungszeiten [sec]		
Lexikon-größe	ohne stat. Sicht	mit einer stat. Sicht
1.000	4,2	7,6
2.000	10,0	15,6
3.000	27,9	45,5
4.000	37,8	64,4
5.000	51,4	79,6
7.500	92,8	139,1
10.000	145,0	211,1
15.000	285,4	401,5
20.000	498,2	696,9
107.427	—	17.222,3

Tabelle 8.1: Laufzeiten des Generators

Tabelle 8.1 zeigt die Generierungszeiten für unterschiedlich große Wörterbücher – jeweils einmal ohne und einmal mit einer statischen Sicht.

Die quadratische Zeitkomplexität kommt daher, daß bei einem neuen Eintrag, der an eine Kette angefügt werden muß, das Ende der Kette durch sequentielles Ablaufen vom Beginn an gesucht werden muß, um dort die Adresse des neuen Wortes einzutragen. Sie wirkt sich jedoch erst ab einer gewissen Größe der Datenmengen aus, nämlich dann, wenn es zu Kollisionen kommt. Eine angestrebte Verbesserung des Generators sieht eine Verwaltung der Kettenenden vor, so daß das zeitraubende Durchlaufen entfällt und die Zeitkomplexität auf $O(n)$ reduziert wird.

8.1.2 Sekundärspeicherplatzbedarf

Die Größen einiger generierten Dateien lassen sich im vorraus abschätzen:

$$\begin{aligned} \text{Hashtabellendatei} &= \text{Hashtabellenlänge} * 4 \text{ Byte} \\ \text{Sichteinstiegsdatei} &= \#\text{Gruppen } (G_l) * \#\text{Ausprägungen} * 4 \text{ Byte} \end{aligned}$$

Da die Datensatzlängen im Hauptwörterbuch unterschiedlich sind, kann die Größe dieser Datei nicht berechnet werden. Jedoch können Abschätzungen gemacht werden. Beim Lexikon ohne Sichten ergibt sich die Dateigröße ungefähr aus der folgenden Gleichung:

Anzahl Einträge * 54 Byte (mittlere Datensatzlänge)

Für jede definierte statische Sicht im System vergrößert sich die Datei um:

Anzahl Einträge * 2 * 4 Byte (2 Hashfunktionen; 4 Byte Pointergröße)

Optionale Sichten belegen nur Platz (4 Byte) bei den darin enthaltenen Wörtern plus 1 Bit je Eintrag.

Wie in Abschnitt 4.4 über die Mikrostruktur zu entnehmen ist, sind einige Informationszellen ungenutzt. Dieser Raum kann durch eine *bedingte Übersetzung* [Kernighan78] des Quellprogramms eingespart werden. Somit reduziert sich die Größe des Hauptwörterbuches auf durchschnitlich:

Anzahl Einträge * 38 Byte (mittlere Datensatzlänge)

8.2 Laufzeitsystem

8.2.1 Vergleich mit *look*

Das Unix-Programm *look* führt eine binäre Suche auf einer lexikographisch sortierten Wortliste durch, die in einer Datei liegt. Dabei muß der Suchschlüssel vollständig bekannt sein. Da *look* für die Suche vollständiger Wörter das schnellste Unix-Tool ist, soll es zum Vergleich mit dem Lexikon herangezogen werden.

Da beide Systeme unterschiedliche Initialisierungsphasen haben und diese nicht den eigentlichen Suchvorgang betreffen, mußten sie von den Zeitmessungen ausgenommen werden. Im Falle des Lexikons ist dies kein Problem, da ein eigenes Modul, welches als "Frontend" zu dem Lexikonsystem dient, erst nach der Initialisierung mit der Zeitmessung beginnt. Komplizierter gestaltet sich die Elimination dieser Phase beim Dienstprogramm *look*, da uns hier keine Programmquellen zur Verfügung stehen und folglich immer nur die Gesamtlaufzeiten ermittelt werden können. Hier muß mit einem kleinen Trick gearbeitet werden: Wir lassen *look* auf einer leeren Datei suchen und messen die dazu benötigte Zeit (sie beträgt 43 msec). Dieser Zeitraum entspricht in etwa dem der Initialiserungsphase und wird bei den späteren Testläufen mit einer 160.015 Einträge umfassenden Datei von Ergebnis subtrahiert.

Ein anderes Problem sind die recht groben Zeiteinheiten (1/50 sec), in denen im Unix-Betriebssystem gemessen werden kann. Bei den hohen Geschwindigkeiten der Anfragen, können die Ergebnisse evtl. auf 0 gerundet werden, was nicht sehr aufschlußreich für die Ergebnisse ist. Aus diesem Grund haben wir alle Anfragen über *look* in einer Schleife einhundertmal in Folge ausgeführt und die Gesamtzeit entsprechend korrigiert. Beim Lexikonsystem wurden die Anfragen sogar zweihundertmal wiederholt, um noch genauere Ergebnisse zu erhalten.

Grundlage des Tests war eine Datei mit 160.015 lexikographisch sortierten Einträgen – sowohl als Basis für *look* als auch als Quelle für den Generator. Aus dieser Datei wurden in konstanten Abständen Wörter extrahiert und zur Suchanfrage benutzt. Die Tabellenlänge der hier relevanten Hashfunktion ($h_{complete}$) beträgt 99.991, was bei idealer Verteilung der Bilder der Funktion zu mittleren Kollisionskettenlängen K_l von etwa 1,6 Einträgen führt. Da wir aber nur eine Füllung von 79,58% haben, hat K_l in unserem Fall den Wert 2,01.

Die Werte in Tabelle 8.2 zeigen die reinen Suchzeiten für einige beispielhafte Anfrage.

Zugriffszeiten [msec]		
Eingabe	*look*	Lexikon
anlockend	4,0	0,167
begeistert	4,0	0,250
berufsforschung	3,0	0,333
dreieinhalbfache	4,0	1,667
einzelkabine	6,0	1,333
erschleicht	4,0	1,583
gebaeudesicherung	4,0	1,667
gemildert	5,0	1,333
herruehrten	2,0	0,167
hugenottenkriege	1,0	0,333
kreditiert	2,0	0,333
kulturfilm	3,0	0,083
oberverwaltungsgericht	5,0	1,583

Tabelle 8.2: Vergleich des Lexikons mit *look*

Man erkennt sehr deutlich, daß die Zeiten bei *look* immer bei etwa 7 (± 2) Millisekunden liegen, beim Zugriff über das Lexikonsystem jedoch enorm weit streuen. Dies rührt daher, daß einige der gesuchten Wörter weiter vorne, andere weiter hinten in den Kollisionsketten liegen und die Zahl der Leseoperationen entsprechend variiert.

8.2.2 Vergleich mit *agrep*

Das Program *agrep* ist eine Verbesserung der Programme der (Unix-)"grep"-Familie. Es ist im direkten Vergleich mit *grep*, *egrep* oder *fgrep* nicht nur schneller, sondern bietet zusätzlich zur Spezifikation *regulärer Ausdrücke* als Eingabe noch eine Fehlertoleranz an, indem beim Vergleich der Wörter mit der Eingabe ein *approximatives Matching* durchgeführt wird [Bunke90].

Aufgrund seiner kurzen Laufzeiten ist *agrep* der ideale Vergleichspartner zu dem Lexikon bei Anfragen mit unvollständig erkannten Wörtern [Wu92].

Bei den in Tabelle 8.3 dokumentierten Tests ist das gleiche Wörterbuch verwendet worden wie schon bei den Tests zu *look*. Die Eingaben wurden "von Hand" zu regulären Ausdrücken geformt, um unvollständig erkannte Wörter zu simulieren. Lediglich die Anzahl der Wiederholungen wurde bei *agrep* auf zehn und beim Lexikon auf zwanzig reduziert, da diese Anfragen ohnehin etwas länger andauern und Ungenauigkeiten bei der Zeitnahme nicht so sehr ins Gewicht fallen.

Zugriffszeiten [msec]			
Eingabe	*agrep*	Lexikon	Matches
?nlo[ec]k*d	720,0	87,5	1
b?[gqj][eco]ist*rt	770,0	245,8	1
be*orschun[gq]	830,0	115,8	1
drei??n[hb]al??ache	670,0	12,5	1
e[ij][rnm]zel[hkb]?bine	670,0	3,3	1
???chleicht	660,0	238,3	1
geb[ax]eu[da][ec]sicheru??	650,0	6,6	1
gemi*t	690,0	21,6	5
[bhk]erru*rten	740,0	625,8	1
*otten[hk]rieg?	1.170,0	258,3	1
??editi*	750,0	1.431,6	8
turfil	710,0	32.920,0	4
ober*ung[xs]ger[ji][ec]ht	1.550,0	15,8	1

Tabelle 8.3: Vergleich des Lexikons mit *agrep*

Aus den Werten wird sehr deutlich, daß die Laufzeit bei *agrep* relativ unabhängig von der Eingabe ist, wohingegen beim Lexikonsystem sehr starke Schwankungen der Zugriffszeiten zu verzeichnen sind. Diese beruhen auf der unterschiedlichen Anzahl der Alternativen die generiert werden und der damit verbundenen Zahl der Kollisionsketten die durchlaufen werden müssen.

Im allgemeinen ist das Lexikon schneller als *agrep*, insbesondere dann, wenn entweder Wortanfang oder Wortende wenige Alternativen zulassen. Jedoch bilden die Eingaben „??editi*" und „*turfil*" recht grasse Ausnahmen. Dies kommt daher, daß die zur Berechnung des Hashcodes relevante Information zerstört ist, und der Alternativengenerator 7.2.3 sehr viele Hashstrings liefert.

8.2.3 Beschleunigung durch statische Sichten

In diesem Abschnitt soll der Zeitgewinn beim Suchen nach unvollständigen Einträgen durch Angabe zusätzlichen Wissens in Form einer statischen Sicht gezeigt werden. Dazu ist ein Wörterbuch notwendig, in dem Sichten definiert sind.

Zur Ermittlung der Anfragezeiten dieses Tests wurde ein Lexikon mit 107.427 Einträgen verwendet. Bei den benötigten Quellen handelt es sich um Dateien mit männlichen, weiblichen und neutralen Substantiven, sowie Verben, Adjektiven und Funktionswörter. Das durch diese Unterteilung implizierte Wissen in den Quellwörterbüchern wurde zur Definition einer Sicht mit den Ausprägungen *SUBST*, *VERB*, *ADJEKT* und *FKTWORT* herangezogen.

Tabelle 8.4 zeigt die Anfragezeiten bei Benutzung der normalen Kollisionszeiger und die Anzahl der dabei gefundenen Einträge sowie Laufzeit und Zahl der Matches bei Spezifikation der Sichtausprägung *SUBST*.

Zugriffszeiten [msec]				
Eingabe	ohne Sicht	Matches	mit Sicht	Matches
[rn][eco][ce][hk][eo]nanl*	60,0	1	37,5	1
[bh][eco]si[ft]*[oc]sigk[eco]i[ft]	10,8	1	5,8	1
c[oa]mpu[tf]?r	7,5	1	5,0	1
d[eco]s[oc]?[oc]ris?[ft]ion	226,7	1	207,5	1
dipl[oc]m?[ft][eco]nsc[kh]r[eco]i[bh][ft]*[kh]	5,8	1	4,2	1
f[eco]ri[eco]n?us[ft]?u*[kh]	10,0	1	9,2	1
g[eco]sc[kh]ickli*k[eco]i[ft]	36,7	2	16,7	1
gul?sc[kh]k?n[oc]n[eco]	9,2	1	9,2	1
h?ndf[eco]g[eco]r	85,8	1	90,8	1
hin[ft][eco]rl?ss[eco]nsc[kh]?f[ft]	15,8	1	8,3	1
i[oc]n[eco]nw?nd[eco]rung	399,2	1	260,0	1
kaiser*	12,5	28	6,7	18
kr?nk[eco]ng[eco]sc[kh]ic[kh][ft][eco]	70,0	1	65,0	1
l[eco]uc[kh][ft]r?k[eco][ft][eco]	16,7	1	6,7	1
me[eo]r[ec]sl[eco]uc[kh][ft][eco]n	8,3	1	7,5	1
m?nito*	83,3	2	47,5	2
n?[bh][eco]ls[ft]r?ng	492,5	1	463,3	1
s[eco]l[bh]s[ft]kos[ft][eco]ns[eco]nkung	75,8	1	64,2	1
s[ft]?d[ft]s[ft]r[eco]ic[kh][eco]r	595,0	1	410,8	1
ur[ft]i[eco]rc[kh][eco]n	8,3	1	5,8	1
v[eco]r?rsc[kh]ung	181,7	1	155,8	1
z[eco]n[ft]im[eco][ft][eco]rm?ss	12,5	1	7,5	1

Tabelle 8.4: Vergleich des Zugriffs mit und ohne Sicht

Die Werte lassen eine Zeitersparnis von 30–40% erkennen. Ob der zusätzlich benötigte Speicherplatz diesen Vorteil rechtfertigt, muß in Abhängigkeit der Anwendung individu-

ell entschieden werden. Falls das Lexikonsystem eine zeitkritische Komponente darstellt (wie im Falle des ALV-Dokumentanalysesystems), ist die Definition einer solchen Sicht sicherlich zu vertreten [14].

Weiterhin fällt auf, daß durch die Spezifikation einer Sicht der Suchraum eingeschränkt wird, was sich in einer reduzierten Anzahl „passender" Wörter niederschlägt.

[14] Vorraussetzung ist auch hier wieder die Verfügbarkeit des entsprechenden Wissens, ohne das die Sichten nicht genutzt werden können.

8.2.4 Binäre Trigramme

Im Vorfeld der Implementierung wurden auch einige probabilistische Ansätze zur Unterstüzung des Lexikonsystems auf ihre Verwendbarkeit untersucht. Darunter waren auch die binären Trigramme. Die zugrundeliegende Idee bestand in der Realisierung eines hybriden Verfahrens aus Wahrscheinlichkeitsmodell und Lexikonabgleich, um die Schnelligkeit des probabilistischen Modells in unserem System zur Falsifizierung falscher Hypothesen zu nutzen.

Zur Ermittlung der Trigramme wurden einmal die laut Meier 8000 häufigsten Wörter [Meier78] und die 160.015 Wörter der Datei `worteCELEX` [CELEX91] herangezogen. Dabei kamen bei Meier nur 3.248 von 27.000 (12%) theoretisch möglichen Trigrammen vor – bei dem wesentlich größeren Wortschatz aus `worteCELEX` wurden hingegen 6.234 (23%) Trigramme als legal ermittelt. Die anschließend gezielt falsch geschriebenen Wörter[15] konnten zum Teil nicht falsifiziert werden. Insbesondere beim größeren Wortschatz als Basis wurden fast alle fehlerhaften Eingaben akzeptiert.

Tabelle 8.5 zeigt einige Eingaben und die Ergebnisse der Trigramm-Vergleiche, sowohl mit 8.000 als auch mit 160.015 Wörtern als Basislexikon. Bei Zurückweisungen wird das erste Trigramm des Wortes angegeben, welches nicht legal ist, ansonsten der Vermerk *OK*.

Tests mit binären Trigrammen			
korrekte Eingabe	verfälschte Eingabe	8.000-er Lexikon	160.015-er Lexikon
folgen	foglen	fog	fog
lesen	leesn	ees	*OK*
geboten	gebohten	boh	*OK*
essen	esen	*OK*	*OK*
fremder	fremnder	mnd	mnd
angebracht	angebraht	*OK*	*OK*
literarischen	litrerarischen	*OK*	*OK*
operationen	operatiomen	iom	*OK*
verwalter	verwater	wat	*OK*
polnischen	polischen	*OK*	*OK*
verknüpft	verknuepft	knu	*OK*
dunkler	dunkeler	*OK*	*OK*
hunderte	hundrete	*OK*	*OK*
neuzeit	neuzzeit	uzz	*OK*
schauspiele	schauspile	pil	*OK*

Tabelle 8.5: Binäre Trigramme zur Falsifizierung von Hypothesen

Man erkennt, daß bei größeren Basiswörterbüchern deutlich mehr Trigramme als legal eingestuft werden und somit auch mehr fehlerhafte Eingaben von dem Test als gültiges Wort akzeptiert werden.

[15] Die Testeingaben wurden aus dem zur Generierung gültiger Trigramme vorgegebenen Lexikon algorithmisch ausgewählt und anschließend von Hand „verfälscht".

Konnte man bei der Basis von 8.000 Wörtern noch 9 der 15 Eingaben als falsch überführen, so waren es beim 160.015-er Lexikon nur noch 2. Durch diese unbefriedigenden Resultate wurde die Integration dieses Verfahrens der binären Trigramme wieder verworfen.

Kapitel 9

Ausblick

Diese Diplomarbeit präsentierte einen völlig neuen Ansatz der *virtuellen Partitionierung* großer Wörterbücher. Doch im Laufe der Konzeption und Implementierung sind auch an diesem System mögliche Verbesserungen und wünschenswerte neue Punkte erkennbar geworden.

- Die Zeitkomplexität des Generators läßt sich von gegenwärtig $O(n^2)$ auf $O(n)$ bei n Lexikoneinträgen verbessern. Dadurch können die Generierungszeiten erheblich beschleunigt werden (siehe Abschnitt 8.1.1). Der hierzu notwendige Aufwand ist recht gering, da nur die Adressen der jeweils letzten Einträge einer jeden Kollisionskette verwaltet werden müssen.

- Um neue Wörter in ein bestehendes Lexikon aufzunehmen, muß bisher ein komplett neuer, zeitaufwendiger Generierungsprozeß gestartet werden. Dies kann auf relativ einfache Weise geändert werden. Eine neue Startoption des Generators läßt diesen auf einem existierenden Lexikon arbeiten. Dazu wird das System-Profile gelesen und die entsprechenden Dateien geöffnet bzw. die Hashtabellen und Profiles eingelesen. Danach befindet sich der Generator im gleichen Zustand, wie schon bei der initialen Generierung und kann anschließend wie bisher neue Einträge im Hauptwörterbuch einfügen.

- Die Matching-Komponente kann momentan nur exakt matchende Einträge feststellen. Hier soll eine fehlertolerante Vergleichfunktion die alte ersetzen. Dadurch können Wörter auch dann gefunden werden, wenn sie im Dokument fehlerhaft geschrieben oder von der Texterkennung falsch erkannt worden sind. Als das hierfür benötigte Ähnlichkeitsmaß soll die gewichtete Levenshteindistanz [Bunke90] verwendet werden.

- Als weitere Neuerung soll es dem Benutzer ermöglicht werden, in ein bestehendes Lexikon Verbindungen (sogenannte *Links*) zwischen den Einträgen zu definieren, um sie später ablaufen zu können. Eine Verwandschaft zu *Hypertext-Systemen* wird hier deutlich.

- Schließlich ist eine Referenz der Einträge auf externe Informationen (Dateien) denkbar. Die dort verborgene Information und ihre Struktur muß dem Lexikon nicht

bekannt sein. Allein das anfragende System muß in der Lage sein, diese Daten zu interpretieren. Ein Beispiel hiefür könnte eine Graphik-Datei sein, die auf dem Monitor ein Bild des angefragten Objekts darstellt. Ebenso sind auch bewegte Bilder möglich. Natürlich kann es sich bei den referenzierten Dateien auch um ausführbare Programme handeln, die gestartet werden. Die Palette der Anwendungen ist recht umfangreich.

Anhang A

Lisp-Schnittstelle

A.1 Beschreibung der Funktionen

Das folgende Kapitel soll die Lisp-Funktionen vorstellen, mit denen auf das Lexikon zugegriffen werden kann. Zu diesem Zweck wird die Datei mit den benötigten Deklarationen nach um nach erläutert und das Ein-/Ausgabeverhalten der Funktionen beschrieben.

Hilfsfunktionen des Foreign-Function-Interface

```
(defun malloc-foreign-string (str)
  (check-type str string)
  (let ((f-str (malloc-foreign-pointer :type
                '(:pointer (:array :character (,(1+ (length str))))))))
    (setf (foreign-string-value f-str) str)
    (setf (foreign-pointer-type f-str) '(:pointer :character))
    f-str))
```

Diese Funktion allokiert einen Speicherbereich außerhalb des Einflußbereiches des Garbage-Collectors und konvertiert den String „str" aus der Lisp- in die C-Darstellung in diesen Bereich hinein (siehe Abbildung 7.1). Die eigentliche Allokation von Speicher übernimmt die Funktion *malloc-foreign-pointer*, *foreign-string-value* konvertiert den String und *foreign-pointer-type* deklariert ihr Argument als einen Zeiger vom Typ Charakter.

Werden C-Funktionen aufgerufen, die einen String als Argument erwarten, so wird zunächst diese Funktion aufgerufen und ihr Ergebnis an eine lokale Variable gebunden. Mit dieser lokalen Variable als Argument kann nun die eigentliche C-Funktion aufgerufen werden. Nach Beendigung dieses Aufrufs muß der Speicherbereich wieder befreit werden. Dies geschieht mit der Funktion *free-foreign-pointer*.

A.1.1 Systemspezifische Funktionen

Initialisierung des Lexikons

```
(def-foreign-function (libc-init
                        (:return-type :int)
                        (:name "_UD_init")
                        (:language :c))
        (name (:pointer :character)))

(defun init-dict (name)
  (check-type name string)
  (let ((f-name (malloc-foreign-string name)))
     (prog1 (libc-init f-name)
        (free-foreign-pointer f-name))))
```

Der erste Ausdruck macht dem Lisp-System eine „fremde" Funktion bekannt. Ihr Name in Lisp lautet *libc-init*; ihr Rückgabewert ist vom Typ integer; der externe Name ist *_UD_init*; die Sprache ist C; sie benötigt ein Argument vom Typ 'Character-Pointer'.

Nachdem diese C-Funktion vereinbart wurde, kann sie zur Definition weiterer Lisp-Funktionen herangezogen werden. Im obigen Fall ist es *init-dict*, die ihrerseits einen Aufruf von *libc-init* beinhaltet. Sie initialisiert das Lexikon-System, was unter Anderem das Einlesen der Profiles, laden der Hashtabellen, Öffnen der Sichteinstiegsdateien und des virtuell unterteilten Haupwörterbuches beinhaltet. Das benötigte Argument ist der Pfadname des System-Profiles, welches bei der Generierung erzeugt wurde. Der Returnwert ist 0 wenn keine Fehler auftreten, ansonsten 1.

Verlassen des Lexikons

```
(def-foreign-function (libc-exit
                        (:return-type :int)
                        (:name "_UD_exit")
                        (:language :c)))

(defun exit-dict () (= 0 (libc-exit)))
```

Auch hier wird zunächst die externe C-Funktion *_UD_exit* bekannt gemacht. Sie wird zur Definition von *exit-dict*, einer Funktion ohne Argumente, benötigt. Dabei werden alle Dateien des Lexikonsystems geschlossen und das temporäre Lexikon auf eine Datei geschrieben. Um nicht versehentlich ein temporäres Lexikon eines früheren Programmlaufs zu löschen, wird der Name dieser Datei aus dem String `TempDict`, dem aktuellen Datum und der Uhrzeit gebildet, ein möglicher Name wäre `TempDict.30.Apr_11:59`. Der Returnwert ist *t* falls keine Fehler auftreten, ansonsten *nil*.

Einlesen des temporären Lexikons

```
(def-foreign-function (libc-read-temporal-dict
                        (:return-type :int)
                        (:name "_UD_read_temporal_dict")
                        (:language :c))
        (name (:pointer :character)))

(defun read-temporal-dict (name)
  (check-type name string)
  (let ((f-name (malloc-foreign-string name)))
     (prog1 (= 0 (libc-read-temporal-dict f-name))
        (free-foreign-pointer f-name))))
```

Mit *read-temporal-dict* kann ein vorhandenes temporäres Lexikon L_{temp} zum momentanen hinzugeladen werden. Dies kann entweder ein generiertes L_{temp} eines früheren Programmlaufs oder eine „von Hand" erzeugte Liste neuer Wörter sein. Auf diese Weise können mehrere Dateien nachgeladen werden. Nach Systeminitialisierung ist L_{temp} leer; es füllt sich nur mittels dieser Funktion oder bei Anfragen nach vollständig erkannten Wörtern, die nicht im Hauptwörterbuch vorhanden sind.

A.1.2 Suche nach Einträgen

Zugriff über die Lesart

```
(def-foreign-function (libc-getword-prepare
                        (:return-type :int)
                        (:name "_UD_find_first")
                        (:language :c))
        (name (:pointer :character)))

(defun getwordprepare (name)
  (check-type name string)
  (let ((f-name (malloc-foreign-string name)))
     (prog1 (libc-getword-prepare f-name)
        (free-foreign-pointer f-name))))
```

Diese Funktion teilt dem Lexikon-System mit, daß wir über einen regulären Ausdruck als Eingabe suchen. Durch mindestens ein Leerzeichen getrennt von dem regulären Ausdruck kann der Name einer Sicht-Ausprägung angegeben werden. Durch diese optionale Angabe wird im Falle einer unvollständigen Eingabe die Kollisionskette der entsprechenden statischen Sicht gewählt und eine verkürzte Anfragedauer erzielt. Die Rückgabewerte sind: 0 falls kein Eintrag gefunden wurde, 1 falls ein Eintrag im Haupt-Wörterbuch gefunden wurde und 2 falls ein vollständig erkanntes Wort nur im temporären Wörterbuch vorliegt.

```
(def-foreign-function (libc-getword-next
                        (:return-type (:pointer :character))
                        (:name "_UD_find_next")
                        (:language :c)))

(defun getwordnext ()
  (foreign-string-value (libc-getword-next)))
```

Wohingegen die letzte Funktion das System auf eine Suche mittels der Lesart vorbereitete, liefert *getwordnext* (ohne Parameter) das jeweils nächste Wort, welches mit dem Eingabestring aus *getwordprepare* matcht und auch das Filtersystem passiert. Falls die letzte der möglichen Kollisionsketten durchlaufen ist, liefert sie *nil* zurück.

```
(def-foreign-function (libc-getword-cardinality
                        (:return-type :int)
                        (:name "_UD_pattern_cardinality")
                        (:language :c))
        (name (:pointer :character)))

(defun getword-cardinality (name)
  (check-type name string)
  (let ((f-name (malloc-foreign-string name)))
     (prog1 (libc-getword-cardinality f-name)
        (free-foreign-pointer f-name))))
```

getword-cardinality sucht ebenfalls nach Wörtern, die auf den Eingabestring matchen. Ihre Eingabe ist identisch mit der von *getwordprepare*, jedoch durchläuft sie sogleich die Ketten und liefert als Ergebnis die Anzahl gefundener Wörter zurück. Im Anschluß an diese Funktion kann *getwordnext* nicht sinnvoll aufgerufen werden.

Man könnte auch *getwordprepare* aufrufen und die Anzahl der erfolgreichen Aufrufe von *getwordnext* zählen. Jedoch würde in diesem Fall jedes Wort durch die Lisp-C-Schnittstelle geschleust werden, was die Anfragedauer erheblich verlängert.

```
(defun getword (name)
  (let ((x t)
        l
        (rc (getwordprepare name)))
    (cond ((= 1 rc)
           (do* ()
                ((null x) l)
                (setq x (getwordnext))
                (if x (setq l (nconc l (list x)))))
           )
           ((= 2 rc)
            (format t "~&Wort ~:A ist im temporaeren Lexikon." name) t)
           (t (format t "~&Wort ~:A ist in keinem Lexikon." name) nil))
  ))
```

getword ist eine beispielhafte Schleife, die alle Wörter, die auf eine Eingabe matchen, in einer Liste sammelt. Wurde ein vollständig erkanntes Wort nur im temporären Lexikon gefunden, so wird eine entsprechende Ausgabe ausgegeben. Diese Ausgabe wird von der Lisp-Funktion selbst erzeugt.

Der Anwender des Lexikon-Systems hat die Möglichkeit, auf dieser Funktion als Gerüst aufbauend, eigene Schleifen zu realisieren, die zusätzlich zu dem zurückgelieferten Einträgen noch die Syntax, Sichtzugehörigkeit oder Ähnliches abfragen und an die Wörter z.B. in Form von Listen zu binden.

Zugriff über optionalen Sichten

```
(def-foreign-function (libc-getview-prepare
                        (:return-type :int)
                        (:name "_UD_prepare_view")
                        (:language :c))
        (name (:pointer :character)))

(defun getviewprepare (name)
  (check-type name string)
  (let ((f-name (malloc-foreign-string name)))
     (prog1 (= 1 (libc-getview-prepare f-name))
        (free-foreign-pointer f-name))))
```

Diese Funktion ist das Pendant zu *getwordprepare*, jedoch leitet sie eine Suche nach den Elementen einer optionalen Sicht ein. Als Argument verlangt sie den Namen einer Sicht. Sie liefert *t*, falls das Argument ein gültiger Name einer optionalen Sicht ist, sonst *nil*.

```
(def-foreign-function (libc-getview-next
                        (:return-type (:pointer :character))
                        (:name "_UD_next_view_entry")
                        (:language :c)))

(defun getviewnext ()
  (foreign-string-value (libc-getview-next)))
```

Dies ist dementsprechend die Funktion, die sukzessive alle Elemente der Sicht zurückliefert. Dabei werden die Filter beachtet, d.h. falls mindestens einer der Filter aktiv ist, so muß auch einer passiert werden.

```
(def-foreign-function (libc-getview-cardinality
                        (:return-type :int)
                        (:name "_UD_view_cardinality")
                        (:language :c))
        (name (:pointer :character)))

(defun getview-cardinality (name)
  (check-type name string)
  (let ((f-name (malloc-foreign-string name)))
     (prog1 (libc-getview-cardinality f-name)
        (free-foreign-pointer f-name))))
```

Hiermit kann die Mächtigkeit einer optionalen Sicht unter Berücksichtigung evtl. aktiver Filter ermittelt werden. Die Berechnung vollzieht sich wie bei *getword-cardinality* auf der C-Seite. Es wird lediglich das Ergebnis (die Mächtigkeit der Menge) über die Schnittstelle zurückgegeben.

```
(defun getview (name)
  (let ((x t)
        l)
    (when (getviewprepare name)
          (do* ()
               ((null x) l)
               (setq x (getviewnext))
               (if x (setq l (nconc l (list x)))))
          )))
```

Auch hier handelt es sich um eine Schleife, die alle Wörter der optionalen Sicht in einer Liste einsammelt, selbstverständlich unter Berücksichtigung evtl. aktiver Filter.

A.1.3 Abfragen von Informationen

In diesem Abschnitt beschreiben wir diejenigen Funktionen, die Informationen zu einem gefundenen Wort liefern. Ihnen muß eine der Funktionen *getwordnext* oder *getviewnext* vorangehen. Der Benutzer erhält durch diese Funktionen die Zeichenkette des Wortes, an dem der lexikoninterne Cursor gerade steht. Die im folgenden erklärten Funktionen liefern Informationen zu gerade diesem aktuellen Wort.

```
(def-foreign-function (libc-get-next-phrase
                        (:return-type (:pointer :character))
                        (:name "_UD_next_phrase")
                        (:language :c)))

(defun get-next-phrase ()
  (foreign-string-value (libc-get-next-phrase)))
```

get-next-phrase liefert sukzessive die Phrasen[16] zurück, die das Wort enthalten. Falls keine Phrase (mehr) das Wort referenziert, so liefert die Funktion *nil*. Die Phrasen selbst stehen im Phrasen-Profile, welches bereits zur Generierung angegeben werden muß.

```
(def-foreign-function (libc-get-next-view-of-word
                        (:return-type (:pointer :character))
                        (:name "_UD_next_view_of_word")
                        (:language :c)))

(defun get-next-view ()
  (foreign-string-value (libc-get-next-view-of-word)))
```

get-next-view gibt nach und nach die Namen der optionalen Sichten zurück, denen der Eintrag angehört; oder *nil* wenn er keiner (weiteren) Sicht angehört.

```
(defun getwordviews ()
  (let ((x t)
        l)
    (do* ()
         ((null x) l)
      (setq x (get-next-view))
      (if x (setq l (nconc l (list x)))))
    ))
```

Diese Funktion stellt wieder eine Schleife dar, die auf einer anderen Funktion aufbaut. Im jetzigen Falle werden alle Sichten eines Wortes in einer Liste gesammelt.

[16] z.B. Redewendungen in Geschäftsbriefen wie „mit freundlichen Grüßen", „sehr geehrte Damen und Herren" ....

```
(def-foreign-function (libc-get-next-syntax-of-word
                        (:return-type (:pointer :character))
                        (:name "_UD_next_syntax_of_word")
                        (:language :c)))

(defun get-next-syntax ()
  (foreign-string-value (libc-get-next-syntax-of-word)))
```

Hier werden die Syntax-Token zurückgeliefert, die zur Generierungsphase dem Wort zugeordnet wurden.

```
(defun getwordsyntax ()
  (let ((x t)
        l)
    (do* ()
         ((null x) l)
      (setq x (get-next-syntax))
      (if x (setq l (nconc l (list x)))))
    ))
```

Diese Funktion sammelt die Syntax-Token des Eintrags in einer Liste. Man erkennt, daß sie analog zu *getwordviews* aufgebaut ist, was durch das gleiche Ein-/ Ausgabeverhalten der Basisfunktionen erst möglich wird.

A.1.4 Manipulation der Filter

Die folgenden Funktionen dienen dem Setzen, Löschen und Negieren einzelner Filter. Dabei muß in der Regel einer der Filter durch einen Index spezifiziert werden. Die Anzahl der zur Verfügung stehenden Filter wird durch die Konstante MAXFILTER in der Datei `constant.h` festgelegt. Ihr momentaner Wert ist 8. Die Indizies bewegen sich somit im Bereich 0 bis 7.

```
(def-foreign-function (libc-set-filter
                        (:return-type :int)
                        (:name "_MD_set_filter")
                        (:language :c))
        (i :signed-32bit)
        (view (:pointer :character)))

(defun set-filter (i view)
  (check-type view string)
  (check-type i integer)
  (let ((f-view (malloc-foreign-string view)))
   (prog1 (= 1 (libc-set-filter i f-view))
   (free-foreign-pointer f-view))))
```

Zum Setzen eines Filters mit *set-filter* müssen sein Index und der Name einer optionalen Sicht angegeben werden. Im Erfolgsfalle liefert die Funktion *t* zurück. Bei Indexüberschreitung oder falschem Sichtnamen liefert sie *nil*. Wird ein Filter auf eine Sicht gesetzt, so läßt er nur diejenigen Wörter passieren, die in entsprechender Sicht enthalten sind. Ein Filter kann mehrfach gesetzt werden, was eine konjunktive Verknüpfung der Filterbedingungen bewirkt.

```
(def-foreign-function (libc-clear-filter
                        (:return-type :int)
                        (:name "_MD_clear_filter")
                        (:language :c))
        (i :signed-32bit))

(defun clear-filter (i)
  (check-type i integer)
  (= 1 (libc-clear-filter i)))
```

Diese Funktion versetzt den angegebenen Filter wieder in seinen Initialzustand. Er gilt danach als inaktiv, was durch seine Statusvariable angezeigt wird.

```
(def-foreign-function (libc-reset-filter
                       (:return-type :int)
                       (:name "_MD_reset_filter")
                       (:language :c)))

(defun reset-filter ()
  (= 1 (libc-reset-filter)))
```

Allein bei dieser Funktion muß nicht der Index eines Filters angegeben werden. Die Funktion wird gänzlich ohne Parameter aufgerufen und bewirkt ein Löschen aller Filter. Nach dem Aufruf von *reset-filter* sind sämtliche Filter inaktiv, was dem initialen Zustand entspricht.

```
(def-foreign-function (libc-negate-filter
                       (:return-type :int)
                       (:name "_MD_negate_filter")
                       (:language :c))
        (i :signed-32bit))

(defun negate-filter (i)
  (check-type i integer)
  (= 1 (libc-negate-filter i)))
```

Mit *negate-filter* wird die Statusvariable eines Filters auf *negativ_use* gesetzt. Er läßt daraufhin nur diejenigen Wörter passieren, die in keiner der für diesen Filter gesetzten Sichten enthalten sind.

```
(def-foreign-function (libc-positive-filter
                       (:return-type :int)
                       (:name "_MD_positive_filter")
                       (:language :c))
        (i :signed-32bit))

(defun positive-filter (i)
  (check-type i integer)
  (= 1 (libc-positive-filter i)))
```

Durch diese Funktion kann die Statusvariable eines Filters auf *normal_use* zurückgesetzt werden. Er arbeitet daraufhin wieder auf seine ursprüngliche Weise.

A.2 Zusammenfassung

init-dict Initialisierung des Lexikon-Systems. Parameter ist der Pfadname des System-Profiles. Returnwert ist 0, wenn keine Fehler auftreten, ansonsten 1.

exit-dict Alle Dateien des Lexikonsystems werden geschlossen und das temporäre Lexikon auf eine Datei gerettet. Returnwert ist *t* falls keine Fehler auftreten, ansonsten *nil*.

read-temporal-dict lädt ein vorhandenes temporäres Lexikon zum momentanen hinzu. Returnwert ist *t* falls keine Fehler auftreten, ansonsten *nil*.

getwordprepare leitet eine Suche über einen regulären Ausdruck als Eingabeparameter ein. Durch mindestens ein Leerzeichen getrennt von dem regulären Ausdruck kann der Name einer Sicht-Ausprägung angegeben werden. Returnwerte sind: 0 falls kein Eintrag gefunden wurde, 1 falls ein Eintrag im Haupt-Wörterbuch gefunden wurde und 2 falls ein vollständig erkanntes Wort nur im temporären Wörterbuch vorliegt.

getwordnext liefert das jeweils nächste Wort zurück, welches mit dem Eingabestring aus *getwordprepare* matcht und auch das Filtersystem passiert. Falls keine weiteren Wörter matchen, liefert die Funktion *nil*.

getword-cardinality zählt die Anzahl der Einträge, die auf den Eingabestring matchen und liefert das Ergebnis zurück. Ihre Eingabe ist identisch mit der von *getwordprepare*.

getword sammelt alle Wörter, die auf eine Eingabe matchen, in einer Liste. Wurde ein vollständig erkanntes Wort nur im temporären Lexikon gefunden, so wird eine entsprechende Ausgabe erzeugt.

getviewprepare leitet eine Suche nach den Elementen einer optionalen Sicht (dem Argument) ein. Returnwert ist *t*, falls das Argument ein gültiger Name einer optionalen Sicht ist, sonst *nil*.

getviewnext liefert sukzessive alle Elemente der Sicht zurück, die auch mindestens einen der gesetzten Filter passieren. Falls das letzte Wort der Sicht bereits übertragen wurde, liefert die Funktion *nil*.

getview-cardinality ermittelt die Mächtigkeit einer optionalen Sicht (dem Argument) unter Berücksichtigung evtl. aktiver Filter und liefert das Ergebnis zurück.

getview ist eine Schleife, die alle Wörter der optionalen Sicht (dem Argument) unter Berücksichtigung evtl. aktiver Filter in einer Liste einsammelt.

get-next-phrase liefert sukzessive alle Phrasen zurück, die das aktuelle Wort enthalten. Falls keine Phrase (mehr) das Wort referenziert, so liefert sie *nil*.

get-next-view gibt nach um nach die Namen der optionalen Sichten zurück, denen der Eintrag angehört (oder *nil*, wenn er keiner (weiteren) Sicht angehört).

getwordviews sammelt alle Sichten eines Wortes in einer Liste.

get-next-syntax liefert sukzessive alle Syntax-Token zurück, die zur Generierungsphase dem Wort zugeordnet wurden.

getwordsyntax sammelt die Syntax-Token des Eintrags in einer Liste.

set-filter setzt den durch das erste Argument spezifizierten Filter auf die durch das zweite Argument spezifizierte optionale Sicht. Im Erfolgsfalle liefert die Funktion *t* zurück. Bei Indexüberschreitung oder bei falschem Sichtnamen liefert sie *nil*.

clear-filter versetzt den angegebenen Filter wieder in seinen Initialzustand. Er gilt danach als inaktiv.

reset-filter bewirkt ein Löschen aller Filter. Nach dem Aufruf von *reset-filter* sind sämtliche Filter inaktiv, was dem initialen Zustand entspricht.

negate-filter setzt die Statusvariable eines Filters auf *negativ_use*. Er läßt daraufhin nur diejenigen Wörter passieren, die nicht in einer der für diesen Filter gesetzten Sichten enthalten sind.

positive-filter setzt die Statusvariable eines Filters auf *normal_use* zurück. Dieser arbeitet daraufhin wieder auf seine ursprüngliche Weise.

Anhang B

Beispielhafte Erzeugung eines Wörterbuches

In diesem Abschnitt soll der Leser mit dem Generator des Lexikons vertraut gemacht werden. Zu diesem Zweck verwenden wir konkrete Programm- sowie Dateinamen der momentanen Implementierung, die es dem Leser erleichtern, das Beispiel nachzuvollziehen. Wie bereits erwähnt, benötigen wir Wörterbuch-Quellen und Profiles. Der Name des Generators lautet **makedict**; er hat zu allen Dateinamen interne Defaultwerte, die durch Parameter beim Aufruf überschrieben werden können. Seine Aufrufstruktur wird durch Eingabe des Befehls

```
makedict h
   Gebrauch : makedict [Optionen]
        -q  WB-Quelldateien (bis zu 32 Quelldateien)
        -o  WB-Zieldatei
        -s  WB-Sichteneinstiegsdatei(en)
        -h  Hashtabellen-Datei(en)
        -lx Hashtabellen-Laenge fuer Funktion Nr.x  x=(1|2|3|x)
        -p  Phrasendatei
        -x  Syntax-Profile
        -y  Statische Sichten Definitions-Datei
        -v  Optionale Sichten Definitions-Datei
```

ausgegeben. Da in diesem Beispiel die Defaultwerte verwendet werden sollen, muß man sich zunächst nicht mit den Parametern des Programms zu befassen. Jedoch wird ein Unterverzeichnis mit dem Namen `dict` benötigt, das von unserem Homedirectory aus durch die folgenden Befehle angelegt wird:

```
cd ~
mkdir dict
```

Das Programm *„makedict"* sollte im Suchpfad liegen. Falls dies nicht der Fall ist, so kann es im Verzeichnis `~alv/src/dictionary` gefunden werden. Jetzt werden einige exemplarische Profiles angelegt. Dazu wechseln wir in das Verzeichnis `dict`.

```
cd dict
```

B.1 Erzeugen der Profiles

B.1.1 Das Syntax-Profile

Das erste Profile soll das Syntax-Profile mit dem Namen **`syntax.pro`** sein. Hierzu erzeugt man die folgende Datei mit einem verfügbaren ASCII-Editor wie z.B. *vi*.

```
SYNTAX SUBST  00000001 00000111
SYNTAX VERB   00000010 00000111
SYNTAX ADJEKT 00000011 00000111
SYNTAX FKTWRT 00000100 00000111
```

Diese Datei speichert man unter dem Namen **`syntax.pro`** im Verzeichnis `dict` ab. Sie ist sehr klein und ist auch nur in der Lage, die vier verschiedenen Wortkategorien *Substantive*, *Verben*, *Adjektive* und *Funktionswörter* abzuspeichern. Die Zeichenkette **SYNTAX** zeigt an, daß diese Zeile ein neues Syntax-Token definiert. Alle Zeilen, die nicht mit diesem String beginnen, können als Kommentarzeilen benutzt werden. Das zweite Wort in jeder Zeile des Profiles kennzeichnet das Token, welches in den Quellwörterbüchern einem Wort die entsprechende Syntax zuordnet. Die dritte und vierte Zeichenkette sind ein Muster und eine Maske, zwei Zahlen in Binärdarstellung. Die Maske ist für alle Token der Wortkategorie gleich, das Muster hingegen dient der Unterscheidung der einzelnen Syntax-Token dieser Kategorie. Führende Nullen müssen nicht angegeben werden (geschah nur aus Gründen der Anschaulichkeit).

Da die Variable für die Wortsyntax ein 32-Bit-Integer ist, können Muster und Maske jeweils 32 Bit lang werden.

B.1.2 Das Profile der statischen Sichten

Als zweites erzeugen wir eine Datei mit dem Namen **`sichten.pro`** und folgendem Inhalt:

```
SICHT Kategorie SUBST VERB ADJEKT FKTWRT
```

Wie schon in der Datei **`syntax.pro`** dient hier das Wort **SICHT** am Anfang einer Zeile dazu, die relevanten Zeilen von Kommentarzeilen zu unterscheiden. Das zweite Wort `Kategorie` gibt der hier definierten statischen Sicht ihren Namen. Die restlichen Wörter kennzeichnen die Ausprägungen dieser Sicht. Es muß gewährleistet sein, daß kein Eintrag in den Quellwörterbüchern zwei dieser Ausprägungs-Token enthält. Ansonsten gibt der Generator eine Fehlermeldung aus und fügt den Eintrag nur in eine Verweiskette ein.

Es fällt auf, daß wir als Ausprägungs-Token die gleichen Namen gewählt haben wie schon im Syntax-Profile. Dies ist keinesfalls eine Bedingung, sondern vielmehr die Ausnutzung einer Freiheit, die dem Designer der Profiles zusteht. Weiß der Benutzer beispielsweise, daß alle Substantive in den Quellwörterbüchern durch das Token SUBST gekennzeichnet sind, so kann er das sowohl zur Syntax-Codierung als auch zur Bildung einer Sicht heranziehen. Es könnte sogar auch im nächsten Profile der optionalen Sichten zur Definition einer optionalen Sicht herangezogen werden.

B.1.3 Das Profile der optionalen Sichten

Hierzu erzeugen wir eine Datei mit dem Namen **logview.pro** und folgendem Inhalt:

```
NUMBER_VIEWS  5
OPT_VIEW Hardware      SUN Macintosh IBM-PC
OPT_VIEW OP_Systeme    Unix Finder DOS
OPT_VIEW Com_Sprache   C Lisp Prolog Pascal Modula
OPT_VIEW Comp_words    ^Hardware ^OP_Systeme ^Com_Sprache
OPT_VIEW Substantive   >SUBST ^Comp_words
```

Die mit **OPT_VIEW** beginnenden Zeilen definieren eine optionale Sicht. Vor der ersten Zeile dieser Art muß eine mit **NUMBER_VIEWS** beginnende Zeile stehen. Die Zahl dahinter gibt an, wieviele verschiedene optionale Sichten definiert werden. Alle Zeilen, die mit anderen Token beginnen, sind Kommentarzeilen.

Das zweite Token der Sichtdefinitionen gibt den Namen der optionalen Sicht an. Groß-/ Kleinschreibung sind dabei relevant. Dies sind die Namen, die zur Laufzeit zur Manipulation der Filter und zu Abfragen optionaler Sichten angegeben werden können. Alle danach folgenden Wörter definieren zur Sicht gehörige Einträge. Wir haben alle drei möglichen unterschiedlichen Arten der Spezifikation optionaler Sichten in unserem Beispiel-Profile vereint. Dies wären:

1. Die **direkte Angabe** von Zeichenketten, die Wörter der Sicht darstellen. Dies ist in unserem Profile in den ersten drei Zeilen der Sichtdefinition der Fall.

2. Definition einer Sicht durch Angabe einer bereits definierten optionalen Sicht. Dies ist in Zeile vier der Fall. Das Sonderzeichen '^' vor den Wörtern bezieht sich auf Namen bereits definierter optionaler Sichten (in diesem Fall ^Hardware, ^OP_Systeme und ^Com_Sprache). Diese neue Sicht **Comp_words** stellt eine Vereinigung der drei anderen Sichten dar. Man kann auch von einer **hierarchischen Sicht** sprechen.

3. Definition eines **Tokens in den Quellwörterbüchern** (hier wieder SUBST). Alle Einträge, in deren Zeile dieses Token vorkommt, werden in diese Sicht (*Substantive*) aufgenommen. An diesem Beispiel in unserem Profile sieht man auch, daß die verschiedenen Arten der Definition miteinander vermischt werden können. In unserem Beispiel werden auch alle Einträge der Sicht *Comp_words* in die neue Sicht **Substantive** aufgenommen.

B.1.4 Das Phrasen-Profile

Das letzte hier beschriebene Profile ist in einer Datei mit Namen `phrasen.pro` gespeichert. Diese Datei hält in jeder Zeile eine Phrase, Kommentarzeilen hingegen sind nicht vorgesehen.

```
sehr geehrte damen und herren
sehr geehrter herr
sehr geehrte frau
als anlage erhalten sie
bezugnehmend auf ihr schreiben
```

Sucht man zur Laufzeit das Wort *„bezugnehmend“* und frägt mit der Funktion *get-next-phrase* nach den zu diesem Wort gehörigen Phrasen, so bekäme man die Phrase *„bezugnehmend auf ihr schreiben“* zurück.

B.2 Erzeugen eines Quellwörterbuches

Wie bereits erwähnt, kann man zur Generierung mehrere Quellwörterbücher angeben. Die Defaultwerte sehen aber nur eine Quelle mit dem Namen `wbsource.txt` vor. Diese Datei enthält in jeder Zeile einen Eintrag. Die einzelnen Zeilen können jedoch zusätzlich verschiedene Arten der Informationen tragen.

```
freundlichen ADJEKT
rot ADJEKT
teuer ADJEKT
der FKTWRT
die FKTWRT
das FKTWRT
mit FKTWRT
auf FKTWRT
und FKTWRT
oder FKTWRT
gehen VERB
arbeiten VERB
trinken VERB
schlafen VERB
kaufen VERB
Empfehlungen SUBST
Computer SUBST
Erde SUBST
Arbeit SUBST
Wohnung SUBST
Fahrrad SUBST
Mensch SUBST
```

Das jeweils erste Wort einer Zeile ist der Schlüssel, unter dem der Eintrag zu finden ist. Die Schreibweise erlaubt sowohl Umlaute (ä, ö, ü) als auch scharfes 's' (ß). Großbuchstaben werden in kleine umgewandelt, wodurch alle Einträge einheitlich vorliegen. Bei Anfragen über die Lesart werden daher die Buchstaben ebenfalls in kleine umgewandelt.

Hat man eine Datei, die beispielsweise nur Substantive enthält, so ist es unzweckmäßig, jedem Eintrag das Token **SUBST** folgen zu lassen. Hierfür gibt es einen weiteren Mechanismus, den wir jetzt vorstellen möchten. Man fügt anstelle eines normalen Eintrages die Zeile

```
#global SUBST
```

in das Quellwörterbuch ein. Dies bewirkt ein automatisches Zufügen des Tokens **SUBST** an alle restlichen Einträge der momentanen Quelle [17]. Eine weitere Zeile dieser Art würde ein zusätzliches Token für den Rest der Datei definieren und zwar ab der Stelle, an der es steht. Das alte Token **SUBST** würde dadurch nicht seine Gültigkeit verlieren.

Um diese Globaldefinition auszunutzen, müßte unser Beispiel-Quellwörterbuch etwas abgeändert werden:

```
...
kaufen VERB
#global SUBST
Empfehlungen
Computer
Erde
Arbeit
...
```

Dieser Vorteil wird besonders bei großen Wortlisten gleichen Typs interessant (z.B. Saarbrücker Lexika), da durch Zufügen einer einzigen Zeile eine Kategorisierung aller Einträge dieser Datei möglich wird.

Dazu können der `#global`-Kennung auch mehrere Token folgen:

```
#global SUBST MAENNLICH
Computer
Monitor
Tisch
...
```

[17] Diese Globaldefinition läßt eine Analogie zum C-Präprozessor erkennen, der in einem der eigentlichen Compilierung vorgeschalteten Lauf den Quellcode u.a. durch die mit dem `#define`-Konstrukt definierten Textersetzungen versieht. Im Gegensatz zu diesen expliziten Änderungen werden die `#global`-Definitionen implizit auf die Daten angewandt, indem sie intern verwaltet werden. Es gibt also keinen separaten Lauf, der die Einträge mit entsprechenden Token versieht.

B.3 Aufruf des Generators

Wir haben nun alle notwendigen Dateien erzeugt, die wir benötigen, um das Programm **makedict** erfolgreich zu starten. Hierzu müssen wir wieder in das nächst höhere Verzeichnis gelangen und von dort *makedict* aufrufen. Zur Kontrolle schauen wir uns vorher und nachher einmal die Inhalte des Verzeichnisses `dict` an.

```
cd ..
ls dict/
   logview.pro     sichten.pro     wbsource.txt
   phrasen.pro     syntax.pro
makedict
   Einlesen von Quelldatei dict/wbsource.txt
ls dict/
   aux.dict.tmp    hash_tab.nr3    sichten.pro     wbsicht.fk3
   dict.inf        logview.ept     syntax.pro      wbsource.txt
   hash_tab.nr1    logview.pro     wbmain.bin
   hash_tab.nr2    phrasen.pro     wbsicht.fk2
```

Wie man sieht, hat *makedict* einige neue Dateien erzeugt. Wieder wurden Default-Namen verwendet.

- Die Dateien mit Namen `hash_tab.nr1`, `hash_tab.nr2` und `hash_tab.nr3` enthalten die Hashtabellen der Hashfunktionen 1 bis 3. Es handelt sich dabei um Binärdateien, die zur Initialisierungsphase vom System eingelesen werden.

- Die Dateien mit Namen `wbsicht.fk2` und `wbsicht.fk3` sind die Sichteinstiegsdateien zu den Hashfunktionen 2 und 3. Sie sind ebenfalls Binärdateien, werden jedoch nicht geladen. Aus ihnen wird mittels errechneter Hashcodes Information zur Laufzeit an bestimmten Stellen ausgelesen. Bei diesen Informationen handelt es sich um die Einstiegspunkte der einzelnen Sichtausprägungen der statischen Sichten.

- Die Datei `logview.ept` enthält Namen, Einstiegsadressen und Indizes der optionalen Sichten im virtuell partitionierten Wörterbuch.

- Die Datei `wbmain.bin` enthält das eigentliche, virtuell partitionierte Wörterbuch selbst. Dabei handelt es sich ebenfalls um eine Binärdatei. In ihr wird zur Laufzeit an den ermittelten Adressen die eigentliche Information ausgelesen.

- Die Datei `aux.dict.tmp` ist, wie ihr Name schon vermuten läßt, eine temporäre Datei, die nach Bedarf gelöscht werden kann. Sie enthält alle Wörter, die über Wortlisten in der Datei `logview.pro` definiert und nach Abarbeitung aller Quellen noch nicht im Hauptwörterbuch aufgenommen wurden. Der Generator erzeugt diese Datei, um sie im Anschluß an die spezifizierten Quellen wie ein weiteres, implizit spezifiziertes Quellwörterbuch abzuarbeiten. Dieses Verfahren garantiert, daß alle zur Definition optionaler Sichten explizit aufgeführten Wörter auch tatsächlich im generierten Lexikon enthalten sind.

- Die Datei `dict.inf` beinhaltet die Namen sämtlicher Profiles und Quellwörterbücher sowie die der generierten Dateien außer (ihrem eigenen und) dem des temporären Hilfswörterbuches `aux.dict.tmp` (nach der Generierung nicht mehr benötigt). Zusätzlich listet sie die Längen der drei Hashtabellen auf, da auch sie zur Laufzeit bekannt sein müssen.

  Zu beachten ist, daß die Pfadnamen der Dateien immer relativ zum aktuellen Verzeichnis der Generierung angegeben sind. Demzufolge kann das Lexikon auch nur von diesem Directory aus gestartet werden, es sei denn die Pfade in `dict/dict.inf` würden von Hand auf die volle Länge ergänzt werden. (Auch dies kann mit einem normalen Editor wie z.B. *vi* geschehen.)

  Allein diese Datei muß zur Initialisierung des Laufzeitsystems angegeben werden – alle übrigen Informationen werden aus ihr ausgelesen.

B.3.1 Ändern der Defaultwerte

Der Benutzer hat die Möglichkeit, alle voreingestellen Werte zu ändern. Auf diese Weise kann man eine Art Profile-Bibliothek verwalten, aus der man zur Generierungsphase das gewünschte Profile auswählt. Man stelle sich vor, man habe eine Quelldatei, die keinerlei zusätzliche Informationen zu ihren Einträgen besitzt. Es wäre wenig sinnvoll, in diesem Falle statische oder optionale Sichten zu definieren. Zu diesem Zweck können besondere Profiles spezifizert werden (leere Dateien). Der Generator legt folglich keinen Platz für Kollisionszeiger der Sichten und für Bitvektoren an und das resultierende virtuell partionierte Hauptwörterbuch wird in seinem Ergebnis wesentlich kompakter. Ebenso können je nach Art und Umfang der vorliegenden Syntaxinformation verschiedene Syntax-Profiles das System individuell an die Quellen anpassen.

Weiterhin können mehrere Quelldateien angegeben werden, falls sich der Wortschatz für das zu generierende Lexikon auf mehrere Dateien verteilt.

Darüberhinaus können andere Namen für die generierten Dateien vorgegeben werden, was das Überschreiben eines existierenden Lexikons mit seinem Dateienkomplex verhindert.

Schließlich hat der Benutzer die Möglichkeit, die Hashtabellen-Längen individuell einzustellen. Hier ist zu beachten, daß größere Werte zu kleineren Kollisionsketten führen, aber gleichzeitig auch zu mehr Platzbedarf – sowohl auf den Dateien als auch im Arbeitsspeicher, da die Hashtabellen zur Laufzeit speicherresident sind. Diese Problematik ist in Kapitel 3.2 ausführlich erklärt.

Wir wollen uns noch einmal die Aufrufstruktur zur Änderung der Defaultwerte in Erinnerung rufen:

```
makedict h
   Gebrauch : makedict [Optionen]
        -q  WB-Quelldateien (bis zu 32 Quelldateien)
        -o  WB-Zieldatei
        -s  WB-Sichteneinstiegsdatei(en)
        -h  Hashtabellen-Datei(en)
        -lx Hashtabellen-Laenge fuer Funktion Nr.x  x=(1|2|3|x)
        -p  Phrasendatei
        -x  Syntax-Profile
        -y  Statische Sichten Definitions-Datei
        -v  Optionale Sichten Definitions-Datei
```

Durch Eingabe eines '–', gefolgt von einem bedeutungstragenden Buchstaben, kann ein bestimmter Parameter verändert werden. Ein Beispiel soll die Wirkungsweise der Funktion veranschaulichen:

```
makedict -q worteCELEX -y empty.pro -v empty.pro
```

Durch diesen Befehl bauen wir ein Lexikon auf, welches keinerlei Sichten enthält, da als Profiles für statische und optionale Sichten eine leere Datei angegeben wurde. Das Ergebnis wird dadurch kompakter. Ferner handelt es sich bei `worteCELEX` um eine einfache

Wortliste ohne jegliche Zusatzinformation [CELEX91]. Die Namen der generierten Dateien bleiben gleich den Vorgabewerten.

Im folgenden Beispiel werden zum Unterschied mehrere Quelldateien angegeben. Die Profiles stehen wie üblich in den bekannten Dateien in einem Verzeichnis mit Namen `dict`.

```
makedict -q Subst Adjekt Verb FktWrt
```

Hier kann man sich den Wortschatz eingeteilt in Substantive, Adjektive, Verben und Funktionswörter vorstellen. Diese Art der Modularität kann z.B. durch das `#global`-Konstrukt begründet sein – bei den einzelnen Einträgen muß nicht jedesmal die Wortkategorie angegeben werden.

Anhang C

Beispielanfragen über die Lisp-Schnittstelle

Dieses Kapitel zeigt eine Lisp-Sitzung, die einige der Schnittstellenfunktionen demonstriert. Das dabei verwendete Lexikon beinhaltet 107.427 Einträge, eine statische Sicht über die Wortkategorie mit den Ausprägungen *SUBST*, *VERB*, *ADJEKT* und *FKTWORT* sowie mehrere optionale Sichten.

```
> (init-dict "/home/kieni/DICT/MschmidtDict.inf")
0
> (getword "kaiser*")
("kaiser" "kaiser" "kaisers" "kaiseradler" "kaisergranat" "kaiserhof"
"kaiserhummer" "kaiserjaeger" "kaiserling" "kaisermantel"
"kaiserschmarren" "kaiserschnitt" "kaiserschwamm" "kaiserin"
"kaiserinmutter" "kaiserkrone" "kaiserwuerde" "kaiserhaus"
"kaiserslautern" "kaisertum" "kaiserlich" "kaiserlichkoeniglich"
"kaiserlos" "kaiserslauterner" "kaiserlichen" "kaiserin"
"kaiserliche" "kaiserslautern")
> (getword "kaiser* SUBST")
("kaiser" "kaiseradler" "kaisergranat" "kaiserhof" "kaiserhummer"
"kaiserjaeger" "kaiserling" "kaisermantel" "kaiserschmarren"
"kaiserschnitt" "kaiserschwamm" "kaiserin" "kaiserinmutter"
"kaiserkrone" "kaiserwuerde" "kaiserhaus" "kaiserslautern"
"kaisertum")
> (getview "LV-city-name")
("langen" "hennef" "kaiserslautern" "konz" "kronberg" "ludwigshafen"
"mannheim" "muenchen" "saarbruecken" "siegen" "tawern" "tuebingen")
> (set-filter 0 "LV-city-name")
T
> (getword "kaiser*")
("kaiserslautern")
> (reset-filter)
T
```

```
> (getword "d[roce][mnu][9gjq][eco]?")
("donjon" "dengel")
> (set-filter 0 "Angest")
T
> (getword "d[roce][mnu][9gjq][eco]?")
("dengel")
> (negate-filter 0)
T
> (getword "d[roce][mnu][9gjq][eco]?")
("donjon")
> (clear-filter 0)
T
> (getwordprepare "damen")
1
> (getwordnext)
"damen"
> (get-next-phrase)
"sehr geehrte damen und herren"
> (get-next-phrase)
NIL
> (getwordprepare "S[ce]h[u]??")
1
> (getwordnext)
"schubs"
> (getwordnext)
"schuft"
> (getwordnext)
"schuhu"
> (getwordnext)
"schund"
> (getwordnext)
"schupf"
> (getwordnext)
"schupp"
> (getwordnext)
"schups"
> (getwordnext)
"schurf"
> (getwordnext)
"schurz"
> (getwordnext)
"schuss"
> (getwordsyntax)
("MASC" "SUBST")
> (getwordnext)
"schutt"
```

```
> (getwordnext)
"schutz"
> (getwordnext)
"schufa"
> (getwordsyntax)
("FEMI" "SUBST")
> (getwordnext)
"schuld"
> (getwordnext)
"schule"
> (getwordsyntax)
("FEMI" "SUBST")
> (getwordviews)
("Weibl" "Subst")
> (time (getview-cardinality "Weibl"))
Elapsed Real Time = 3.28 seconds
Total Run Time    = 3.12 seconds
User Run Time     = 2.95 seconds
System Run Time   = 0.17 seconds
Process Page Faults    =          5
Dynamic Bytes Consed   =          0
Ephemeral Bytes Consed =        736
30739
> (exit-dict)
T
> (quit)
kieni@serv-400 1:57pm [DICT]
```

Literaturverzeichnis

[Bunke90] H. Bunke: *String matching for structural pattern recognition*, Series in Computer Science – Vol.7, World Scientific, 1990, pp.119-144.

[CELEX91] CELEX: *Temporary German Linguistic Guide*, German database version D1.0, Centre for Lexical Information, University of Nijmegen, The Netherlands, January 1991.

[Comer79] D. Comer: *The ubiquitous B-tree*, Computing Surveys, Vol.11, No.2, June 1979, pp.121-137.

[Dengel92] A. Dengel, A. Pleyer, R. Hoch: *Fragmentary string matching by selective Access to hybrid tries*, Proc. of 11th International Conference on Pattern Recognition. The Hague, The Netherlands, August 30 – September 3, 1992, Vol.2, pp.149-153.

[Dittrich92] S. Dittrich: *Automatische, Dekriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen*, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, Mai 1992.

[Eliman90] D. G. Elliman: *A review of segmentation and contextual analysis techniques for text recognition*, Pattern Recognition, Vol.23, No.3/4, 1990, pp.337-346.

[Ide91] N. M. Ide, J. Veronis, J. Le Maitre: *Outline of a database model for electronic Dictionaries*, Proc. of RIAO 91 Intelligent Text and Image Handling, Barcelona, April 1991, Vol.1, pp.375-393.

[Harris85] M. D. Harris: *Introduction to Natural Language Processing*, Reston Publishing Company Inc., Reston, Virginia, 1985.

[Hoch92] R. Hoch, M. Malburg: *Designing a structured lexicon for document analysis*, Proc. of Seventh Intl. Summer School on Information Technologies and Programming, June 28 – July 5, Sofia, 1992, pp.71-76.

[Hull88] J.J.Hull: *A computational theory of visual word recognition*, Technical Report 88-07, Department of Computer Science, State University of new York at Buffalo, 1988 (Dissertation).

[John93] I. John: *Automatische Generierung von Wortlisten zum Aufbau eines Lexikons für die Dokumentanalyse*, Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1993.

[Kernighan78] B. W. Kernighan, D. M. Ritchie: *The C programming language*, Prentice-Hall, 1978.

[Kirchmann93] H. Kirchmann: *Einsatz von Insel-Parsing-Strategien bei der Dokumentanalyse*, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, März 1993.

[Knuth73] D. E. Knuth: *The Art of Computer Programming*, Vol. 1-3, Addison-Wesley 1968-73.

[Lucid] LUCID, Inc.: *Lucid Common Lisp/SUN, Advanced User's Guide*, Version 4.0, October 1991.

[Maurer75] W. D. Maurer, T. G. Lewis: *Hashtable Methods*, Computing Surveys, Vol.7, No.1, 1975, pp.5-19.

[Meier78] H. Meier: *Deutsche Sprachstatistik*, 2. Auflage, Georg Olms Verlag, Hildesheim, New York, 1978.

[Nix81] R. Nix: *Experience with a space efficient way to store a dictionary*, Communications of the ACM, Mai 1981, Vol.24, No.5, pp.297-298.

[Peterson80] James L. Peterson: *Computer programs for detecting and correcting spelling errors*, Communications of the ACM, Dezember 1980, Vol.23, No.12, pp.676-687.

[Pleyer91] A. Pleyer: *Ein Ansatz zum wahlfreien Zugriff auf ein Wörterbuch bei bruchstückhafter Information*, Projektarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1991.

[Riseman74] E. M. Riseman, A. R. Hanson: *A contextual postprocessing system for error correction using binary n-grams*, IEEE Transactions on Computers, Vol.c-2, No.5, Mai 1974, pp.480-493.

[Sinha90] R. M. K. Sinha: *On partitioning a dictionary for visual text recognition*, Pattern Recognition, Vol.23, No.3/4, 1990, pp.497-500.

[Srihari83] S. N. Srihari, J. J. Hull, R. Choudhari: *Integrating Diverse Knowledge Sources in Text Recognition*, ACM Transactions on Office Information Systems, Vol.1, No.1. Januar 1983, pp.68-87.

[Takahashi90] H.Takahashi, N.Itoh, T.Amano, A.Yamashita: *A spelling correction method and its application to an OCR-system*, Pattern Recognition, Vol.23, No.3/4, 1990, pp.363-377.

[Wagner92] A. Wagner: *Organisation und Zugriffsstrukturen eines Lexikons zur Dokumentanalyse - Ein hashbasierter Ansatz*,Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, Mai 1992.

[Wirth75] N. Wirth: *Algorithmen und Datenstrukturen*, Teubner Studienbücher Informatik, 1975.

[Wu92] S. Wu, U. Manber: *Fast text searching allowing errors*, Communications of the ACM, Vol.35, No.10, Oktober 1992, pp.83-91.

[ISO8613] ISO 8613: *Information Processing - Text and Office Systems - Office Document Architecture and Interchange Format (ODA/ODIF)*, Vol.1-3, parts 1-8, 1988.

[ISO9735] ISO 9735: *Electronic Date Interchange for Administration, Commerce and Transport (EDIFACT); Application Level Syntax Rules*, 1988.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

DFKI Publications

The following DFKI publications or the list of all published papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

DFKI Research Reports

RR-92-25
Franz Schmalhofer, Ralf Bergmann, Otto Kühn, Gabriele Schmidt: Using integrated knowledge acquisition to prepare sophisticated expert plans for their re-use in novel situations
12 pages

RR-92-26
Franz Schmalhofer, Thomas Reinartz, Bidjan Tschaitschian: Intelligent documentation as a catalyst for developing cooperative knowledge-based systems
16 pages

RR-92-27
Franz Schmalhofer, Jörg Thoben: The model-based construction of a case-oriented expert system
18 pages

RR-92-29
Zhaohui Wu, Ansgar Bernardi, Christoph Klauck: Skeletel Plans Reuse: A Restricted Conceptual Graph Classification Approach
13 pages

RR-92-30
Rolf Backofen, Gert Smolka:
A Complete and Recursive Feature Theory
32 pages

RR-92-31
Wolfgang Wahlster:
Automatic Design of Multimodal Presentations
17 pages

RR-92-33
Franz Baader: Unification Theory
22 pages

RR-92-34
Philipp Hanschke: Terminological Reasoning and Partial Inductive Definitions
23 pages

RR-92-35
Manfred Meyer:
Using Hierarchical Constraint Satisfaction for Lathe-Tool Selection in a CIM Environment
18 pages

RR-92-36
Franz Baader, Philipp Hanschke:
Extensions of Concept Languages for a Mechanical Engineering Application
15 pages

RR-92-37
Philipp Hanschke: Specifying Role Interaction in Concept Languages
26 pages

RR-92-38
Philipp Hanschke, Manfred Meyer:
An Alternative to H-Subsumption Based on Terminological Reasoning
9 pages

RR-92-40
Philipp Hanschke, Knut Hinkelmann: Combining Terminological and Rule-based Reasoning for Abstraction Processes
17 pages

RR-92-41
Andreas Lux: A Multi-Agent Approach towards Group Scheduling
32 pages

RR-92-42
John Nerbonne:
A Feature-Based Syntax/Semantics Interface
19 pages

RR-92-43
Christoph Klauck, Jakob Mauss: A Heuristic driven Parser for Attributed Node Labeled Graph Grammars and its Application to Feature Recognition in CIM
17 pages

RR-92-44
Thomas Rist, Elisabeth André: Incorporating Graphics Design and Realization into the Multimodal Presentation System WIP
15 pages

RR-92-45
Elisabeth André, Thomas Rist: The Design of Illustrated Documents as a Planning Task
21 pages

RR-92-46
Elisabeth André, Wolfgang Finkler, Winfried Graf, Thomas Rist, Anne Schauder, Wolfgang Wahlster: WIP: The Automatic Synthesis of Multimodal Presentations
19 pages

RR-92-47
Frank Bomarius: A Multi-Agent Approach towards Modeling Urban Traffic Scenarios
24 pages

RR-92-48
Bernhard Nebel, Jana Koehler:
Plan Modifications versus Plan Generation: A Complexity-Theoretic Perspective
15 pages

RR-92-49
Christoph Klauck, Ralf Legleitner, Ansgar Bernardi: Heuristic Classification for Automated CAPP
15 pages

RR-92-50
Stephan Busemann:
Generierung natürlicher Sprache
61 Seiten

RR-92-51
Hans-Jürgen Bürckert, Werner Nutt:
On Abduction and Answer Generation through Constrained Resolution
20 pages

RR-92-52
Mathias Bauer, Susanne Biundo, Dietmar Dengler, Jana Koehler, Gabriele Paul: PHI - A Logic-Based Tool for Intelligent Help Systems
14 pages

RR-92-53
Werner Stephan, Susanne Biundo:
A New Logical Framework for Deductive Planning
15 pages

RR-92-54
Harold Boley: A Direkt Semantic Characterization of RELFUN
30 pages

RR-92-55
John Nerbonne, Joachim Laubsch, Abdel Kader Diagne, Stephan Oepen: Natural Language Semantics and Compiler Technology
17 pages

RR-92-56
Armin Laux: Integrating a Modal Logic of Knowledge into Terminological Logics
34 pages

RR-92-58
Franz Baader, Bernhard Hollunder:
How to Prefer More Specific Defaults in Terminological Default Logic
31 pages

RR-92-59
Karl Schlechta and David Makinson: On Principles and Problems of Defeasible Inheritance
13 pages

RR-92-60
Karl Schlechta: Defaults, Preorder Semantics and Circumscription
19 pages

RR-93-02
Wolfgang Wahlster, Elisabeth André, Wolfgang Finkler, Hans-Jürgen Profitlich, Thomas Rist:
Plan-based Integration of Natural Language and Graphics Generation
50 pages

RR-93-03
Franz Baader, Berhard Hollunder, Bernhard Nebel, Hans-Jürgen Profitlich, Enrico Franconi:
An Empirical Analysis of Optimization Techniques for Terminological Representation Systems
28 pages

RR-93-04
Christoph Klauck, Johannes Schwagereit:
GGD: Graph Grammar Developer for features in CAD/CAM
13 pages

RR-93-05
Franz Baader, Klaus Schulz: Combination Techniques and Decision Problems for Disunification
29 pages

RR-93-06
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: On Skolemization in Constrained Logics
40 pages

RR-93-07
Hans-Jürgen Bürckert, Bernhard Hollunder, Armin Laux: Concept Logics with Function Symbols
36 pages

RR-93-08
Harold Boley, Philipp Hanschke, Knut Hinkelmann, Manfred Meyer: COLAB: A Hybrid Knowledge Representation and Compilation Laboratory
64 pages

RR-93-09
Philipp Hanschke, Jörg Würtz:
Satisfiability of the Smallest Binary Program
8 Seiten

RR-93-10
Martin Buchheit, Francesco M. Donini, Andrea Schaerf: Decidable Reasoning in Terminological Knowledge Representation Systems
35 pages

RR-93-11
Bernhard Nebel, Hans-Juergen Buerckert:
Reasoning about Temporal Relations:
A Maximal Tractable Subclass of Allen's Interval Algebra
28 pages

RR-93-12
Pierre Sablayrolles: A Two-Level Semantics for French Expressions of Motion
51 pages

RR-93-13
Franz Baader, Karl Schlechta:
A Semantics for Open Normal Defaults via a Modified Preferential Approach
25 pages

RR-93-14
Joachim Niehren, Andreas Podelski,Ralf Treinen:
Equational and Membership Constraints for Infinite Trees
33 pages

RR-93-15
Frank Berger, Thomas Fehrle, Kristof Klöckner, Volker Schölles, Markus A. Thies, Wolfgang Wahlster: PLUS - Plan-based User Support
Final Project Report
33 pages

RR-93-16
Gert Smolka, Martin Henz, Jörg Würtz: Object-Oriented Concurrent Constraint Programming in Oz
17 pages

RR-93-20
Franz Baader, Bernhard Hollunder:
Embedding Defaults into Terminological Knowledge Representation Formalisms
34 pages

RR-93-23
Andreas Dengel, Ottmar Lutzy:
Comparative Study of Connectionist Simulators
20 pages

RR-93-24
Rainer Hoch, Andreas Dengel:
Document Highlighting —
Message Classification in Printed Business Letters
17 pages

DFKI Technical Memos

TM-91-12
Klaus Becker, Christoph Klauck, Johannes Schwagereit: FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

TM-91-13
Knut Hinkelmann: Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

TM-91-14
Rainer Bleisinger, Rainer Hoch, Andreas Dengel:
ODA-based modeling for document analysis
14 pages

TM-91-15
Stefan Busemann: Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

TM-92-01
Lijuan Zhang: Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

TM-92-02
Achim Schupeta: Organizing Communication and Introspection in a Multi-Agent Blocksworld
32 pages

TM-92-03
Mona Singh:
A Cognitiv Analysis of Event Structure
21 pages

TM-92-04
Jürgen Müller, Jörg Müller, Markus Pischel, Ralf Scheidhauer:
On the Representation of Temporal Knowledge
61 pages

TM-92-05
Franz Schmalhofer, Christoph Globig, Jörg Thoben:
The refitting of plans by a human expert
10 pages

TM-92-06
Otto Kühn, Franz Schmalhofer: Hierarchical skeletal plan refinement: Task- and inference structures
14 pages

TM-92-08
Anne Kilger: Realization of Tree Adjoining Grammars with Unification
27 pages

TM-93-01
Otto Kühn, Andreas Birk: Reconstructive Integrated Explanation of Lathe Production Plans
20 pages

DFKI Documents

D-92-13
Holger Peine: An Investigation of the Applicability of Terminological Reasoning to Application-Independent Software-Analysis
55 pages

D-92-14
Johannes Schwagereit: Integration von Graph-Grammatiken und Taxonomien zur Repräsentation von Features in CIM
98 Seiten

D-92-15
DFKI Wissenschaftlich-Technischer Jahresbericht 1991
130 Seiten

D-92-16
Judith Engelkamp (Hrsg.): Verzeichnis von Softwarekomponenten für natürlichsprachliche Systeme
189 Seiten

D-92-17
Elisabeth André, Robin Cohen, Winfried Graf, Bob Kass, Cécile Paris, Wolfgang Wahlster (Eds.): UM92: Third International Workshop on User Modeling, Proceedings
254 pages
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-92-18
Klaus Becker: Verfahren der automatisierten Diagnose technischer Systeme
109 Seiten

D-92-19
Stefan Dittrich, Rainer Hoch: Automatische, Deskriptor-basierte Unterstützung der Dokumentanalyse zur Fokussierung und Klassifizierung von Geschäftsbriefen
107 Seiten

D-92-21
Anne Schauder: Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

D-92-22
Werner Stein: Indexing Principles for Relational Languages Applied to PROLOG Code Generation
80 pages

D-92-23
Michael Herfert: Parsen und Generieren der Prolog-artigen Syntax von RELFUN
51 Seiten

D-92-24
Jürgen Müller, Donald Steiner (Hrsg.): Kooperierende Agenten
78 Seiten

D-92-25
Martin Buchheit: Klassische Kommunikations- und Koordinationsmodelle
31 Seiten

D-92-26
Enno Tolzmann:
Realisierung eines Werkzeugauswahlmoduls mit Hilfe des Constraint-Systems CONTAX
28 Seiten

D-92-27
Martin Harm, Knut Hinkelmann, Thomas Labisch: Integrating Top-down and Bottom-up Reasoning in COLAB
40 pages

D-92-28
Klaus-Peter Gores, Rainer Bleisinger: Ein Modell zur Repräsentation von Nachrichtentypen
56 Seiten

D-93-01
Philipp Hanschke, Thom Frühwirth: Terminological Reasoning with Constraint Handling Rules
12 pages

D-93-02
Gabriele Schmidt, Frank Peters, Gernod Laufkötter: User Manual of COKAM+
23 pages

D-93-03
Stephan Busemann, Karin Harbusch(Eds.):
DFKI Workshop on Natural Language Systems: Reusability and Modularity - Proceedings
74 pages

D-93-04
DFKI Wissenschaftlich-Technischer Jahresbericht 1992
194 Seiten

D-93-06
Jürgen Müller (Hrsg.):
Beiträge zum Gründungsworkshop der Fachgruppe Verteilte Künstliche Intelligenz Saarbrücken 29.-30. April 1993
235 Seiten
Note: This document is available only for a nominal charge of 25 DM (or 15 US-$).

D-93-07
Klaus-Peter Gores, Rainer Bleisinger:
Ein erwartungsgesteuerter Koordinator zur partiellen Textanalyse
53 Seiten

D-93-08
Thomas Kieninger, Rainer Hoch: Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse
125 Seiten

Ein Generator mit Anfragesystem für strukturierte Wörterbücher zur Unterstützung von Texterkennung und Textanalyse

D-93-08
Document

Thomas Kieninger, Rainer Hoch