Technische Universität Wien
Vienna University of Technology

D I P L O M A R B E I T

Effiziente Gestaltung der Wortanalyse in SiSiSi

ausgeführt am

Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter Anleitung von

em.Univ.-Prof. Dr. rer. nat. Wilhelm Barth
Univ.-Ass. Dr. rer. soc. oec. Gabriele Koller
Univ.-Ass. Dr. techn. Martin Schönhacker

durch

Martin Gruber
Hietzinger Hauptstraße 97/10
A-1130 WIEN

Datum

Unterschrift

Zusammenfassung

Die automatische Silbentrennung in der deutschen Sprache ist, dank der Möglichkeit von fast beliebigen Wortzusammensetzungen, ein komplexes Problem, das in vielen Bereichen noch nicht zufriedenstellend gelöst ist, wie z.B. ein Blick in deutschsprachige Tageszeitungen beweist.

Am Institut für Computergraphik und Algorithmen wird bereits seit längerer Zeit an der sicheren, sinnentsprechenden Silbentrennung (SiSiSi) gearbeitet, welche deutsche Wörter mit Hilfe einer Wortbildungsgrammatik in ihre kleinsten relevanten Bestandteile (Atome) aufteilt und diese Zerlegung als Grundlage für eine korrekte, den Sinn nicht entstellende Silbentrennung heranzieht. Dieses Programmpaket hat nun einen Entwicklungsstand erreicht, der die Verwendung von SiSiSi auch für Anwender außerhalb des universitären Bereiches interessant macht. Entsprechende Anfragen existieren bereits.

Diese Arbeit beschäftigt sich mit der Reimplementierung von SiSiSi in Java™. Diese wurde notwendig, da das bisherige System, das hauptsächlich zum Testen und Verfeinern des Wortanalysealgorithmus eingesetzt wurde, relativ ineffizient und durch die Verwendung von Delphi™ als Programmiersprache nicht plattformunabhängig war. Besonderes Augenmerk wurde dabei auf die verwendeten Datenstrukturen gelegt, die nun deutlich besser an ihre jeweiligen Aufgaben angepasst werden konnten. Insgesamt konnte damit das Laufzeitverhalten von SiSiSi deutlich verbessert werden.

Schlüsselwörter:
Wortanalyse, Deutsch, Silbentrennung, Java™.

Abstract

Automatic hyphenation of German words is a complex problem. The German language uses a substantial number of compound words formed in such way that careless hyphenation can subtly distort word meaning and hamper reading comprehension. As even a quick glance into almost any German newspaper readily shows, the problem has not been solved satisfactorily yet.

At the Institute of Computer Graphics and Algorithms we are working on SiSiSi, a word analysis system for reliable, sense-conveying hyphenation. The main idea behind SiSiSi is to split words into their smallest relevant parts (atoms) based on German word synthesis rules. These decompositions can then be used to hyphenate words reliably. As of recently, SiSiSi is no longer purely experimental but fit for use in real-world scenarios.

The core of this diploma thesis is a Java™ implementation of SiSiSi which became necessary because the existing proof-of-concept implementation had not been designed for efficiency and, having been written in Delphi™, is not platform-independent, either. In particular, the use of new sophisticated data structures has led to a significant performance increase of the SiSiSi system.

Keywords:
word analysis, German, hyphenation, Java™

Danksagung

An dieser Stelle möchte ich mich bei all jenen Personen bedanken, die im Laufe der Zeit auf unterschiedlichste Art und Weise dazu beigetragen haben, dass ich diese Diplomarbeit verwirklichen konnte.

Ganz besonders danke ich meinem Betreuer *em.Univ.-Prof. Dr. Wilhelm Barth*, der mich mit zahlreichen konstruktiven Gesprächen, Ideen und Anregungen für meine Diplomarbeit, aber auch in der letzten Phase meines Studiums tatkräftig mit viel persönlichem Einsatz unterstützt hat.

Mein Dank gilt auch *Dr. Gabriele Koller* und *Dr. Martin Schönhacker*, die mir nicht nur als Autoren von Delphi-SiSiSi immer mit Rat und Tat zur Seite standen und mir tiefe Einblicke in den zugrunde liegenden Algorithmus ermöglichten, sondern mir auch wertvolle Hinweise für diese Arbeit lieferten und sich fleißig als Korrekturleser betätigten.

Bei Frau *Univ.-Prof. Dr. Petra Mutzel* und meinen Kolleginnen und Kollegen am Institut für Computergraphik und Algorithmen bedanke ich mich sowohl für die angenehme Arbeitsatmosphäre als auch für den Freiraum neben meiner beruflichen Tätigkeit für das Institut. Beides hat nicht unwesentlich zur Realisierung dieser Arbeit beigetragen.

Nicht zuletzt möchte ich mich auch bei meinen Eltern ganz herzlich bedanken, die mir das Studium ermöglicht und mich immer voll und ganz unterstützt haben, sowie bei meinen Studienkollegen und Freunden, die mich in dieser Zeit begleitet haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Silbentrennung, warum?

Die Lesbarkeit eines Textes hängt natürlich primär von seinem Inhalt ab, unterstützt wird der Lesefluss aber durch ein angenehmes, ruhiges Schriftbild ohne gefranste Ränder oder unansehnliche Wortzwischenräume.

Besonders das Deutsche mit seinen zum Teil sehr langen Wörtern benötigt daher für ein schönes Schriftbild eine gut funktionierende Silbentrennung. Dass es auf diesem Gebiet noch sehr viel zu tun gibt, beweisen immer wieder auftauchende Trennfehler in aktuellen deutschsprachigen Tageszeitungen oder Zeitschriften, wo meist schmale Spalten verwendet werden, für die eine korrekte Silbentrennung besonders wichtig ist. Die Silbentrennung umfangreicher Textdokumente ist sehr aufwändig und sollte daher vorzugsweise automatisch vorgenommen werden.

1.2 Methoden zur automatischen Silbentrennung

Die wohl bekannteste Methode zur Silbentrennung wurde von Frank M. Liang [Lia83] entwickelt und wird u.a. im Textsatzsystem TEX verwendet. Dieses für die englische Sprache konzipierte Verfahren basiert auf Buchstabenmustern (Folge von Buchstaben, *patterns*) unterschiedlicher Priorität, die die Trennung eines Wortes an einer bestimmten Stelle erzwingen oder aber verhindern. Diese Muster können automatisch und relativ effizient aus Wörterbüchern mit Vortrennungen generiert werden. Der Trennalgorithmus selbst ist vergleichsweise einfach und schnell, da er sich im Wesentlichen auf die Suche nach bestimmten Buchstabenfolgen in Wörtern beschränkt.

Allerdings geht dieses Verfahren davon aus, dass für die Erzeugung der verschiedenen Trennmuster alle möglichen Wörter und Wortformen in einer Tabelle aufgelistet sind. Aus dieser Tabelle können dann die *patterns* automatisch erzeugt werden. Für die englische Sprache ist das relativ gut durchführbar. Sprache ist aber etwas Lebendiges und ist daher immer Veränderungen unterworfen.

Es können daher nie tatsächlich alle Wörter und Wortformen aufgezählt werden. Das gilt besonders für die deutsche Sprache, da durch die Kombination von zwei oder mehreren Wörtern ständig neue Wörter kreiert werden können. Ein gutes Beispiel hierfür ist der zumindest im Donauraum berühmt-berüchtigte *Donaudampfschifffahrtskapitän*.

Genau an den Nahtstellen von zusammengesetzten Wörtern, den Wortfugen, unterlaufen dem Verfahren von Liang für die deutsche Sprache (bzw. für alle Sprachen, die Wortzusammensetzungen gestatten, wie z.B. auch das Schwedische oder Ungarische) öfter Fehler, da die hier auftretenden Buchstabenkombinationen oft abweichend von den für das Wortinnere geltenden Regeln zu trennen sind. Diese Methode kann daher zu mitunter unterhaltsamen, in letzter Konsequenz aber den Lesefluss störenden, falschen Trennstellen führen, wie dieses Beispiel der deutschen Silbentrennung von TEX deutlich zeigt: *Grammati-kregeln*.

Eine weitere Möglichkeit der Silbentrennung besteht in der Verwendung von Trennregeln, die aus der Grammatik abgeleitet wurden. Dabei werden mit den erzeugten Regeln bestimmte Vokal-Konsonanten-Folgen getrennt. Dieses Verfahren kann durch eine gesonderte Behandlung von Vorsilben und Endungen verbessert werden, besitzt aber die gleiche Schwäche an den Wortfugen wie die Methode von Liang.

Es ist auch vorstellbar, alle Wörter mit korrekt eingetragenen Trennstellen in einem großen Wörterbuch zu speichern. Abgesehen von dem doch beachtlichen Speicherplatzbedarf eines solchen Wörterbuches (mehrere 100.000 Einträge) wurde bereits die Sinnlosigkeit des Versuchs aufgezeigt, alle deutschen Wörter zu erfassen. Daher ist auch diese Methode zum Scheitern verurteilt, will man beliebige Wortzusammensetzungen korrekt trennen. Allerdings hat das Wörterbuch den Vorteil, keine Trennfehler zu produzieren, wenn unbekannte Wörter einfach nicht getrennt werden. Trotzdem ist das kein befriedigender Zustand, da gerade die langen Wortzusammensetzungen Kandidaten für eine Silbentrennung sind.

1.3 Sichere sinnentsprechende Silbentrennung

Die sichere, sinnentsprechende Silbentrennung (SiSiSi) geht einen anderen Weg. Hier wird versucht, das ganze Wort zu analysieren und in seine syntaktischen Bestandteile zu zerlegen. Dabei wird die Zerlegung durch die Implementierung einer deutschen Wortbildungsgrammatik unterstützt, um grammatikalisch ungültige Zusammensetzungen von Wortbestandteilen auszuschließen.

Diese Bestandteile, im Weiteren auch Atome genannt, sind u.a. Vorsilben, Stämme und Endungen. Selten sind es auch ganze Wörter, dann nämlich, wenn diese Wörter nicht mit Hilfe der deutschen Wortbildungsgrammatik erzeugt werden können oder bei der anschließend noch auf die gefundenen Wortteile bzw. Kombinationen von Wortteilen angewandten Trennung nach Sprechsilben Fehler auftreten würden, z.B. bei (eingedeutschten) Fremdwörtern oder Eigennamen. Die Atome werden in einer eigenen Liste, der Atomtabelle, gespeichert. Dabei

handelt es sich nicht um ein vollständiges Wörterbuch! Es werden ausschließlich die kleinstmöglichen Wortbestandteile abgelegt. Zusammengesetzte Wörter und Wortformen (z.B. Deklinationsendungen), die sich daraus mit Hilfe der Wortgrammatik generieren lassen, müssen nicht zusätzlich eingetragen werden. Aktuell enthält die Atomtabelle rund 9.200 Atome, um Größenordnungen weniger als ein auch nur halbwegs vollständiges deutsches Wörterbuch inklusive aller Wortformen.

Die Vorteile dieses Verfahrens liegen auf der Hand: Im Gegensatz zu der reinen Wörterbuchmethode ist der Speicherverbrauch gering (Atomtabelle und Regelautomat für die Wortbildungsgrammatik), trotzdem können beliebige Wortzusammensetzungen erkannt, zerlegt und damit auch korrekt getrennt werden.

Doch nicht immer ist die Zerlegung eindeutig. Und manche davon sind zwar grammatikalisch gültig, aber trotzdem nicht sinnvoll. So kann *Personalmangel* richtig in *Personal-mangel* aufgeteilt werden, aber auch in *Person-alm-angel* (Wortzusammensetzung aus drei Substantiven). Und es gibt auch Wörter wie *Staubecken*, die wirklich zwei oder mehr Bedeutungen je nach Zerlegung haben können: *Stau-becken* bzw. *Staub-ecken*.

Die Sicherheit, bei der Silbentrennung keine falschen Trennstellen zu setzen, resultiert bei SiSiSi aus der Tatsache, dass alle nach der Wortbildungsgrammatik gültigen Zerlegungen erzeugt werden und nur die Trennstellen ohne Bedenken verwendet werden können (und damit *sicher* sind), die in all den gefundenen Zerlegungsvarianten vorkommen. Alle anderen sind *unsicher* und bedürfen einer Sonderbehandlung. Es gilt:

> Wenn die Atomtabelle alle Atome der deutschen Sprache enthält und die Grammatikregeln keine korrekte Wortbildung ausschließen, dann ist bei einem korrekt geschriebenen Wort keine der gefundenen und als sicher markierten Trennstellen falsch.

Für das Beispielwort *Personalmangel* bedeutet dies, dass nur die Trennstellen verwendet werden können, die sowohl in *Personal-mangel* als auch in *Person-alm-angel* vorkommen, das sind nach der Trennung nach Sprechsilben der einzelnen Wortbestandteile *Per-sonalman-gel*.

Eine weitere Stärke von SiSiSi ist die Unterscheidung zwischen verschiedenen Prioritäten für Trennstellen, denn nicht jede an sich gültige und (nach obiger Definition) sichere Trennstelle sollte mit Rücksicht auf den Lesefluss auch wirklich verwendet werden. Bei einer Wortzusammensetzung bietet sich als bevorzugte Trennstelle natürlich die Wortfuge an (höchste Priorität). Die Ausnutzung anderer Trennstellen, z.B. direkt nach einer Wortfuge, kann unschöne bzw. sogar den Sinn entstellende Folgen haben. Beispiele hierfür sind die prinzipiell möglichen und richtigen Trennvarianten *Spargel-der* oder *Kurdi-rektor*. SiSiSi stuft daher nach der Wortzerlegung und der anschließenden Trennung nach Sprechsilben die Priorität der Trennstellen an ungünstigen Positionen herab, um einer möglichen Sinnverfälschung des getrennten Wortes vorzubeugen.

Begonnen hat die Entwicklung von SiSiSi mit „UrSi“ (ursprüngliches SiSiSi) [BN85]. Die Wortbildungsgrammatik (und damit die Klassifikation der Atome) war zu diesem Zeitpunkt noch eher rudimentär. Es wurde nur zwischen Vorsilbe, Stamm, Endung und allein stehendem Stamm unterschieden. Es stellte sich schnell heraus, dass diese Grammatik ungenügend war, um sinnlose Kombinationen von Wortbestandteilen zu verhindern (z.B. darf eine Konjugationsendung nur auf einen Verbstamm folgen, nicht aber auf einen Substantiv- oder Adjektivstamm).

Eine genauere Klassifikation der Atome als in die oben genannten vier Klassen wurde notwendig, der Regelautomat für die Wortbildungsgrammatik musste deutlich erweitert werden. Die daraus resultierende SiSiSi-Version „HelSi“ [Ste95] kannte bereits knapp 200 verschiedene Atomklassen. Schließlich wurde durch die Rechtschreibreform der deutschen Sprache 1998 eine weitere Anpassung notwendig. Mit neuerlichen Erweiterungen und Verfeinerungen (momentan rund 250 Atomklassen) entstand „ReSi“ (SiSiSi für die reformierte Rechtschreibung) [Kod01]. Diese Version stellt den algorithmisch aktuellen Stand der Entwicklung von SiSiSi dar.

1.4 Einsatzmöglichkeiten von SiSiSi

Die Einsatzmöglichkeiten von SiSiSi gehen weit über die einfache Silbentrennung hinaus. So kann SiSiSi z.B. zur Rechtschreibkontrolle dienen, denn ein Wort, das mit Hilfe der Atomtabelle und Grammatikregeln nicht gültig zerlegt werden kann, enthält womöglich einen Tippfehler.

Auch bei der Volltextsuche [Bar90] eröffnet eine Wortanalyse des Suchbegriffes und des zu durchsuchenden Textes zusätzliche Möglichkeiten. So kann der Suchbegriff in seine syntaktischen Bestandteile zerlegt werden und dann von etwaigen Endungen, z.B. Deklinationsendungen, befreit werden. Damit kann die Qualität der Suche deutlich gesteigert werden, da nicht mehr „blind“ nach einem bestimmten Buchstabenmuster gesucht werden muss (*string matching*), sondern wirklich die einzelnen Wortbestandteile herangezogen werden können. Dann ist es egal, ob der Suchbegriff im durchsuchten Text im Singular oder Plural, im Nominativ, Dativ oder Akkusativ, im Präsens oder Perfekt oder als Teil in einem (langen) zusammengesetzten Wort vorkommt, er wird gefunden.

SiSiSi kann ferner zur automatischen Ermittlung der Silbenzahl aller Wörter eines Textes verwendet werden. Es hat sich gezeigt, dass sehr wohl Interesse an einem Programm existiert, das automatisch die Anzahl der Silben eines Wortes bestimmt. Diese Daten werden beispielsweise dazu verwendet, um eine leichter fassbare Maßzahl für die Lesbarkeit und Verständlichkeit eines Textes zu erhalten, da man davon ausgeht, dass ein langes Wort mit entsprechend mehr Silben schwerer zu lesen und zu verstehen ist.

1.5 Beweggründe für Java-SiSiSi

SiSiSi (aktuell ReSi) ist eine über die Jahre gewachsene Applikation mit einer Reihe von Problemen, die sich zwangsläufig bei einem solchen Wachstumsprozess ergeben. Ursprünglich in Turbo Pascal implementiert, wurde es später als umfangreicheres Testsystem in Delphi neu implementiert und weiterentwickelt.

Jetzt, wo das Konzept ausgereift scheint und das Interesse an diesem Programm auch außerhalb des universitären Bereichs wächst, bot sich die Möglichkeit einer sauberen Reimplementierung an.

Die Ziele von Java-SiSiSi können daher folgendermaßen zusammengefasst werden:

- Die bisherige Implementierung unter Delphi (im Folgenden kurz Delphi-SiSiSi genannt) ist bezüglich Betriebssystemplattform äußerst unflexibel. Um SiSiSi einem breiteren Benutzerkreis zugänglich zu machen, war eine Portierung in eine möglichst plattformunabhängige Programmiersprache notwendig. Die Wahl fiel dabei auf Java™ von Sun Microsystems.

- Viele der bisher in Delphi-SiSiSi eingesetzten Datenstrukturen wurden primär unter dem Gesichtspunkt der Flexibilität und Erweiterbarkeit ausgewählt, was während der Entwicklung eine sinnvolle Entscheidung war, um sich nicht selbst zu beschränken und Wege zu verbauen. Die Entwicklung von SiSiSi geht natürlich weiter, doch können bereits jetzt die Anforderungen, die an die eingesetzten Datenstrukturen gestellt werden, deutlich genauer formuliert werden. Daher ist nun eine bessere Optimierung auf ihren Verwendungszweck hin möglich, als es noch bei Delphi-SiSiSi der Fall war.

- Delphi-SiSiSi wird vor allem als Testumgebung für den Wortanalysealgorithmus eingesetzt. Damit sollten z.B. die Auswirkungen eines neu eingefügten Atoms oder einer neu aufgenommenen Regel untersucht werden. Für diesen Zweck müssen aber während der Wortzerlegung Unmengen an Testinformationen generiert und gespeichert werden, um sie später auswerten zu können. Dieser zusätzliche Aufwand benötigt entsprechend viel Zeit, ist aber für den ursprünglichen Zweck dieser Anwendung nicht notwendig. Java-SiSiSi beschränkt sich daher auf das Wesentliche und kann auf diese Art und Weise die Effizienz gegenüber Delphi-SiSiSi weiter steigern.

- Eine weitere Anforderung an Java-SiSiSi ist die Bereitstellung einer klar definierten und sauber implementierten Schnittstelle zum Wortanalysealgorithmus, um SiSiSi als Bibliothek in fremde Programme einbinden zu können. Damit soll es z.B. Textsystemen ermöglicht werden, die Silbentrennung von SiSiSi mit all ihren Vorteilen bezüglich Sicherheit und Bewertung der verwendeten Trennstellen zu importieren. Auch kann eine SiSiSi-Bibliothek in Client-Server-Applikationen eingesetzt werden, wo einige interessante Anwendungen vorstellbar sind, u.a. die Silbentrennung von Dokumenten

als Webservice oder zentrale SiSiSi-Server für die Bearbeitung der in einem Unternehmen anfallenden Dokumente.

Kapitel 2

Die Wortanalyse

Ziel von SiSiSi ist es, ein Wort in seine einzelnen Bestandteile (Atome) aufzuspalten. Damit soll erreicht werden, dass nicht nur Stämme, Vorsilben und Endungen, sondern bei Wortzusammensetzungen auch die Wortgrenzen der Einzelwörter erkannt und für eine richtige Silbentrennung herangezogen werden können. Das ist notwendig, um Trennfehler an Wortfugen, wie sie gerne von anderen Verfahren gemacht werden, zu vermeiden.

Der Kern von SiSiSi ist daher der Wortzerlegungsalgorithmus. Doch bei seiner Arbeit ist er auf nicht weniger wichtige Komponenten angewiesen:

- Atomtabelle(n) und die in ihr/ihnen gespeicherten Atome
- Regelautomat für die Wortbildungsgrammatik
- Trennung nach Vokal-Konsonanten-Folgen (Dudentrennung)

Eine ausführliche Darstellung ist in [Kod01] enthalten, eine Kurzform auch auf den Webseiten des Projekts (http://www.ads.tuwien.ac.at/research/SiSiSi/).

2.1 Atomtabelle(n)

Die Aufgaben einer Atomtabelle sind schnell umrissen: Atome speichern und verwalten. Dazu gehört auch, auf Anfrage möglichst effizient nach bestimmten Atomen zu suchen und diese, wenn gefunden, dem Wortzerlegungsalgorithmus zu übergeben.

Zu einem Atom (eigentlich ist diese Bezeichnung nicht ganz korrekt, da es sich hier nicht um eine unter allen Umständen unteilbare Worteinheit handelt) werden mehrere zusätzliche Informationen benötigt und eingetragen:

- Das Atom selbst und seine Länge.

- Ein Flag, das angibt, ob dieses Atom auch aus anderen Atomen gültig zusammengesetzt werden darf. So muss *aber* (Konjunktion) für das Wort *aberkennen* auch in *ab* und *er* zerlegbar sein.
- Ob dieses Atom Ausnahmetrennstellen besitzt. Wenn ja, dann werden auch die Position(en) der Ausnahmetrennstelle(n) gespeichert und weiters die Information, ob dieses Atom zusätzlich auch mit der normalen Dudentrennung getrennt werden darf.
- Die Liste der Atomklassen, die diesem Atom zugeordnet sind. Diese Klassifikation der Atome ist für den Regelautomaten der Wortbildungsgrammatik entscheidend, dazu gleich mehr. Weiters wird zu einigen Atomklassen eine Liste von Wortfamilien geführt. Die Wortfamilien sind für die Silbentrennung selbst zwar nicht von Interesse, werden aber für die Volltextsuche benötigt. So wird z.B. zum Atom *schwomm* (Klasse: Verbstamm) als Wortfamilie *schwimmen* vermerkt, damit, wenn in einem Text nach *schwimmen* gesucht wird, auch *geschwommen* als Treffer geliefert werden kann.

Eine Atomtabelle ist für die Wortanalyse in SiSiSi zwingend erforderlich, da sie die einzelnen Bausteine für die Wortzerlegung liefert. Doch beim Einsatz von SiSiSi in verschiedenen Anwendungsgebieten kann es durchaus Sinn machen, mehr als nur eine Atomtabelle zu verwenden. So ist es für die Standard-Atomtabelle, die mit SiSiSi ausgeliefert wird, nicht sinnvoll bzw. reichen auch die am Institut verfügbaren Kapazitäten nicht aus, alle Wortstämme für jede nur erdenkliche Spezialanwendung aufzunehmen, so z.B. medizinische oder geographische Fachbegriffe und Eigennamen.

Es besteht daher die Möglichkeit, neben der Standard-Atomtabelle auch noch so genannte Benutzer-Atomtabellen zu verwenden, die zusätzliche Wortstämme für den jeweiligen Anwendungszweck beinhalten. Dieser Mechanismus macht SiSiSi enorm flexibel und ermöglicht es dem Benutzer, den Wortschatz selbstständig den eigenen Bedürfnissen anzupassen, zu erweitern und zu strukturieren (Sammlung von Wortstämmen zu einem bestimmten Thema in einer eigenen Atomtabelle).

2.2 Regelautomat (Wortbildungsgrammatik)

Eine einfache Zerlegung eines Wortes in eine beliebige Folge von Atomen ist nicht zielführend. Dabei werden Unmengen von sinnlosen Zerlegungen generiert, die der deutschen Wortbildung zuwiderlaufen. Als Beispiel kann hier das Wort *Abteilungsleiter* dienen. Für dieses Wort ergibt sich ohne weitere Kontrolle, lediglich aus der Aneinanderreihung von Atomen aus der aktuellen Atomtabelle, die folgende, nicht vollständige Liste von Zerlegungen:

```
abteil·ung·s·leit·er
abteil·ung·s·l·eiter
```

```
abt·ei·lung·s·leit·er
abt·ei·lung·s·l·eiter
ab·te·i·lung·s·leit·er
ab·te·i·lung·s·l·eiter
```

Um möglichst viele unsinnige Zerlegungen zu eliminieren, werden die Atome genau klassifiziert, d.h. es wird vermerkt, welche Aufgabe(n) ein Atom bei der Wortbildung übernehmen kann (bei den momentan rund 9.200 in der Atomtabelle abgespeicherten Atomen eine äußerst zeitraubende Aufgabe). Beispiele für Atomklassen, von denen es zur Zeit knapp 250 gibt, sind allgemeine Vorsilben (*in*, *auf*), Vorsilben zur Ableitung von Verben (*ver*), Substantivstämme mit Nominativ Plural auf -*e* (*spiel*), Adverbien (*an*) oder Steigerungsendungen für den Superlativ (*est*). Einem Atom können mehrere Atomklassen zugewiesen sein, so z.B. dem Atom *spiel*, das sowohl als Substantiv (das *Spiel*) als auch – mit entsprechenden Endungen – als Verb (*spielen*, *spielte*, ...) verwendet werden kann.

Die Zustände des Regelautomaten (aktuell kennt SiSiSi 107) beschreiben den momentanen Status während der Wortanalyse, also z.B. den Wortanfang, den Zustand nach allgemeiner Vorsilbe, nach regulärem Verbstamm oder nach einer Kardinalzahl.

Die im Automaten gespeicherten Regeln (gegenwärtig rund 4.300) definieren nun den Übergang von einem Automatenzustand über eine Atomklasse in einen neuen Zustand, also z.B. Wortanfang (Ausgangszustand) → allgemeine Vorsilbe (Atomklasse) → Zustand nach einer allgemeinen Vorsilbe (Zielzustand). Mit diesen Regeln ist es jetzt möglich, exakt zu steuern, welche Atome (genauer: Atomklassen) bei der Wortzerlegung aneinandergefügt werden dürfen und welche nicht. Einige der Regeln setzen auch bereits die ersten Trennzeichen, zum Beispiel zwischen zwei Stämmen, d.h. an einer Wortfuge, oder nach Vorsilben.

Wie man sieht, ist die Klassifikation der Atome sehr detailliert und ermöglicht zusammen mit dem Regelautomaten das Ausscheiden von sehr vielen unsinnigen Wortzerlegungen (für jede spezielle Stammklasse sind nur wenige Endungen erlaubt, das Aneinanderfügen mehrerer Flexionsendungen wird stark eingeschränkt, usw.). Unter Anwendung des Regelautomaten während der Wortanalyse bleibt bei unserem Testwort *Abteilungsleiter* lediglich eine einzige gültige Zerlegung (noch keine Trennung) übrig (was allerdings nicht immer der Fall sein muss):

```
abteil·ung·s·leit·er
```

Der Regelautomat prüft allerdings nur die grammatikalische Richtigkeit der Zerlegung, der Sinn des Wortes bleibt ihm dabei verschlossen. So können damit auch nicht alle unsinnigen Zerlegungen bei Wörtern, wie z.B. *Personalmangel* in *Person-alm-angel* (eine grammatikalisch korrekte Wortzusammensetzung aus einzelnen Substantiven), verhindert werden.

Weitere Details zu den Atomklassen, den Automatenzuständen und den im Automaten gespeicherten Regeln zur Wortbildungsgrammatik sind in [Ste95] und [Kod01] nachzulesen.

2.3 Trennung nach Vokal-Konsonanten-Folgen

Bei der Trennung nach Vokal-Konsonanten-Folgen (Dudentrennung) handelt es sich ebenfalls um einen endlichen Automaten. Seine Aufgabe ist es, die bei der Wortzerlegung gefundenen Bestandteile (im Wesentlichen Stämme mit Endungen sowie mehrsilbige Vorsilben) einer zusätzlichen Trennung zu unterziehen, die auf Sprechsilben beruht. Die Regeln hierfür haben sich bei der Rechtschreibreform der deutschen Sprache 1998 verändert und sind im Duden [dDAKu98] im Abschnitt über die Richtlinien zur Rechtschreibung nachzulesen.

Der genaue Aufbau und die Arbeitsweise dieses Automaten ist erschöpfend in [Kod01] abgehandelt (dort auch als „Dudentrennung“ bezeichnet).

2.4 Wortzerlegungsalgorithmus

Im Wortzerlegungsalgorithmus laufen schließlich alle Fäden zusammen. Dabei handelt es sich um einen rekursiven Algorithmus, der ein Wort entgegennimmt, in der Atomtabelle nach Atomen sucht, die als Wortanfang in Frage kommen, und mit Hilfe des Regelautomaten für die Wortbildungsgrammatik den Wortrest rekursiv weiter zerteilt. Am Ende des Wortes angelangt wird überprüft, ob der Regelautomat sich in einem gültigen Endzustand befindet. Wenn das der Fall ist, wurde eine zulässige Zerlegung gefunden und es wird zum Abschluss der Silbentrennung noch die Trennung auf Basis der Vokal-Konsonanten-Folgen durchgeführt.

Nähere Einblicke in den genauen Ablauf einer Wortzerlegung liefert nun der folgende Pseudocode. Wie jeder rekursive Algorithmus zerfällt auch dieser grob in zwei große Bereiche: den Teil, wo das Wortende erreicht und damit möglicherweise eine neue, gültige Zerlegungsvariante gefunden wurde (Abbruchbedingung der Rekursion), und einen Teil, der rekursiv das Wort weiter zerteilt. Anbei einige zusätzliche Erläuterungen:

- Zeilen 2 und 5: Bereits das Eingabewort *word* kann Trennzeichen enthalten. Dabei werden zwei Typen unterschieden: Trennzeichen, die zur Schreibweise des Wortes gehören (*forced_strict*, d.h. Bindestriche), und bereits eingefügte optionale Trennstellen (*forced_optional*). Nur Wortzerlegungen, die diese vorgegebenen Trennstellen ebenfalls enthalten, sollen erlaubt sein. Auf diese Weise können u.a. Mehrdeutigkeiten, wie sie z.B. bei den Wörtern *Baumast* (*Baum-ast*, *Bau-mast*) oder *Wachstube* (*Wach-stube*, *Wachs-tube*) auftreten, bereits durch Setzen von Trennzeichen bei der Eingabe vermieden werden. Trennzeichen vom Typ *forced_strict* können – im Gegensatz zu

Algorithmus 1 Wortzerlegung*(word, word_pos, state)*

Input: das zu testende Wort *(word)*, die aktuelle Position innerhalb dieses Wortes *(word_pos)* und der aktuelle Zustand des Regelautomaten *(state)*
Start der Rekursion: *word*, *word_pos = 1*, Startzustand des Regelautomaten *(state)*

Output: kombinierter Trennvektor und Liste der gefundenen Trennvarianten

```
1: falls Wortende erreicht (word_pos > word.length) dann {
2:   falls Regelautomat in Endzustand und bisher aktueller Trennvektor alle
     forced_strict Trennstellen enthält dann {
3:     Dudentrennung der noch nicht getrennten Wortteile (z.B. Stämme mit
       Endungen) durchführen und aktuellen Trennvektor ergänzen
4:     unschöne Trennstellen des aktuellen Trennvektors abwerten
5:     falls aktueller Trennvektor alle forced_optional Trennstellen enthält
       dann {
6:       falls aktueller Trennvektor noch nicht in Liste der bisher gefundenen
         Trennvarianten dann {
7:         Trennvektor in diese Liste aufnehmen
8:         Trennvektor mit allen bisher gefundenen Trennvektoren kombi-
           nieren
9:       }
10:    }
11:  }
12: } sonst {
13:   alle Präfixatome von word ab wordpos aus der/den Atomtabelle(n) holen
      und nach Länge sortiert in einer Liste speichern
14:   solange noch unbearbeitetes Präfixatom in Liste (= atom) {
15:     falls Position word_pos+atom.length im Sperrvektor noch nicht als be-
        arbeitet markiert dann {
16:       Position word_pos+atom.length in Sperrvektor markieren
17:       atom besitzt Ausnahmetrennstelle(n) → behandeln
18:       zu atom gehörende Atomklassenliste holen
19:       solange noch unbearbeitete Atomklasse in Liste (= atom_class) {
20:         falls atom_class keine Vorbedingung benötigt oder geforderte Vor-
            bedingung in used_atom_classes gefunden dann {
21:           atom_class in Liste used_atom_classes einfügen
22:           Zielzustandsliste für das Paar (state,atom_class) aus dem Regel-
              automaten holen
23:           solange noch unbearbeiteter Zielzustand in Liste (= target_state)
              {
24:             von Regel (state, atom_class) → target_state geforderte Trenn-
                stelle setzen
25:             Wortzerlegung(word , wordpos+atom.length, target_state)
26:           }
27:         }
28:       }
29:     }
30:   }
31: }
```

forced_optional Trennzeichen – nur an Wortfugen auftreten, daher bereits die Überprüfung vor der in Zeile 3 durchgeführten Trennung anhand von Sprechsilben.

- Zeile 3: Die Wortzerlegung liefert mit Hilfe des Regelautomaten u.a. Trennstellen an Wortfugen oder zwischen einer Vorsilbe und einem Stamm. Die noch nicht getrennten Wortbestandteile (z.B. Stamm mit Endung) werden unter Beachtung vorgeschriebener Ausnahmetrennstellen (siehe auch Zeile 17) mit Hilfe der Dudentrennung nach Sprechsilben getrennt.

- Zeile 4: Aus Gründen der Lesbarkeit sollte nicht jede gefundene, wenn auch korrekte Trennstelle, ausgenutzt werden. So ist nach der Reform der deutschen Rechtschreibung aus dem Jahr 1998 das Abtrennen eines einzelnen Buchstaben gestattet (z.B. *A-der*), schön ist das aber nicht. Auch kann es bei der ersten Trennstelle nach einer Wortfuge zu sinnentstellenden Situationen kommen, so z.B. bei dem Wort *Spargelder* : an der Wortfuge getrennt ergibt sich *Spar-gelder*; die nächste Trennstelle nach der Wortfuge würde, wenn ausgenützt, zu *Spargel-der* führen. Trifft man auf eine solche ungünstige Trennstelle, ist der Lesefluss unweigerlich gestört.
  SiSiSi arbeitet daher mit einer feineren Abstufung als „Trennstelle ja oder nein“. Es wird unterschieden zwischen Haupttrennstellen an Wortfugen, Nebentrennstellen erster Ordnung (die „normalen“ Trennstellen innerhalb von Wörtern) und Nebentrennstellen zweiter Ordnung, die zwar korrekt sind, aber unschön bzw. sinnentstellend sein können.

- Zeile 8: Mit der Kombination aller gefundenen Trennvarianten (Dateils dazu in [Kod01]) wird erreicht, dass nur sichere Trennstellen in das Ergebnis aufgenommen werden. Denn nur Trennstellen, die in allen Zerlegungsvarianten des Wortes gefunden wurden, können als sicher betrachtet werden und sollen sich dann schlussendlich im fertig kombinierten Trennvektor wiederfinden.

- Zeile 13: Als Präfixatome werden alle diejenigen Atome bezeichnet, die die bisherige Zerlegung bis *word_pos-1* fortsetzen können, also Präfixe von dem noch zu untersuchenden Wortteil ab *word_pos* sind. Bei dem – fiktiven – Wort *abcdef* muss also bei *word_pos=3* (*c*) in der Atomtabelle und eventuell vorhandenen Benutzer-Atomtabellen nach den Atomen *cdef*, *cde*, *cd* und *c* gesucht werden. Werden entsprechende Atome gefunden, dann werden diese, nach ihrer Länge absteigend sortiert, in einer Liste dem Wortzerlegungsalgorithmus übergeben.
  Werden mehrere Atomtabellen verwendet, so kann es an dieser Stelle zu einer weiteren Komplikation kommen: Wie sollen aus verschiedenen Atomtabellen stammende, gleiche Atome (z.B. *cde*) mit unterschiedlichen Attributen (u.a. Atomklassen) behandelt werden?
  Prinzipiell gibt es hier drei Möglichkeiten:

  1. Gleiche Atome in verschiedenen Atomtabellen nicht zulassen.

2. In irgendeiner Form Prioritäten festlegen, welches Atom schlussendlich bei der Wortanalyse verwendet werden soll.

3. Die Attribute gleicher Atome miteinander vereinigen.

Die erste Variante klingt zunächst am einfachsten, entpuppt sich in der Praxis aber als schwer durchsetzbar. Die Standard-Atomtabelle von SiSiSi wird laufend ergänzt und erweitert. Diese Verbesserungen sollen natürlich auch dem Anwender zugute kommen. Wie soll aber das Prozedere aussehen, wenn ein bisher in der Standard-Atomtabelle fehlender Stamm, der in eine Benutzer-Atomtabelle aufgenommen wurde, nun plötzlich nach einer Aktualisierung doch vorhanden ist? Sollen die vom Benutzer selbst spezifizierten Daten einfach gelöscht werden? In diesem Zusammenhang stellen sich noch einige weitere Fragen, die nicht wirklich zufriedenstellend für den Benutzer gelöst werden können.
Auch die zweite Variante erscheint vorerst relativ einsichtig und trivial in der Implementierung, wirft aber ebenfalls einige Fragen auf. Soll die Standard-Atomtabelle Priorität haben oder die Benutzer-Atomtabelle(n)? Für den ersten Fall spricht, dass Atome in der Standard-Atomtabelle deutlich genauer klassifiziert sind. Einem normalen Benutzer ist es unmöglich zumutbar, einem selbst eingetragenen Stamm eine oder mehrere der aktuell rund 250 Atomklassen zuzuordnen. Das dafür notwendige Wissen besitzen neben Germanisten wohl nur wenige Personen. Im zweiten Fall gibt man dem Benutzer die durchaus interessante Möglichkeit, einen vielleicht in der Standard-Atomtabelle gefundenen Fehler selbst durch die Aufnahme eines eigenen Atoms zu korrigieren. Aber auch hier ist das Problem die ungenaue bzw. auch ungenügende Klassifikation der neuen Stämme durch den Benutzer. Wenn z.B. eine wichtige Endung wie *en* plötzlich nicht mehr als solche klassifiziert ist, dann kann als Folge daraus kein Wort mit dieser Endung mehr korrekt zerlegt werden.
Damit ist die einzig sichere Alternative die Vereinigung der Attribute aller gefundenen, gleichen Atome. So bleiben die genauen Klassifikationen der Standard-Atomtabelle erhalten, womit auch alle Zerlegungen, die ohne die Verwendung einer Benutzer-Atomtabelle gefunden worden wären, weiterhin erzeugt werden und nicht verloren gehen. Andererseits bietet sich dem Benutzer, wenn er einen Fehler in der Standard-Atomtabelle entdeckt, mit der Aufnahme eines bereits existierenden Atoms in seine Atomtabelle die Möglichkeit, zwar keine Fehler zu verhindern, aber zusätzliche, korrekte Zerlegungen zu generieren.

- Zeilen 14 bis 16: Ziel von SiSiSi ist es, im Gegensatz zu den auf riesigen Wörterbüchern basierenden Methoden, mit einer möglichst kleinen Atomtabelle das Auslangen zu finden. Mit ihrer Hilfe und derer der Wortbildungsgrammatik des Regelautomaten sollen alle Wörter der deutschen Sprache möglichst vollständig abgedeckt werden. Lange Atome werden daher nur dann in die Atomtabelle aufgenommen, wenn Wortbestandteile entweder nicht

aus kürzeren Atomen zusammengesetzt werden können (ein eigenständiger Stamm) oder wenn es bei der Zusammensetzung zu Fehlern kommt, die durch ein eigens aufgenommenes Atom verhindert werden sollen.
Das bedeutet, dass längere Präfixatome Priorität haben müssen, die Liste also vom längsten zum kürzesten Atom hin abgearbeitet wird. Auch ist es nicht erwünscht, dass ein bereits behandeltes Atom durch zwei oder mehrere andere Atome zusammengesetzt wird. Hierfür wird der „Sperrvektor“ verwendet. In diesem wird für einen Rekursionszweig (alle Wortzerlegungen ab *word_pos*) die Stelle markiert, an der die Grenze eines bearbeiteten Atoms zu liegen gekommen ist. Würde im weiteren Verlauf der Rekursionen ab *word_pos* ein anderes Atom mit seiner Grenze an diese Stelle gelangen, so wird die weitere Bearbeitung durch die Abfrage in Zeile 15 verhindert.
Als Beispiel soll wieder das fiktive Wort *abcdef* mit *word_pos=3* (*c*) dienen. Wird in der Atomtabelle ein Atom *cde* gefunden und bearbeitet, dann wird im Sperrvektor für diesen Rekursionszweig die Position *word_pos+length(cde)=6* markiert. Danach ist eine Zerlegung z.B. in *c* und (Rekursionsschritt) *de* nicht mehr möglich (*word_pos+1+2* ergibt wieder *6*, diese Zerlegung ist aber laut Sperrvektor verboten; keine Atomgrenze dieses Rekursionszweiges darf mehr auf dieser Position zu liegen kommen). Für einen anderen Rekursionszweig gilt dieser Sperrvektor aber nicht, d.h. bei *word_pos=2* (*b*) ist eine Zerlegung in z.B. *bcd* und *e* (*word_pos+3+1=6*) weiterhin gültig!
Doch keine Regel ohne Ausnahmen. Es gibt durchaus Atome, wo es notwendig und daher gewünscht ist, dass auch Zerlegungsvarianten, bei denen dieses Atom aus anderen Atomen zusammengesetzt wird, weiter untersucht werden. Ein Beispiel wäre hier das Atom *steil*, das für das Wort *Bekleidungsteil* auch aus *s* (Fugenzeichen) und *teil* gebildet werden können muss. Diese Atome werden in der Atomtabelle speziell gekennzeichnet, in Zeile 16 unterbleibt dann das Setzen der Markierung im Sperrvektor.
Und auch für die Bearbeitungsreihenfolge der Atome gibt es eine Ausnahme, dann nämlich, wenn *state* den Zustand nach einem unvollständigen Stamm beschreibt. Ein unvollständiger (oder auch reduzierter) Stamm kann nicht alleine stehen, z.B. *reb*, sondern benötigt im Normalfall noch ein Suffix, hier also z.B. ein *e* für die *Rebe*. Dabei sind alle möglichen Zerlegungen mit Suffixen gewünscht, ohne dass Varianten durch den Sperrvektor unterbunden werden. Um das zu erreichen, wird die Liste der Präfixatome hier vom kürzesten zum längsten Eintrag hin durchlaufen. Ist die Suche nach passenden Suffixen abgeschlossen, muss die Liste der Präfixatome aber nochmals, jetzt in der Gegenrichtung (also vom längsten zum kürzesten Präfixatom), ganz normal abgearbeitet werden. Das ist notwendig, um auch Wortzusammensetzungen zu finden, bei denen nicht unbedingt ein Suffix nach dem reduzierten Stamm folgen muss, z.B. das *Rebhuhn* oder die *Reblaus*.

- Zeile 17: SiSiSi unterscheidet momentan vier verschiedene Typen von Ausnahmen: Ausnahmetrennstelle(n) mit einer Haupttrennstelle oder einer Nebentrennstelle (siehe auch Erläuterungen zu Zeile 4), Ausnahmeendungen

und untrennbare Wortteile (siehe [Kod01]). Je nach Ausnahmetyp müssen nun an der oder den entsprechenden Stellen Trennzeichen eingefügt bzw. müssen Wortteile vor einer möglichen Bearbeitung durch den Trennautomaten (siehe Zeile 3) geschützt werden, da sie untrennbar sind.

- Zeilen 20 und 21: Nicht jede Atomklasse *atom_class* eines Atoms darf, selbst wenn eine entsprechende Regel *(state,atom_class)* → *Zielzustand* im Regelautomaten definiert ist, auch wirklich immer ohne Vorbehalte verwendet werden. Manche Atome fordern für eine bestimmte Verwendung eine Vorbedingung (eine andere Atomklasse). So kann das Atom *arzt* als Substantiv (der *Arzt*) immer verwendet werden (wenn es bei der Wortzerlegung gefunden wird), als Verb benötigt das Atom aber für eine korrekte Wortbildung eine Vorsilbe (z.B. jemanden *verarzten*).
  Während der Wortzerlegung wird daher eine Liste (*used_atom_classes*) aller vom Wortanfang bis *word_pos* verwendeten Atomklassen mitgeführt. Handelt es sich nun bei *atom_class* um eine bedingte Atomklasse, so wird *used_atom_classes* nach dieser Vorbedingung durchsucht. Wird die benötigte Atomklasse nicht gefunden, dann darf *atom_class* bei dieser Zerlegung nicht verwendet werden, und die nächste Atomklasse des Atoms *atom* wird bearbeitet. Wird die als Vorbedingung benötigte Atomklasse gefunden, so darf *atom_class* verwendet werden. Die Vorbedingung muss danach allerdings aus *used_atom_classes* gelöscht werden, da eine Atomklasse nur einmal in einem Rekursionszweig als Vorbedingung genutzt werden darf.

- Zeilen 26 bis 30: Der Wortzerlegungsalgorithmus greift bei seiner Arbeit auf eine doch recht beachtliche Anzahl von für die Rekursion globalen Variablen zurück (hauptsächlich aus Effizienzgründen, da sonst bei jedem Rekursionsschritt u.a. massenweise Arrays umkopiert werden müssten). Darunter fallen z.B. der kombinierte Trennvektor aus Zeile 8, der Sperrvektor aus den Zeilen 15 und 16 oder auch die Liste *used_atom_classes* aus den Zeilen 20 und 21. Das führt allerdings dazu, dass nach Beendigung der aktuellen Rekursionsstufe Veränderungen an diesen globalen Variablen wieder rückgängig gemacht werden müssen.
  Als Beispiel soll hier die Liste der bisher verwendeten Atomklassen, *used_atom_classes*, dienen. Die in Zeile 21 eingefügte Atomklasse *atom_class* muss, nachdem alle möglichen Zielzustände inklusive Rekursionen durchlaufen wurden, zwischen den Zeilen 26 und 27 wieder aus der Liste gelöscht werden, denn diese Atomklasse ist nun fertig bearbeitet. Hingegen muss vielleicht eine Atomklasse, die in Zeile 20 als Vorbedingung für *atom_class* benötigt und daher an dieser Stelle aus der Liste gelöscht wurde, erneut in *used_atom_classes* eingefügt werden, damit sie anderen Atomklassen wieder als Vorbedingung dienen kann.
  Weitere Beispiele wären durch Regeln oder durch Ausnahmen gesetzte Trennstellen, Markierungen im Sperrvektor usw.

- Keinen Niederschlag im Pseudocode gefunden hat auch eine durch die Rechtschreibreform 1998 erzwungene Doppelgleisigkeit in fast allen Berei-

chen des Wortzerlegungsalgorithmus. Denn bei ins Deutsche übernommenen Fremdwörtern oder nicht mehr als Zusammensetzungen erkennbaren Wortkombinationen ist es nun möglich, sie nicht nur nach Herkunft (meist mit einer Ausnahmetrennstelle), sondern auch nach Sprechsilben zu trennen (z.B. *Magnet*: *Ma-gnet* nach Herkunft, *Mag-net* nach Sprechsilben; *dar-in* nach Herkunft, *da-rin* nach Sprechsilben). Beides sind nun korrekte Trennvarianten, die ab dem Zeitpunkt, an dem eine solche spezielle Ausnahmetrennstelle gefunden wurde, im Algorithmus mitgeführt werden müssen. Weitere Trennstellen, wie sie z.B. in Zeile 24 durch den Regelautomaten gesetzt werden, müssen ab dann in zwei Trennvektoren eingetragen werden, in einem nach Herkunft und einem nach Sprechsilben.

Die Implementierung des Wortzerlegungsalgorithmus in Java-SiSiSi entspricht weitestgehend der in der Delphi-Version. Neben unvermeidlichen Anpassungen an die verwendete Programmiersprache sind die wesentlichsten Unterschiede:

- Delphi-SiSiSi erzeugt während der Wortanalyse eine Vielzahl von Testinformationen, die benötigt werden, um eine Wortzerlegung bis ins kleinste Detail nachvollziehen zu können. Dieser doch erhebliche Mehraufwand ist für die Wartung der Atomtabelle und des Regelautomaten, aber auch für den Zerlegungsalgorithmus selbst während der Weiterentwicklung von SiSiSi notwendig, für die übliche Anwendung aber unnötig. Java-SiSiSi beschränkt sich daher auf die absolut notwendigen Dinge für die geforderte Funktionalität. Das Laufzeitverhalten kann dadurch deutlich verbessert werden.

- Es existieren Zustände im Regelautomaten der Wortbildungsgrammatik, die in keiner einzigen Regel als Ausgangszustand definiert sind, d.h. aus diesen Zuständen gibt es kein Entkommen, nicht einmal einen Übergang in sich selbst. Solche Zustände dürfen natürlich nur am Ende des zu analysierenden Wortes erreicht werden; mitten im Wort führen sie zwangsläufig zu einer unvollständigen und damit ungültigen Zerlegung. Java-SiSiSi prüft nun während der Wortanalyse auf diese Situation und erspart sich in einem solchen Fall einen kompletten Rekursionsschritt inklusive Zugriff auf die Atomtabelle (mit Durchlaufen aller Atomklassen der gefundenen Atome) und den Regelautomaten.

- Für ein am Institut laufendes Praktikum, eine Client-Server-Webapplikation zur Volltextsuche, wurden erweiterte Abfragefunktionen zur Wortanalyse implementiert. In diesem Fall ist es notwendig, die bei der Wortzerlegung erzeugten Atome zu kennen, eine Information, die bei der Silbentrennung nicht ausgegeben werden musste.

- Eine Folge des kontinuierlichen Wachstumsprozesses von SiSiSi ist, dass der Algorithmus und die verwendeten Datenstrukturen nur ungenügend voneinander getrennt sind, d.h. der Wortzerlegungsalgorithmus greift sehr direkt auf Atomtabelle und Regelautomat zu und ist damit an eine spezielle

Implementierung dieser Datenstrukturen gebunden. Java-SiSiSi führt hier saubere Schnittstellen zwischen Algorithmus und Datenstrukturen ein.

Kapitel 3

Die Datenstrukturen

Während der Wortanalyse benötigt der Wortzerlegungsalgorithmus Zugriff auf weitere Komponenten von SiSiSi, die bereits kurz vorgestellt wurden: Die Atomtabelle, den Regelautomaten für die deutsche Wortbildungsgrammatik und den Automaten für die Dudentrennung (Trennung nach Vokal-Konsonanten-Folgen).

Aus Sicht der Datenstruktur ist der Automat für die Dudentrennung nicht weiter spektakulär. Es handelt sich dabei um einen fest implementierten, endlichen Automaten mit nur fünf Zuständen, der zunächst manche Buchstabenfolgen zu Lauten zusammenfasst (z.B. *eu* oder *sch*) und danach die bei der Wortzerlegung gefundenen Wortteile nach Sprechsilben trennt. Nähere Details zu diesem Automaten finden sich in [Kod01].

Für die Effizienz von SiSiSi viel entscheidender sind hier die Datenstrukturen für die Atomtabelle und den Regelautomaten, nicht zuletzt auch deswegen, da auf sie während der Wortzerlegung bei jedem Rekursionsschritt wiederholt zugegriffen wird (im Gegensatz zur Dudentrennung, die nur am Ende eines Rekursionszweiges, wenn eine gültige Zerlegung gefunden wurde, aufgerufen wird).

Dieses Kapitel beschäftigt sich nun eingehend mit den Datenstrukturen für die Atomtabelle und den Regelautomaten, beschreibt Designentscheidungen der bisherigen Delphi-Implementierung und stellt die Neuerungen und Verbesserungen in Java-SiSiSi vor.

3.1 Die Atomtabelle

Bekanntermaßen ist das primäre Ziel von SiSiSi, ein Eingabewort genau zu analysieren und in seine einzelnen Bestandteile wie Vorsilbe(n), Stämme oder Endung(en) zu zerlegen. Dazu ist es aber notwendig zu wissen, aus welchen kleinsten syntaktischen Teilen (Atomen) deutsche Wörter gebildet werden können. SiSiSi verwendet zu diesem Zweck die so genannte Atomtabelle, in der genau diese Wortteile und alle für die Wortzerlegung notwendigen Zusatzinformationen gespeichert werden. Weitere Anforderungen an die Atomtabelle sind:

- Das effiziente Auffinden aller gespeicherten Atome, die Präfix eines Wortes bzw. Wortteiles sind. Die gefundenen Präfixe müssen dabei der Länge nach sortiert dem Wortzerlegungsalgorithmus übergeben werden.
- Der Speicherplatzbedarf soll möglichst gering sein.
- Die Datenstruktur muss dynamisch erweiterbar sein, d.h. es muss gewährleistet bleiben, dass zu einem beliebigen Zeitpunkt noch weitere Atome in die Atomtabelle aufgenommen werden können.

3.1.1 Atome und deren Zusatzinformationen

Bevor nun näher auf die Atomtabelle eingegangen wird, folgt ein kurzer Blick auf die für die Speicherung eines Atoms verwendete Datenstruktur.

Neben dem eigentlichen Atom-String (Wortbestandteil), der das Atom definiert, und dessen Länge werden noch zusätzliche Daten vom Wortzerlegungsalgorithmus benötigt und daher bei einem Atom gespeichert (Atom-Attribute): ein Flag zur Kennzeichnung, ob ein Atom als unteilbar zu behandeln ist oder nicht, Informationen zu möglichen Ausnahmetrennstellen (Art, Position(en), reguläre Silbentrennung zusätzlich erlaubt oder nicht). Einzelheiten zur Bedeutung und Verwendung dieser Zusatzinformationen finden sich bei den Erläuterungen zum Pseudocode des Wortzerlegungsalgorithmus (2.4, Algorithmus 1).

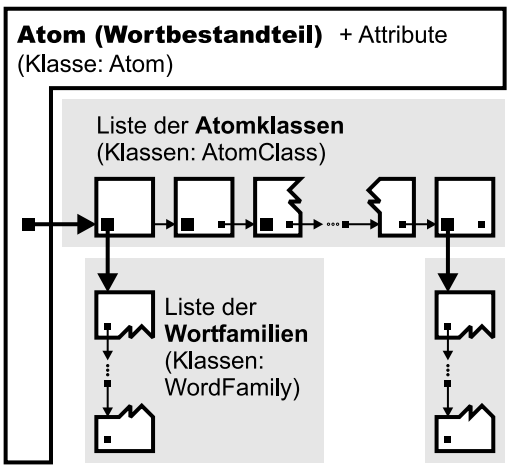

Abbildung 3.1: Datenstruktur zur Repräsentation eines Atoms

Ein wichtiger Bestandteil von SiSiSi, um möglichst viele sinnlose Wortzerlegungen unterdrücken zu können, ist die Klassifikation von Atomen in Atomklassen, d.h. zu jedem Atom wird genau spezifiziert, welche Aufgaben es bei der deutschen Wortbildung übernehmen kann. So kann das Atom *haus* sowohl als Substantivstamm (das *Haus*) als auch als Verbstamm (*hausen*) verwendet werden mit entsprechenden Möglichkeiten bzw. Einschränkungen bezüglich Wortzusammensetzungen oder erlaubten Endungen. Es sind also für ein Atom mehrere Atomklassen zulässig, die in einer linearen Liste gespeichert werden (inklusive

möglicher Vorbedingungen, die eine Atomklasse fordern kann, z.B. das Vorhandensein einer bestimmten Vorsilbe). Eine Liste ist in diesem Fall effizient, da während der Wortanalyse immer alle Atomklassen eines Atoms der Reihe nach bearbeitet werden müssen.

Ähnliches gilt auch für die optionalen Wortfamilienlisten zu jeder Atomklasse. Für die Silbentrennung sind sie zwar nicht weiter von Interesse, werden aber für andere Einsatzgebiete von SiSiSi benötigt, beispielsweise für die in Arbeit befindliche Volltextsuche. So wird zum Atom *schwamm* bei der Verwendung als Verbstamm die Wortfamilie *schwimmen* gespeichert, als Substantivstamm aber die Wortfamilie *Schwamm*. Auch hier ist eine lineare Liste geeignet, da immer alle Wortfamilien einer Atomklasse sequentiell benötigt werden.

3.1.2 Frühere Datenstruktur für die Atomtabelle

Wie auch später beim Regelautomaten zu sehen sein wird, prägte eine Anforderung die Designentscheidungen in den Anfängen von SiSiSi: möglichst geringer Speicherplatzverbrauch. Das verwundert nicht weiter, wenn man bedenkt, dass die Entwicklung von SiSiSi nun schon fast 20 Jahre andauert und damals ein Rechner mit mehreren Megabyte Hauptspeicher nicht verfügbar war.

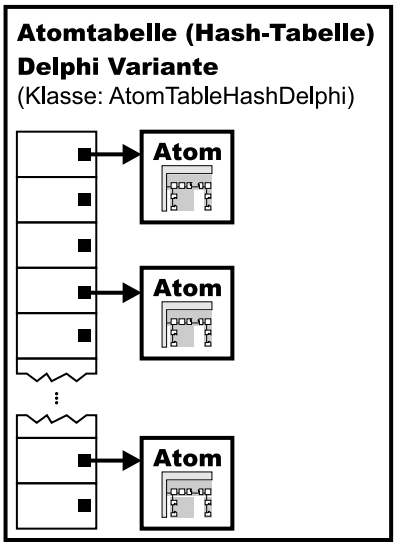

Abbildung 3.2: Atomtabelle als Hash-Tabelle mit innerer Verkettung

Als Kompromiss zwischen den zum Teil gegensätzlichen Anforderungen geringer Speicherplatzverbrauch, schneller Zugriff auf die gespeicherte Atome und dynamische Erweiterbarkeit der Atomtabelle wurde eine Hashtabelle mit innerer Verkettung (Double Hashing) als Datenstruktur gewählt. Hier wird auf Basis des Atom-Strings ein Index in die Hashtabelle berechnet, an dem das entsprechende Atom direkt abgelegt werden soll. Kommt es dabei zu einer Kollision mit einem bereits gespeicherten Atom, so wird eine zusätzliche Hashfunktion ausgewertet,

deren Wert nun wiederholt zum bisher bestimmten Index hinzuaddiert wird (modulo Hashtabellengröße), bis ein freier Platz zum Einfügen gefunden wurde.

Auch wenn heutzutage der Speicherplatzverbrauch bei weitem kein so großes Problem mehr darstellt wie vor 20 Jahren, vor allem in den Bereichen, in denen sich SiSiSi bewegt (rund zwei bis drei Megabyte), wurde doch diese Form der Atomtabelle mehr oder weniger unverändert bis hin zu Delphi-SiSiSi übernommen.

Das von Grund auf neu implementierte Java-SiSiSi bricht nun mit dieser Tradition und führt neue, besser den Anforderungen des Wortanalysealgorithmus angepasste und damit deutlich effizientere Datenstrukturen ein. Das sind die Hash-Tabelle mit äußerer Verkettung, die Hash-Hash-Tabelle und der Trie, die in den folgenden Abschnitten beschrieben werden. Alle diese Datenstrukturen wurden im Zuge dieser Diplomarbeit implementiert.

3.1.3 Hash-Tabelle mit äußerer Verkettung

Auch wenn man an einer Hash-Tabelle als zugrunde liegende Datenstruktur festhält, ist eine deutliche Verbesserung möglich. Lange Zeit wurden standardmäßig Hash-Tabellen mit innerer Verkettung verwendet. Doch es existiert eine Variante, die Hash-Tabelle mit äußerer Verkettung, die in einigen Bereichen deutlich angenehmere Eigenschaften besitzt.

Hierbei wird ein Array von Referenzen (Zeigern) angelegt. Wie bei der inneren Verkettung auch wird aus dem Atom-String mit Hilfe einer Hash-Funktion ein Index in die Hash-Tabelle berechnet. Allerdings werden nun alle Atome mit identischem Hash-Index auch genau bei diesem eingetragen, und zwar in linearen Listen, auf die die Referenzen der Hash-Tabelle verweisen. Auch andere Datenstrukturen als lineare Listen sind vorstellbar, aber aus Gründen der Effizienz sollte das primäre Ziel sein, so wenig Atome wie möglich unter einem Hash-Index speichern zu müssen. Das ist mit einer guten Hash-Funktion und einem entsprechenden Füllgrad auch zu erreichen (kaum mehr als ein bis zwei Atome pro Liste), womit sich weitere Überlegungen in diese Richtung erübrigen. Gegenüber einer Hash-Tabelle mit innerer Verkettung steigt auf Grund der für die linearen Listen benötigten Referenzen (Zeiger) der Speicherplatzverbrauch etwas, allerdings nur in unbedeutendem Ausmaß.

Während der Wortanalyse entscheidend für SiSiSi ist das Verhalten der Atomtabelle bei wiederholten Suchanfragen. Dabei ist die mittlere Anzahl von notwendigen Zugriffen bei der Suche nach einem in der Atomtabelle tatsächlich gespeicherten Atom bei der Hash-Tabelle mit äußerer Verkettung relativ gering. Dafür muss zunächst der Hash-Index berechnet werden und der Anfang der an diesem Index gespeicherten linearen Liste (wenn überhaupt vorhanden) geholt werden (erster Zugriff). Danach muss diese Liste im Mittel noch bis zur Hälfte durchsucht werden, bis das Atom gefunden ist. Mit einer guten Hash-Funktion, die die Atome gleichmäßig über die Hash-Tabelle verteilt, und einem Füllgrad $\alpha = \frac{n}{N}$ (n: Anzahl der eingefügten Atome; N: Größe der Hash-Tabelle) beträgt

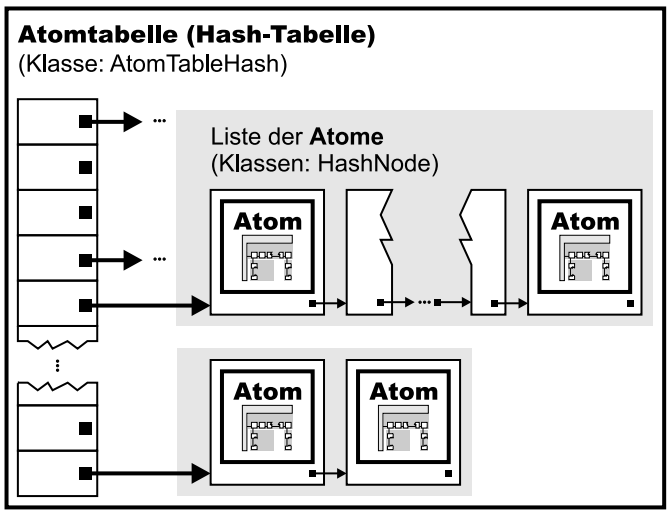

Abbildung 3.3: Atomtabelle als Hash-Tabelle mit äußerer Verkettung

die Länge der Listen im Durchschnitt α. Damit ergibt sich für eine Hash-Tabelle mit äußerer Verkettung die mittlere Anzahl von Zugriffen bei der Suche nach einem gespeicherten Atom zu $1 + \frac{\alpha}{2}$. Bei innerer Verkettung sind für die gleiche Aufgabe etwas mehr Zugriffe in die Hash-Tabelle notwendig, nämlich $\frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$.

Noch wichtiger ist diese Analyse aber für den Fall, dass der gesuchte Wortteil nicht als Atom in der Hash-Tabelle gespeichert ist. Denn während der Wortzerlegung benötigt der Algorithmus von der Atomtabelle immer alle Präfixatome eines Wortteiles. Der Hash-Tabelle bleibt in diesem Fall nichts anderes übrig, als alle Möglichkeiten durchzuprobieren, d.h. für einen Wortteil *abcd* muss nach den Atomen *a*, *ab*, *abc* und *abcd* gesucht werden. Verständlich, dass man bei diesen Anfragen eher selten wirklich auf ein Atom stößt. Messungen haben ein ungefähres Verhältnis zwischen erfolgloser und erfolgreicher Suche von 5:1 ergeben.

Für die Hash-Tabelle mit äußerer Verkettung erhöht sich in diesem Fall die mittlere Anzahl der Zugriffe nur unwesentlich. Um festzustellen, dass ein Wortteil nicht als Atom gespeichert ist, muss nur die entsprechende lineare Liste (wenn vorhanden) bis zu ihrem Ende durchlaufen werden. Bei einer durchschnittlichen Listenlänge von α ergibt sich die gesuchte Anzahl daher zu $1 + \alpha$.

Die Hash-Tabelle mit innerer Verkettung zeigt hier ein deutlich ungünstigeres Verhalten, denn hier weiß man erst dann, dass ein Atom nicht vorhanden ist, wenn man bei der Suche (Schritte wie beim Einfügen) auf einen freien Platz stößt. Dafür benötigt man im Mittel aber $\frac{1}{1-\alpha}$ Zugriffe. In Worten: Bei einem Füllgrad von 90% (wie er in Delphi-SiSiSi bis vor kurzem noch verwendet wurde) ist nur jeder zehnte Platz in der Hash-Tabelle frei, d.h. durchschnittlich sind 10 Zugriffe in die Hash-Tabelle notwendig, um festzustellen, dass ein gesuchter Wortteil nicht als Atom gespeichert ist. Bei gleichem Füllgrad benötigt die Hash-Tabelle mit äußerer Verkettung nur rund zwei Zugriffe.

Abbildung 3.4 zeigt einen Vergleich der beiden Hash-Tabellen-Varianten bei

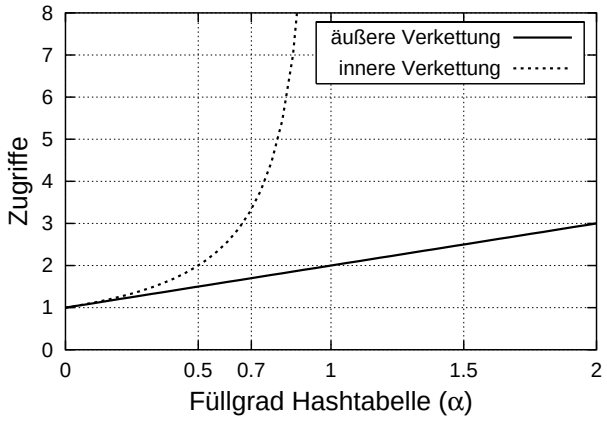

Abbildung 3.4: Mittlere Anzahl von Zugriffen nach Füllgrad der Atomtabelle bei der Suche nach einem nicht vorhandenen Atom

der Suche nach einem nicht als Atom gespeicherten Wortteil. Schon bei einem typischen Füllgrad von rund 70-75% ist ein deutlichen Unterschied zu erkennen, der mit wachsendem α stetig zunimmt. Bedingt durch ihren Aufbau kann eine Hash-Tabelle mit innerer Verkettung maximal N Atome aufnehmen, was einem Füllgrad von 100% entspricht. Zu diesem Zeitpunkt würde aber der oben beschriebene Algorithmus bei der Suche nach einem nicht vorhandenen Atom in eine Endlosschleife geraten, da keine freien Plätze mehr in der Hash-Tabelle existieren.

Länge der linearen Listen	*Füllgrad* *10%*		***70%***		*90%*	
0	75.780	(90,47%)	**5.944**	**(49.66%)**	3.797	(40.78%)
1	7.598	(9,07%)	**4.147**	**(34.65%)**	3.425	(36.78%)
2	371	(0,44%)	**1.484**	**(12.40%)**	1.472	(15.81%)
3	12	(0,01%)	**322**	**(2.69%)**	495	(5.32%)
4	0	(0,0%)	**66**	**(0.55%)**	93	(1.00%)
5	0	(0,0%)	**5**	**(0.04%)**	24	(0.26%)
6	0	(0,0%)	**1**	**(0.01%)**	5	(0.05%)

Tabelle 3.1: Anzahl der linearen Listen bestimmter Länge einer Hash-Tabelle mit äußerer Verkettung, abhängig vom Füllgrad der Atomtabelle

Die Hash-Tabelle mit äußerer Verkettung ist hier deutlich unempfindlicher gegenüber einem hohen Füllgrad, was auch die Statistik zur Länge der linearen Listen bei der Verwendung als Atomtabelle in SiSiSi zeigt (Stand Juli 2003, 8.376 eingefügte Atome). Damit ist auch bei einer dynamischen Erweiterung der Atomtabelle zur Laufzeit nicht unbedingt eine Reorganisation (Vergrößerung) der Hash-Tabelle notwendig, trotzdem bleiben die für das Laufzeitverhalten entschei-

denden linearen Listen relativ kurz.

3.1.4 Hash-Hash-Tabelle

Eine Variante der gerade vorgestellten Hash-Tabelle mit äußerer Verkettung, die in Java-SiSiSi ebenfalls implementiert wurde, ist die Hash-Hash-Tabelle. Der einzige Unterschied besteht dabei darin, dass statt des Atom-Strings S ein verdichteter Schlüssel $V(S)$ abgespeichert wird. Zur Kollisionsbehandlung werden ebenfalls lineare Listen verwendet, womit die Statistik über die Länge der einzelnen Listen aus Tabelle 3.1 also auch für die Hash-Hash-Tabelle ihre Gültigkeit hat. Der verdichtete Schlüssel ist vom berechneten Index $H(S)$ vollständig unabhängig, es werden hierfür zwei voneinander komplett getrennte Hash-Funktionen $H(S)$ und $V(S)$ verwendet. Auch ist dieser Schlüssel nicht mit dem Hash-Wert für das Double Hashing der Hash-Tabelle mit innerer Verkettung zu verwechseln. In der aktuellen Implementierung wird ein Atom-String auf eine Integer-Zahl mit 32 Bit abgebildet („verdichtet“).

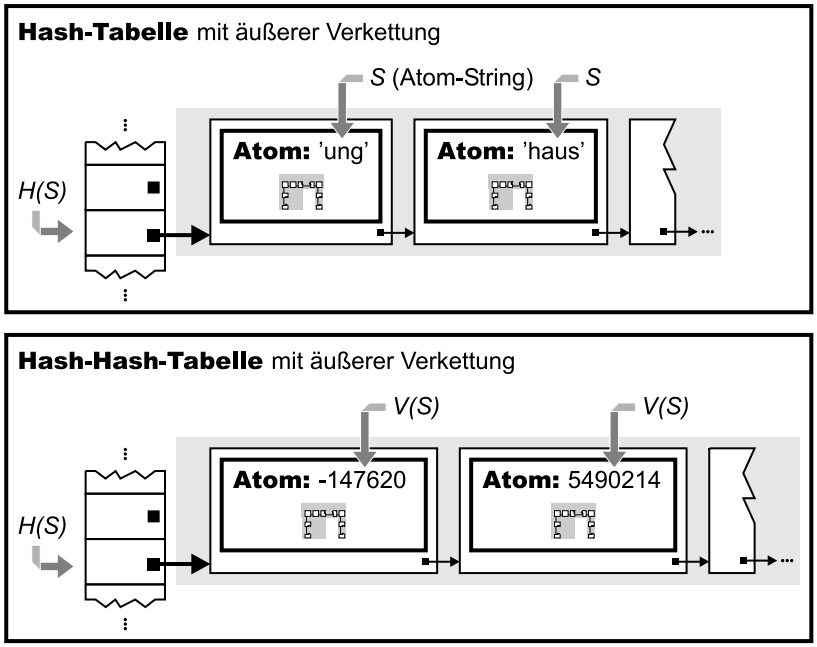

Abbildung 3.5: Vergleich Hash- und Hash-Hash-Tabelle mit äußerer Verkettung

Durch die Hash-Hash-Tabelle erreicht man eine erhebliche Speicherplatzersparnis. Denn im Mittel verbraucht ein verdichteter Schlüssel pro Atom deutlich weniger Speicherplatz als der Atom-String, zumal Java zur Kodierung von Zeichen den Unicode-Standard verwendet, d.h. es werden zwei Bytes pro Buchstabe benötigt.

Ein schöner Nebeneffekt, den Hash-Hash erreicht, ist die Verschlüsselung der Atomtabelle. Dabei entspricht die Berechnung des verdichteten Schlüssels einer

von der Kryptographie her bekannten Einwegfunktion: Die Berechnung der Hash-Funktion selbst ist einfach, die (eindeutige) Rückführung des Hash-Wertes zurück in den Atom-String ist aber (theoretisch und/oder praktisch) unmöglich. In die Atomtabelle ist über die Jahre hinweg enorm viel Arbeitszeit geflossen: das Eintragen von neuen Atomen, die genaue Klassifikation für den Regelautomaten, die Spezifikation von Ausnahmen und Wortfamilien, zahlreiche Tests, Wartung und vieles mehr. Es ist daher zumindest momentan nicht erwünscht, die Atomtabelle vollständig zu veröffentlichen.

Für den Aufbau der Hash-Hash-Tabelle sind die Atom-Strings unbedingt erforderlich, denn aus ihnen (den einzelnen Buchstaben) werden für jedes Atom sowohl der Index in die Hash-Tabelle als auch der verdichtete Schlüssel berechnet. Die Verschlüsselung ist daher nur dann gegeben, wenn die Atomtabelle nicht im Klartext ausgeliefert und erst bei einem Anwender aufgebaut werden muss, sondern bereits fertig, als Hash-Hash-Tabelle mit verdichteten Schlüsseln, vorliegt. Das hat zusätzlich den Vorteil, dass das Einlesen der Datenstruktur deutlich beschleunigt werden kann, da die aufwändigen Einfügeoperationen inklusive Berechnungen der verschiedenen Hash-Werte nicht mehr notwendig sind.

Für den Wortzerlegungsalgorithmus ist es irrelevant, dass in dieser Form der Atomtabelle die Atom-Strings selbst nicht mehr eingetragen sind. Die Schnittstelle zwischen diesen beiden Bestandteilen von SiSiSi ist so definiert, dass der Atomtabelle lediglich ein Wortteil übergeben wird, für das der Zerlegungsalgorithmus alle gefundenen Präfixatome inklusive der Zusatzinformationen auf einem Stack zurück erwartet. Die Atomtabelle ist nun selbst dafür verantwortlich, diesen Stack aufzubauen, eine Aufgabe, die auch die Hash-Hash-Tabelle erledigen kann.

Die Erwartungen bezüglich der Laufzeit des Wortzerlegungsalgorithmus mit einer Hash-Hash-Tabelle im Vergleich zu der normalen Hash-Tabelle mit äußerer Verkettung waren widersprüchlich. Bei der Suche nach Präfixatomen muss nun eine zusätzliche Hash-Funktion (basierend auf den einzelnen Buchstaben des zu untersuchenden Wortteiles) ausgewertet werden, dafür werden aber aufwändige String-Vergleiche durch einfache Vergleiche von zwei Integer-Zahlen ersetzt. Näheres zu diesem Thema später bei der Vorstellung der Ergebnisse.

Doch wie die Verringerung des Speicherplatzes schon andeutet, so geht mit der Abbildung der Atom-Strings auf 32-Bit-Schlüsselwerte eine Reduktion von Information einher. Es kann nicht mehr garantiert werden, dass zwei unterschiedlichen Atom-Strings (bzw. allgemeiner: Buchstabenfolgen) auch zwei unterschiedliche verdichtete Schlüssel zugewiesen werden.

Das kann durchaus zu Problemen führen: Möchte man z.B. bei der Untersuchung eines Wortes wissen, ob ein Wortteil S_1 als Atom in der Atomtabelle gespeichert ist und was seine Zusatzinformationen sind, so werden dazu zunächst der verdichtete Schlüssel $V_1(S_1)$ und der Index $H_1(S_1)$ berechnet. Angenommen, in der Hash-Hash-Tabelle existiere für ein Atom mit Atom-String S_2 ($S_1 \neq S_2$) die Eintragung mit verdichtetem Schlüssel $V_2(S_2)$ und Index $H_2(S_2)$, wobei zufällig $V_1 = V_2$ und $H_1 = H_2$ gilt, dann wird in der bei Index H_1 in der Hash-Hash-Tabelle gespeicherten linearen Liste nach dem verdichteten Schlüssel V_1 gesucht und

dieser wird auch gefunden. Es kann aber nicht mehr überprüft werden, ob auch wirklich $S_1 = S_2$ gilt, denn S_2 selbst wurde ja nicht gespeichert. Man muss sich also mit dem Vergleich der beiden verdichteten Schlüssel begnügen, die in diesem Fall übereinstimmen. Fälschlicherweise wird daraus geschlossen, dass der gesuchte Wortteil und der Atom-String identisch sind, und es werden die Zusatzinformationen von Atom S_2 bei der Suche nach S_1 ausgegeben, die im Allgemeinen falsch sein werden.

Diese Erscheinung wird als GAU bezeichnet, als größtes anzunehmendes Unglück. Aber es ist äußerst unwahrscheinlich, dass ein solches überhaupt eintritt: Die Wahrscheinlichkeit, dass zwei voneinander verschiedene Strings S_1 und S_2 sowohl in den verdichteten Schlüsseln $V_1(S_1)$ und $V_2(S_2)$, als auch in den Indizes $H_1(S_1)$ und $H_2(S_2)$ übereinstimmen, ist:

$$W = \frac{1}{P} \cdot \frac{1}{N}$$

(P: Wertebereich der verdichteten Schlüssel, hier also 2^{32}; N: Größe der Hash-Hash-Tabelle).

Diese Übereinstimmung kann aber mit jedem der ungefähr N in der Hash-Hash-Tabelle eingetragenen Atome auftreten, daher ergibt sich schlussendlich als Wahrscheinlichkeit für einen GAU:

$$W(\text{GAU}) = \left(\frac{1}{P} \cdot \frac{1}{N}\right) \cdot N = \frac{1}{P} \approx_{(P=2^{32})} 2.33 \cdot 10^{-10}$$

Es ist also relativ unwahrscheinlich, dass es zu einem solchen Problem kommt. Beim Aufbau der Hash-Hash-Tabelle aus den Klartextdaten der Atome kann ein GAU aber auf jeden Fall verhindert werden, da das Wissen ausgenützt werden kann, dass jedes Atom nur ein einziges Mal in der Atomtabelle vorkommen darf. Stößt man beim Eintragen eines Atoms mit verdichtetem Schlüssel $V(S)$ auf eben diesen in der entsprechenden linearen Liste, so liegt mit Sicherheit ein GAU vor. Durch Adaptierung der verwendeten Hash-Funktion $V(S)$ kann das Problem beseitigt werden, die Atomtabelle muss danach aber komplett neu eingelesen werden. Damit kann also sichergestellt werden, dass es nicht schon bei den eingetragenen Atomen zu einem GAU kommt. Während der Wortanalyse ist ein GAU allerdings, wie oben bereits gezeigt, nicht mehr erkennbar, in diesem Fall werden zu einem Wortteil Informationen eines falschen Atoms an den Wortzerlegungsalgorithmus geliefert.

Praktische Tests mit der aktuellen Atomtabelle haben jedenfalls bisher zu keinen sichtbaren Problemen geführt. Für diese Tests wurde eine Datei mit 209.500 verschiedenen, zum Teil sehr langen deutschen Wörtern verwendet, wobei während der Wortzerlegung für 879.020 unterschiedliche Wortteile überprüft werden musste, ob es sich hierbei um ein gespeichertes Atom handelt oder nicht. Diese Wörter wurden nun mittels SiSiSi einer Silbentrennung unterzogen, zunächst mit einer normalen Hash-Tabelle als Datenstruktur für die Atomtabelle. Das Ergebnis, die 209.500 Wörter mit allen ihren von SiSiSi gefundenen, potenziellen

Trennstellen, wurde als Referenzlösung abgespeichert. Die Silbentrennung wurde danach mit einer Hash-Hash-Tabelle als Atomtabelle wiederholt, das Ergebnis dieses Testlaufes war identisch mit der Referenzlösung. Für zusätzliche Tests wurde weiters die Hash-Funktion $V(S)$ für den verdichteten Schlüssel mehrmals variiert, doch auch hier traten bisher keine Fehler auf.

3.1.5 Trie

Hash-Tabellen sind im Allgemeinen äußerst gut dafür geeignet, schnell in großen Datenmengen zu suchen. Doch es gibt eine Datenstruktur, die darauf spezialisiert ist, Sequenzen (Wörter) von Zeichen (Buchstaben) über einem endlichen Alphabet (bei SiSiSi *a* bis *z*, die Umlaute *ä*, *ö* und *ü* sowie *ß*) zu speichern und darin effizient zu suchen, den Trie. Tries gibt es in verschiedenen Ausprägungen. Für SiSiSi von Interesse sind davon zwei Varianten, der Indexed Trie und der Linked Trie.

Indexed Trie

Bei einem Trie handelt es sich um eine Baumstruktur. Ein Pfad von der Wurzel des Tries in Richtung eines Blattes beschreibt eine Zeichensequenz und damit ein Wort. Jede Ebene des Tries entspricht dabei genau einer Position eines Zeichens in einem Wort, beginnend mit der Wurzel, die für den ersten Buchstaben steht.

Jeder Knoten des Indexed Tries enthält ein Array über das gesamte Alphabet, aus dem die Wörter gebildet werden. Dabei ist jedem einzelnen Buchstaben des Alphabets genau ein fester Index zugewiesen. In diesem Array werden zu jedem Buchstaben zwei Dinge gespeichert: Ein Flag, ob an dieser Stelle ein Wort zu Ende sein kann, und eine Referenz auf einen möglichen Nachfolgeknoten des Tries, der dann den nächsten Buchstaben eines gespeicherten Wortes enthalten würde.

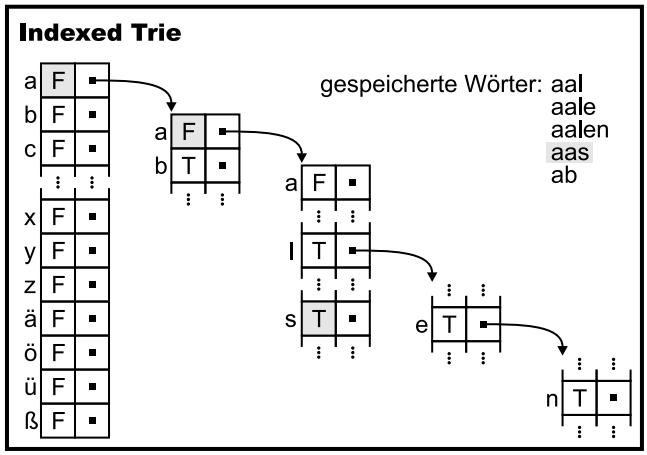

Abbildung 3.6: Aufbau eines Indexed Trie

Der in Abbildung 3.6 gezeigte kleine Ausschnitt eines Indexed Tries enthält bereits fünf verschiedene Wörter. Möglich wird das durch die implizite Präfix-Kompression des Tries, d.h. Wörter mit identischem Präfix liegen auf demselben Pfad durch den Trie, diese Informationen müssen daher nur ein einziges Mal gespeichert werden.

Möchte man nun verifizieren, ob ein bestimmtes Wort, z.B. das in der Abbildung hervorgehobene Wort *Aas*, im Trie gespeichert ist, sind folgende Schritte notwendig: Das Wort beginnt mit einem *a* (Groß-/Kleinschreibung wird vernachlässigt). Da *Aas* hier noch nicht zu Ende ist, wird von der Wurzel des Tries aus die Referenz weiterverfolgt, die für den Buchstaben *a* gespeichert ist (direkter Zugriff in das Array). Auch der zweite Buchstabe des Wortes ist ein *a*, und da hier das Wortende ebenfalls noch nicht erreicht ist, wird wieder der entsprechenden Referenz gefolgt. Der letzte Buchstabe ist nun ein *s*. Im aktuellen Trie-Knoten muss nun überprüft werden, ob bei diesem Buchstaben das Flag für ein gültiges Wortende gesetzt ist, was in diesem Beispiel der Fall ist (*T* für *true*).

Soll der Trie in SiSiSi als Atomtabelle eingesetzt werden, so ist es nicht genug zu wissen, dass ein bestimmter String als Atom vorhanden ist. Es müssen auch die für die Wortanalyse notwendigen Zusatzinformationen gespeichert werden. Das kann allerdings sehr einfach erreicht werden, indem das Flag für das Wortende durch eine Referenz auf ein Objekt mit diesen Zusatzinformationen eines Atoms ersetzt wird, wobei eine Null-Referenz einem *false* des Flags entspricht.

Die Vorteile für SiSiSi bei der Verwendung eines Tries liegen auf der Hand. Die wichtigste Aufgabe der Atomtabelle ist ja die Bereitstellung aller Präfixatome eines Wortteiles für den Wortzerlegungsalgorithmus. Die Hash-Tabellen müssen zu diesem Zweck alle Möglichkeiten einzeln ausprobieren, der Trie kann diese Atome mit nur einer einzigen Suche liefern. Denn es genügt, nach diesem Wortteil im Trie zu suchen und alle auf dem durchlaufenen Pfad liegenden Atome einzusammeln. Möchte man also z.B. in dem Trie aus Abbildung 3.6 alle Präfixe für das Wortteil *aalen* bestimmen, so findet man auf dem Weg von der Wurzel weg die Wörter *Aal*, *Aale* und *aalen* selbst. Weiters kann im Normalfall die Suche nach Präfixatomen bereits frühzeitig abgebrochen werden. So erreicht man im obigen Beispiel-Trie beim Wortteil *Aasfresser* nach *Aas* eine Sackgasse, aus *s* führt keine Referenz zu einem weiteren Knoten. Damit steht aber fest, dass keine weiteren Präfixe von *Aasfresser* im Trie existieren, die Suche kann daher an diesem Punkt gestoppt werden.

Doch auch wenn der Trie implizit eine Präfix-Kompression durchführt, ist das Hauptproblem des Indexed Tries der enorme Speicherplatzverbrauch. Denn vor allem gegen die Blätter des Tries hin sind die Knoten fast leer, enthalten nur noch für wenige oder gar nur einen einzigen Buchstaben Daten. Trotzdem wird in diesen Knoten der Speicher für das gesamte Array belegt.

Um dieses Problem in den Griff zu bekommen, sind zwei Verfahren sehr beliebt:

- *Suffix-Kompression:* Tries werden gerne eingesetzt, um ganze Wörterbücher zu speichern, z.B. für die Implementierung einer Rechtschreibprüfung. Doch

in jeder Sprache gibt es immer wiederkehrende Endungen, im Deutschen z.B. *ung* oder *lich*, die bei einem normalen Indexed Trie mehrfach gespeichert werden müssen. Ziel der Suffix-Kompression ist es nun, die Trie-Knoten für solche Endungen nur ein einziges Mal zu generieren und alle entsprechenden Wörter auf diese verweisen zu lassen, die Knoten also für mehr als nur ein einziges Wort zu verwenden.

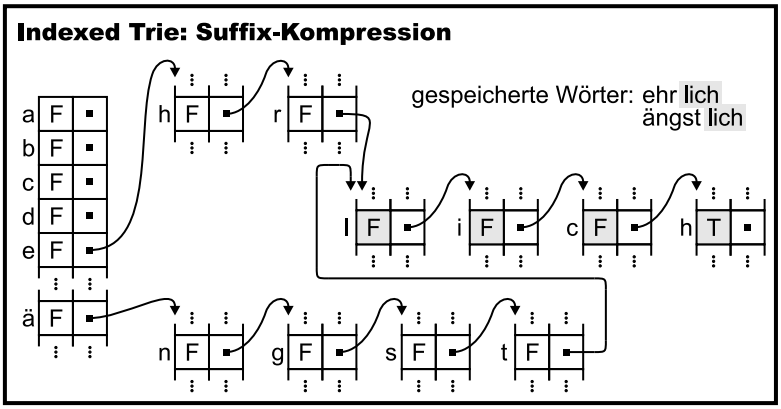

Abbildung 3.7: Suffix-Kompression für die Wörter *ehrlich* und *ängstlich*

Doch die zu erwartende Speicherplatzersparnis bei SiSiSi würde sich in sehr engen Grenzen bewegen, denn in der Atomtabelle werden nur die syntaktischen Grundbausteine deutscher Wörter gespeichert, häufig auftretende Endungen kommen daher auch nur ein Mal als Atom darin vor.

Entscheidender ist aber, dass die Suffix-Kompression eine für SiSiSi notwendige Eigenschaft des Tries zerstört, denn von der strikten Baumstruktur bleibt „nur“ noch ein allgemeiner, gerichteter Graph übrig. Das bedeutet, dass es zu einem Knoten des Tries nicht mehr unbedingt einen eindeutigen Pfad geben muss, sondern es können mehrere existieren. Verschiedene Pfade entsprechen aber unterschiedlichen Zeichenfolgen, also unterschiedlichen Wörtern bzw. Atomen. Im Normalfall stellt das kein Problem dar, da im Allgemeinen ein Trie nur die Frage zu beantworten hat, ob ein entsprechendes Wort darin gespeichert ist oder nicht, und die Gültigkeit einer Wortende-Markierung nicht vom zurückgelegten Pfad durch den Trie abhängig ist. Doch für SiSiSi ist das inakzeptabel, da hier der Trie Zusatzinformationen zu einem Atom aufnehmen muss, die sich natürlich von Atom zu Atom unterscheiden, und daher verschiedene Pfade nicht auf denselben Knoten mit identischen Daten zusammenlaufen dürfen.

Der Einsatz einer Suffix-Kompression kommt also für SiSiSi nicht in Frage.

- *Packed Trie:* Die Strategie des Packed Tries ist es, die einzelnen Knoten eines Indexed Tries, die man sich auch als Puzzleteile vorstellen kann, so ineinander zu verzahnen, dass die bei einem Knoten mit Daten gefüllten Stellen (das Wortende-Flag ist gesetzt und/oder eine Referenz auf einen

Nachfolgeknoten existiert) dort zu liegen kommen, wo ein anderer Knoten freie Plätze besitzt.

Logisch betrachtet bleibt der Trie weiterhin ein Baum, gespeichert wird er jetzt aber in einem einzigen, großen Array. Statt der Referenzen auf weitere Trie-Knoten werden die Basisindizes der Nachfolgeknoten verwendet, d.h. die Indizes innerhalb des Arrays, an denen jeweils der erste Buchstabe eines Knotens (in SiSiSi das *a*) zu liegen kommt. Zusätzlich wird noch zu jedem belegten Index des Arrays der Buchstabe benötigt, für den die an dieser Stelle gespeicherten Daten gültig sind. Das ist notwendig, um bei ineinander verwobenen Knoten entscheiden zu können, zu welchem dieser Knoten die Daten gehören.

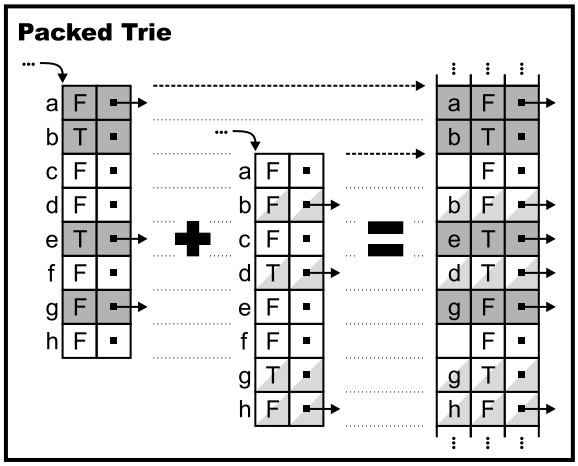

Abbildung 3.8: Aufbauprinzip eines Packed Tries (reduziertes Alphabet $[a \ldots h]$)

Sucht man also in einem Trie-Knoten nach den Daten zu dem Buchstaben *d*, findet aber an der Position Basisindex plus Index für den Buchstaben *d* z.B. ein *b* vor, so weiß man, dass im aktuellen Knoten keine Informationen für *d* vorhanden sind (siehe Abbildung 3.8). Damit dieses Verfahren aber eindeutig funktioniert, muss gewährleistet werden, dass niemals zwei Knoten derselbe Basisindex zugewiesen wird [Lia83].

Insgesamt gesehen ist ein Packed Trie aber eine eher statische Datenstruktur, Änderungen sind nur mehr schwer durchführbar, wenn er einmal aufgebaut ist. Zwar gibt es Möglichkeiten, einen Packed Trie dynamisch zu erweitern, wie von SiSiSi als – wenn auch nicht besonders dringliche – Bedingung für die Atomtabelle gefordert, doch der dafür zu treibende Aufwand ist enorm (um effizient freie Plätze im Array suchen zu können ist eine eigene Datenstruktur zu deren Verwaltung notwendig; sollte das Array zu klein für die zusätzlich benötigten Trie-Knoten sein, so muss es vergrößert/kopiert werden; etc.).

Ein Packed Trie wäre daher nur beschränkt für den Einsatz in SiSiSi geeignet.

Die Datenstruktur des Indexed Tries ist daher als Atomtabelle für SiSiSi geradezu ideal bei der Suche nach Präfixatomen, doch wegen des hohen Speicherplatzverbrauches in dieser Form leider nicht einsetzbar.

Linked Trie

Der Aufbau eines Linked Tries entspricht größtenteils dem eines Indexed Tries, lediglich die Speicherung der Daten innerhalb eines Trie-Knotens unterscheidet sich. Statt eines Arrays mit fester Größe (Anzahl der Zeichen des Alphabets, aus dem die zu speichernden Wörter kommen) wird hier eine dynamische Datenstruktur verwendet, etwa eine lineare Liste oder ein (balancierter) Suchbaum, womit nur noch Buchstaben gespeichert werden müssen, zu denen auch wirklich Daten existieren.

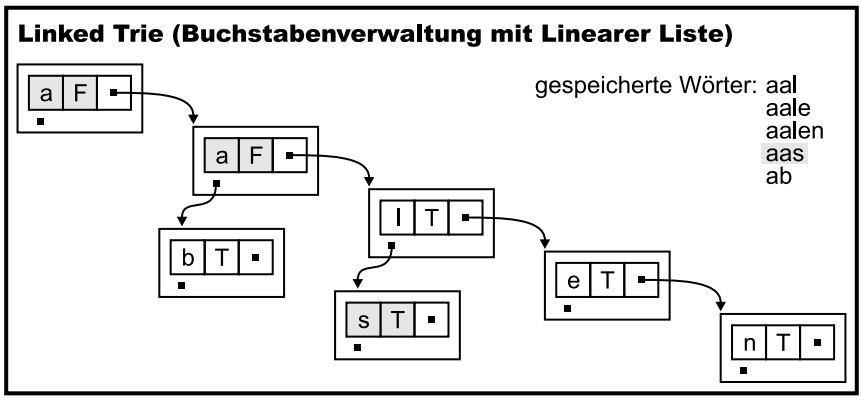

Abbildung 3.9: Trie aus Abbildung 3.6 als Linked Trie

Beim Indexed Trie ist durch die Position im Array eindeutig bestimmt, welchem Zeichen die dort abgelegten Daten zuzuordnen sind. Das funktioniert nun beim Linked Trie nicht mehr, zur Wortendemarkierung und Referenz auf einen möglichen Trie-Nachfolgeknoten muss der entsprechende Buchstabe extra vermerkt werden. Weiters sind zusätzliche Referenzen für den Aufbau der dynamischen Datenstruktur notwendig. Insgesamt kann aber trotzdem mit einem Linked Trie der Speicherplatzverbrauch gegenüber einem Indexed Trie deutlich verringert werden.

Allerdings verschenkt man den Geschwindigkeitsvorteil des direkten Zugriffs auf die Daten zu einem bestimmten Buchstaben (konstanter Aufwand) beim Indexed Trie, denn dieser muss nun erst in der dynamischen Datenstruktur gesucht werden (meist linearer bzw. logarithmischer Aufwand).

Hybrider Trie in SiSiSi

Eigentlich ist der Indexed Trie die ideale Datenstruktur für die Atomtabelle, wäre da nicht der enorm hohe Speicherplatzverbrauch. Die in der Literatur zur Lösung

dieses Dilemmas häufig vorgeschlagene Suffix-Kompression und der Packed Trie sind für SiSiSi, wie bereits gezeigt, nicht anwendbar bzw. nicht praktikabel. Geht man aber davon ab, ausschließlich einen Indexed Trie verwenden zu wollen, kann man dessen Stärken nutzen und versuchen, die Schwächen zu vermeiden.

Die Wurzel des Tries und die Knoten in deren Nähe haben im Normalfall einen relativ hohen Füllgrad, d.h. zu vielen der Buchstaben sind auch wirklich Daten vorhanden, die Speicherplatzverschwendung eines Indexed Tries mit seinem konstant großen Array ist hier also minimal. Anders ist die Situation, wenn man sich in der Umgebung der Blätter befindet. Dort sind in den Trie-Knoten die meisten Plätze der Arrays für die einzelnen Buchstaben unbesetzt.

Knuth schlägt nun z.B. in [Knu98] vor, für die ersten Buchstaben der zu speichernden Wörter einen normalen Indexed Trie zu erzeugen. Sobald aber ein Teilbaum des Tries nur noch für die Unterscheidung einiger weniger Wörter zuständig ist, wird dieser Teilbaum nicht mehr als Trie mit Knoten für jeden einzelnen Buchstaben ausgeführt (bei längeren Wörtern werden so relativ lange Listen von Trie-Knoten erzeugt, jeder Buchstabe bedeutet eine neue Ebene im Trie), sondern diese Wörter werden in einer anderen Datenstruktur, etwa einer linearen Liste oder einem Suchbaum, abgelegt. D.h. man erhält einen Indexed Trie, dessen Blätter Container mit mehreren darin enthaltenen Wörtern sind. Das entspricht im Wesentlichen der Methode, mit der ein Mensch in einem Wörterbuch nachschlägt: Zunächst gleicht man die ersten Buchstaben ab, bis man zu einem Punkt kommt, wo nur noch eine kleine, überschaubare Menge in Frage kommender Wörter vorhanden ist, die man dann eventuell sequentiell durchläuft oder mit Hilfe eines anderen Suchverfahrens abarbeitet.

In SiSiSi gibt es das Problem der langen Buchstabenketten nicht wirklich, da die zu speichernden Atome im Regelfall eher kurz sind. Weiters sucht man ja auch nicht nach nur einem Wort, sondern nach allen Präfixatomen für einen Wortteil, und diese Atome möchte man zusätzlich der Länge nach sortiert vorfinden. Setzt man als Blätter des Tries andere Datenstrukturen ein, würde man auf die Vorteile des Tries zu einem großen Teil verzichten.

Java-SiSiSi verwendet nun als weitere Variante für die Atomtabelle eine Mischung aus Indexed und Linked Trie. Die ersten beiden Ebenen (sie enthalten die ersten beiden Buchstaben eines Atoms) sind dabei fest als eine Adaptierung des Indexed Tries implementiert, die übrigen Knoten sind die eines Linked Tries.

Die Indexed-Trie-Komponente besteht aus einem einzigen großen Array für alle möglichen Kombinationen von zwei Buchstaben, mit denen Atome beginnen können, d.h. in diesem einen Array werden die ersten beiden Ebenen eines Indexed Tries zusammengefasst. Etwas Speicherplatz kann gespart werden, da keine syntaktischen Bestandteile der deutschen Sprache mit einem „ß“ beginnen. Andererseits muss berücksichtigt werden, dass es auch Atome mit nur einem einzigen Buchstaben gibt (z.B. einfache Endungen oder Fugenzeichen), das Alphabet muss also für den zweiten Buchstaben um das Leerzeichen „␣“ erweitert werden.

Neben den für SiSiSi statt des Flags zur Markierung eines Wortendes notwendigen Verweisen auf etwaige Zusatzinformationen zu einem Atom werden in

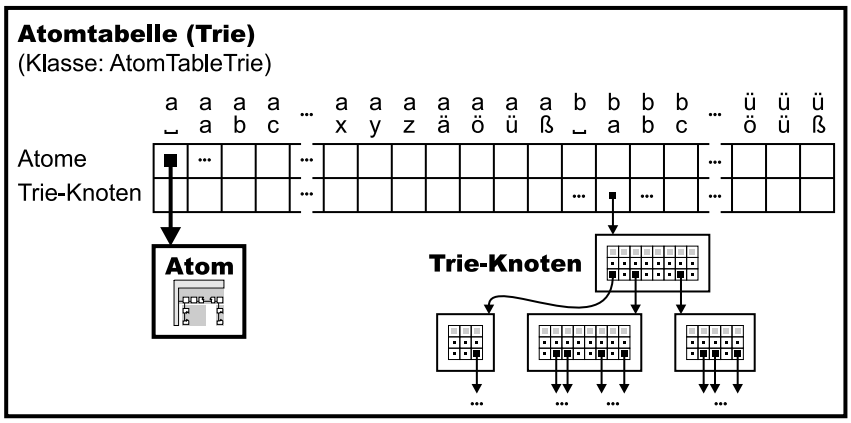

Abbildung 3.10: Atomtabelle als Kombination von Indexed und Linked Trie

diesem Array, für Atome mit mehr als zwei Buchstaben, auch Referenzen auf weitere Trie-Knoten gespeichert, der Beginn der Linked-Trie-Komponente.

Innerhalb dieser Knoten werden die Daten ebenfalls in ein Array geschrieben, was Vorteile beim Zugriff und beim Speicherplatzverbrauch (Referenzen für dynamische Datenstrukturen wie lineare Listen entfallen) mit sich bringt. Es handelt sich aber um ein dynamisches Array, d.h. es enthält nur Platz für die gerade gespeicherten Informationen und wächst bei Bedarf automatisch, wenn also ein neuer Buchstabe in diesen Trie-Knoten aufgenommen werden soll. Das Einfügen wird damit aufwändiger als bei der Verwendung anderer dynamischer Datenstrukturen. Das kann aber durchaus in Kauf genommen werden, da die Atomtabelle zur Laufzeit nur selten erweitert werden muss und auch der Trie – wie die Hash-Hash-Tabelle – dem Anwender bereits fertig aufgebaut geliefert werden kann.

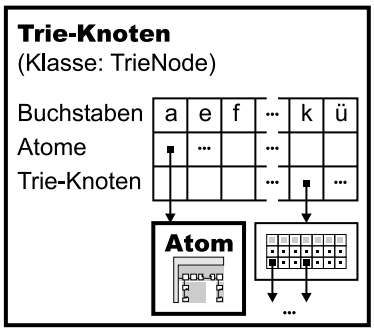

Abbildung 3.11: Aufbau eines Trie-Knotens für die Linked-Trie-Ebenen

Auch wenn hier ein Array und nicht etwa eine lineare Liste verwendet wird, so handelt es sich trotzdem aus logischer Sicht um Knoten eines Linked Tries, da keinem Buchstaben ein fester Platz im Array zugewiesen ist, auf den man direkt über seinen Index (mit konstantem Aufwand) zugreifen könnte. Das hat zur

Folge, dass beim Durchlaufen des Tries innerhalb dieses Arrays der gewünschte Buchstabe erst gesucht werden muss.

Es sind nun verschiedene Varianten vorstellbar, die Buchstaben in diesem dynamischen Array anzuordnen und dann möglichst effizient nach diesen zu suchen, was zur Implementierung von drei unterschiedlichen Arten von Trie-Knoten führte (wobei innerhalb eines Tries immer nur eine Variante zur Anwendung kommt):

1. Die einzelnen Buchstaben werden in alphabetischer Ordnung (d.h. dem Unicode-Standard folgend) sortiert in das Array eingefügt. Durchsucht wird es linear vom Anfang bis zum Ende, wobei unter Umständen die Suche frühzeitig abgebrochen werden kann, nämlich wenn man im Array auf einen Buchstaben stößt, der bereits „größer“ ist als der gesuchte (linearer Aufwand).

2. Die Buchstaben werden ebenfalls in alphabetischer Ordnung sortiert gespeichert, gesucht wird aber mittels binärer Suche (logarithmischer Aufwand).

3. Die Buchstaben werden nach ihrer Häufigkeit in deutschen Wörtern sortiert, wobei Buchstaben mit hoher Auftrittswahrscheinlichkeit wie e oder n im Array nach vorne gereiht werden. Dabei wird allerdings keine Rücksicht auf den Kontext innerhalb des Wortes genommen, sondern es wird nur die allgemeine Wahrscheinlichkeit berücksichtigt (e ist der häufigste Buchstabe im Deutschen, allerdings ist die Wahrscheinlichkeit, dass nach einem e derselbe Buchstabe gleich wieder folgt, eher gering). Durchsucht wird das Array wieder linear vom Anfang bis zum Ende, ein vorzeitiger Abbruch bei erfolgloser Suche ist hier aber nicht effizient möglich (linearer Aufwand).

Werden nun aus diesem hybriden Trie alle Präfixatome eines Wortteiles angefordert, so läuft die Suche in zwei Phasen ab: Zunächst sind im Normalfall (Wortteil besteht aus mehr als einem Buchstaben) zwei Zugriffe in das Array der Indexed-Trie-Komponente notwendig, und zwar für den ersten Buchstaben alleine (Buchstabe plus Leerzeichen) und danach für die Kombination der ersten beiden Buchstaben des Wortteiles. Findet man dabei Verweise auf Zusatzinformationen zu einem Atom, so können diese bereits zwischengespeichert werden. Besteht der Wortteil aus mindestens drei Buchstaben, so muss die Suche nun in der Linked-Trie-Komponente fortgesetzt werden, wobei alle am Pfad liegenden Atome aufgesammelt werden. Ist man am Ende des Wortteiles angelangt bzw. befindet man sich bei einem Trie-Knoten in einer Sackgasse (keine Referenz auf einen Nachfolgeknoten vorhanden), so kann die Suche beendet werden und die gefundenen Atome können dem Wortzerlegungsalgorithmus zur weiteren Verarbeitung übergeben werden.

Das Laufzeitverhalten innerhalb der Linked-Trie-Komponente (der bestimmende Faktor) wird entscheidend von der Anzahl der Buchstaben beeinflusst, die ein Linked-Trie-Knoten zu verwalten bzw. bei einer Anfrage zu durchsuchen hat. Tabelle 3.2 bietet hierzu eine Übersicht (Stand Juli 2003, 8.376 eingefügte Atome, insgesamt 13.001 Linked-Trie-Knoten).

Anzahl Buchstaben	*Anzahl Trie-Knoten*		*Anzahl Buchstaben*	*Anzahl Trie-Knoten*	
1	10.362	(79.70%)	11	21	(0.16%)
2	1.495	(11.50%)	12	15	(0.12%)
3	428	(3.29%)	13	11	(0.08%)
4	199	(1.53%)	14	13	(0.10%)
5	120	(0.92%)	15	5	(0.04%)
6	89	(0.68%)	16	6	(0.05%)
7	84	(0.65%)	17	7	(0.05%)
8	60	(0.46%)	18	3	(0.02%)
9	45	(0.35%)	19	1	(0.01%)
10	35	(0.27%)	20	2	(0.02%)

Tabelle 3.2: Anzahl der Trie-Knoten, die eine bestimmte Menge an Buchstaben innerhalb der Linked-Trie-Komponente des hybriden Tries verwalten müssen

Übrigens befinden sich nicht unbedingt alle der gut gefüllten Trie-Knoten in der dritten Trie-Ebene (der dritte Buchstabe eines Atoms, also direkt nach der Index-Trie-Komponente), wie man vielleicht vermuten mag. So gibt es Knoten mit mehr als 10 Buchstaben nach *gra*, *schwa* oder *tra*. Eine Ausdehnung der Indexed-Trie-Komponente auf die ersten drei Buchstaben wäre daher nur bedingt geeignet, die Anzahl der Trie-Knoten mit hohem Füllgrad zu reduzieren, würde dafür aber den Speicherplatzverbrauch deutlich in die Höhe schrauben.

Zusammenfassend kann festgehalten werden, dass der hybride Trie für SiSiSi die Vorteile von Indexed (schneller, direkter Zugriff auf einzelne Buchstaben innerhalb eines Trie-Knotens) und Linked Trie (geringer Speicherplatzverbrauch) verbindet, die Nachteile dieser beiden Varianten des Tries aber weitestgehend vermeidet: Der enorme Speicherplatzverbrauch des Indexed Tries wird durch die Linked-Trie-Komponente abgefangen, und der zusätzliche Aufwand bei der Suche nach dem entsprechenden Buchstaben innerhalb eines Linked-Trie-Knotens hält sich in der Umgebung der Blätter in Grenzen, da hier nur noch Daten zu sehr wenigen, meist sogar nur noch einem einzigen Buchstaben gespeichert sind.

Eine weitere Variante eines Tries für die Anforderungen von SiSiSi, die allerdings nicht im Zuge dieser Diplomarbeit realisiert wurde, wäre ein sich selbst organisierender, hybrider Trie: Die einzelnen Knoten beginnen als Knoten eines Linked Tries. Wird aber beim Einfügen neuer Atome ein frei definierbarer Füllgrad für den Knoten überschritten, wandelt sich dieser selbstständig in einen Indexed-Trie-Knoten um. Mit dem Füllgrad als Parameter hätte man die Kontrolle darüber, ob man mehr Wert auf Geschwindigkeit oder eher auf schonenderen Speicherplatzverbrauch legt, Anforderungen, die auch vom jeweiligen Anwendungsfall abhängen können.

3.1.6 Vererbungshierarchie der implementierten Atomtabellen

Als Zusammenfassung des Abschnittes über die verschiedenen Atomtabellen soll noch eine Übersicht über die Vererbungshierarchie der in Java-SiSiSi für diesen Zweck implementierten Datenstrukturen gegeben werden.

Ziel der Entwicklung war es, die Atomtabellen sauber zu kapseln, um sie vollständig von anderen Komponenten von SiSiSi zu entkoppeln und die einzelnen Implementierungen auch noch zur Laufzeit des Programms gegeneinander austauschen zu können.

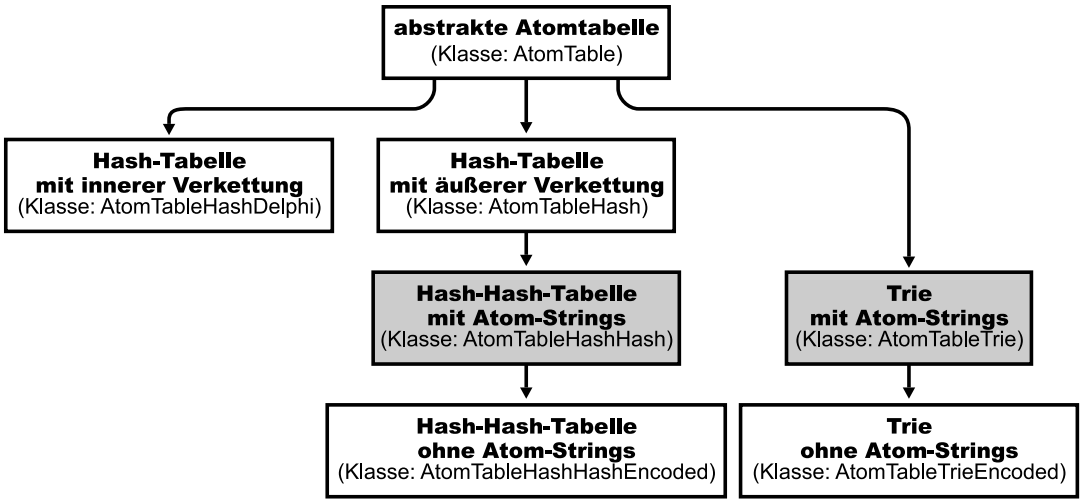

Abbildung 3.12: Vererbungshierarchie der in Java-SiSiSi implementierten Atomtabellen

Die Hash-Tabelle mit innerer Verkettung (3.1.2), wie sie Delphi-SiSiSi verwendet, wurde lediglich zu Testzwecken implementiert, um einen möglichst fairen Vergleich mit den neuen Datenstrukturen durchführen zu können. Ebenfalls programmiert wurden die in diesem Kapitel vorgestellte Hash-Tabelle mit äußerer Verkettung (3.1.3), die Hash-Hash-Tabelle (3.1.4) und der hybride Trie (3.1.5), die beiden Letztgenannten jeweils ohne explizite Speicherung des Atom-Strings.

In der Abbildung grau hinterlegt sind zwei Varianten, deren Einsatzgebiet nicht die Silbentrennung in SiSiSi, sondern die Volltextsuche ist. Hierfür werden die kompletten Zerlegungsdaten inklusive der Atom-Strings benötigt. Zwar können diese während der Wortzerlegung generiert werden (was auch geschieht, wenn eine Hash-Hash-Tabelle oder ein Trie ohne Atom-Strings für die Volltextsuche verwendet wird), allerdings sind dafür zusätzliche Stringoperationen erforderlich, die den Algorithmus verlangsamen. Für den internen Gebrauch am Institut (wo eine Geheimhaltung bzw. Verschleierung der Atom-Strings nicht notwendig ist) oder aber für verschiedene Tests kann so auf diese etwas schnelleren Implementierungen der Atomtabelle zurückgegriffen werden.

3.2 Der Regelautomat

Aufgabe des Regelautomaten während der Wortanalyse ist es, sinnlose Wortzerlegungen, d.h. unbrauchbare Aneinanderreihungen von Atomen, durch Berücksichtigung der deutschen Wortbildungsgrammatik weitestgehend zu verhindern. So darf z.B. keine Endung unmittelbar auf eine Vorsilbe folgen, d.h. ohne einen Wortstamm dazwischen.

Der aktuelle Stand der Zerlegung entspricht dabei einem Zustand im Regelautomaten, z.B. *„nach einer allgemeinen Vorsilbe“* oder *„nach Substantivstamm mit Deklinationstyp S2“*. Begonnen wird in einem speziellen Startzustand. Details zu den einzelnen Zuständen finden sich in [Ste95] und [Kod01]. Ob ein gefundenes Atom im aktuellen Zustand die bisherige Zerlegung fortsetzen kann, das hängt von den Aufgaben ab, die es bei der Wortbildung übernehmen kann, d.h. von den zu einem Atom vermerkten Atomklassen. Der Automat speichert also Regeln der Form

Ausgangszustand → Atomklasse → Zielzustand/-zustände

wobei gilt, dass wenn für ein konkretes Paar *Ausgangszustand* (aktueller Stand der Zerlegung) und *Atomklasse* keine Regel definiert ist, das Atom zumindest mit dieser Atomklasse im momentanen Zustand nicht verwendet werden darf.

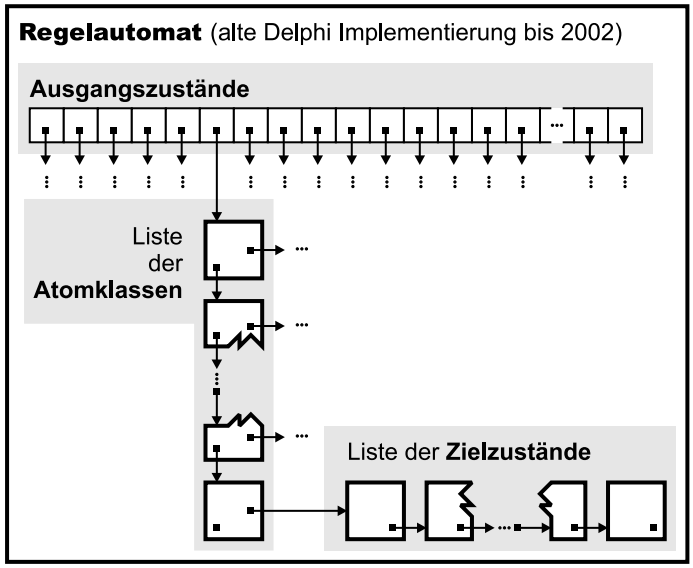

Abbildung 3.13: Implementierung des Regelautomaten in Delphi bis 2002

Wie schon bei der Atomtabelle war die treibende Kraft hinter der lange Zeit in SiSiSi für den Regelautomaten eingesetzten Datenstruktur ein so geringer Speicherplatzverbrauch wie möglich. Die Ausgangszustände wurden in einem dynamischen Array verwaltet. Zu jedem Ausgangszustand wurden in einer linearen Liste die Atomklassen gespeichert, mit deren Hilfe dieser Zustand verlassen werden

kann. Bei jeder dieser Atomklassen wurden, ebenfalls wieder in linearen Listen, alle möglichen Zielzustände für die entsprechende Regel vermerkt.

Um trotzdem möglichst effizient mit dem Regelautomaten arbeiten zu können, wurden alle diese linearen Listen sortiert, auch die Listen der zu Atomen in der Atomtabelle gespeicherten Atomklassen. Mit dieser Maßnahme wurde erreicht, dass der Aufwand während der Wortzerlegung für die Abarbeitung aller Atomklassen eines Atoms nur linear in der Länge der zu einem Ausgangszustand gespeicherten linearen Liste von Atomklassen war. Ohne Sortierung wäre dieser Aufwand quadratisch ausgefallen.

Für Java-SiSiSi bzw. auch für die aktuelle Version von Delphi-SiSiSi wurde diese Form des Regelautomaten durch eine Übergangsmatrix ersetzt. Damit ist für jedes Paar *Ausgangszustand* und *Atomklasse* ein direkter Zugriff auf die Liste der Zielzustände möglich, wenn eine solche Regel definiert ist. Dabei kann eine lineare Liste als Datenstruktur zur Verwaltung der Zielzustände beibehalten werden, da diese bei Anwendung einer Regel sowieso alle der Reihe nach abgearbeitet werden müssen.

Der direkte Zugriff bedeutet eine deutliche Reduktion des Aufwandes während der Wortanalyse, da dieser nun nur noch linear von der Anzahl zu einem Atom gespeicherter Atomklassen abhängt, und das sind durchschnittlich nur rund 1,61 Klassen pro Atom (Maximum: 17). Hingegen besitzt im Vergleich dazu ein Ausgangszustand im Regelautomaten im Durchschnitt 40,43 Atomklassen als Regelübergänge (Maximum: 185).

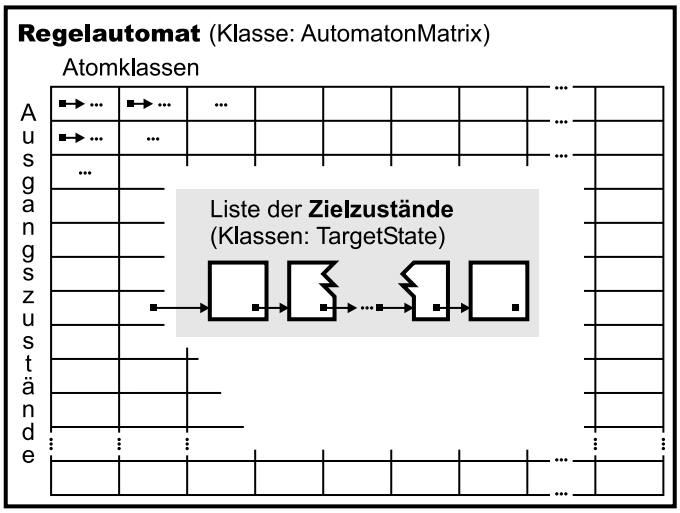

Abbildung 3.14: Als Übergangsmatrix implementierter Regelautomat

Die Matrix ist eher schwach besetzt: Aktuell (Stand Oktober 2003) kennt SiSiSi 107 Regelautomatenzustände und weist Atomen eine Auswahl von insgesamt 245 Atomklassen zu. Die Übergangsmatrix besitzt demnach eine Dimension von 107 x 245, bei 4.326 gespeicherten Regeln entspricht das einem Füllgrad von lediglich knapp 16,5%. Doch der Speicherplatzverbrauch inklusive der darin

gespeicherten Regeln beläuft sich auf weit unter einem halben Megabyte. Er bewegt sich damit in einem Bereich, der sowohl für den Hauptspeicherausbau eines heutigen Computers absolut kein Problem mehr darstellt als auch gerne für den schnellen, direkten Zugriff in Kauf genommen wird.

Neben den Regeln für die deutsche Wortbildung speichert das Objekt für den Automaten noch zusätzliche Informationen zu Zuständen und Atomklassen. Dazu zählen u.a., ob es sich bei einem Zustand um einen Endzustand handelt (Pseudocode des Wortzerlegungsalgorithmus, Abschnitt 2.4, Zeile 2), ob sich die Wortanalyse in einem Zustand nach einem unvollständigen Stamm befindet oder ob es sich bei einem Atom mit entsprechender Atomklasse um ein Suffix handelt (Pseudocode, 2.4, Zeilen 14 bis 16).

In Java-SiSiSi neu hinzugekommen ist eine Markierung von so genannten Sackgassenzuständen, d.h. von Zuständen, aus denen keine Regel mehr über eine Atomklasse in irgendeinen Zielzustand herausführt, z.B. nach bestimmten Deklinationsendungen. Prüft man während der Wortzerlegung (wenn man noch nicht am Wortende angelangt ist) auf diese Zustände, so können sinnlose Rekursionsschritte inklusiver aller damit verbundenen Operationen frühzeitig vermieden werden.

3.3 Ergebnisse: Vergleich der verschiedenen Atomtabellenimplementationen

Eines der erklärten Ziele von Java-SiSiSi war die Effizienzsteigerung gegenüber der bisherigen Implementierung. Diese Vorgabe konnte unter anderem mit Hilfe neuer Datenstrukturen auch erreicht werden. Die Ergebnisse werden nun im Detail vorgestellt.

3.3.1 Speicherplatzverbrauch

Der tatsächliche Speicherplatzverbrauch der einzelnen Datenstrukturen im Hauptspeicher lässt sich nur schwer exakt feststellen. Die zu diesem Zweck von Java zur Verfügung gestellten Methoden `freeMemory()` und `totalMemory()` der Klasse `Runtime` liefern interessanterweise keine deterministischen Werte (auch bei expliziter Ausführung des Java Garbage Collectors). Tabelle 3.3 listet daher nur ungefähre, über mehrere Aufrufe gemittelte Werte auf, die bei jedem Start des Programms im Bereich von wenigen Prozent schwanken können.

Doch auch wenn es sich nicht um ganz genaue Daten handelt, die generellen Größenordnungen sind erkennbar. Der hybride Trie ist trotz der größtenteils verwendeten Linked-Trie-Knoten die Datenstruktur mit dem größten Speicherplatzverbrauch, die verschiedenen Varianten der Hash-Tabellen sind hier deutlich bescheidener. Die Hash-Tabelle mit äußerer Verkettung benötigt, bedingt durch die Verwendung linearer Listen zur Verwaltung mehrerer Atome pro Hashtabellenindex, etwas mehr Hauptspeicher als die Hash-Tabelle mit innerer Verkettung,

Datenstruktur	*Speicherplatz*
Hash-Tabelle mit innerer Verkettung	1.626 KB
Hash-Tabelle mit äußerer Verkettung	1.738 KB
Hash-Hash-Tabelle ohne Atom-Strings	1.218 KB
Hash-Hash-Tabelle mit Atom-Strings	1.824 KB
Trie ohne Atom-Strings	2.261 KB
Trie mit Atom-Strings	2.850 KB

Tabelle 3.3: Speicherplatzverbrauch der Datenstrukturen im Hauptspeicher ((Hash-)Hash-Tabellen mit einem Füllgrad von 70%)

wie sie bisher in Delphi-SiSiSi verwendet wurde. Am sparsamsten geht die Hash-Hash-Tabelle mit dem vorhandenen Speicherplatz um, da sie darauf verzichten kann, die doch recht voluminösen Atom-Strings zu speichern, und diese durch die verdichteten Schlüssel, jeweils einen Integer-Wert, ersetzt.

Auch wenn dieser Vergleich beim Hauptspeicherausbau heutiger Computer fast wie Haarspalterei wirkt, so gibt es durchaus Argumente für einen möglichst geringen Speicherplatzverbrauch der verwendeten Datenstrukturen. So können sie schneller eingelesen werden, was den Programmstart von SiSiSi beschleunigt, und es passt auch mehr relevante Information in die verschiedenen Caches moderner Rechnerarchitekturen. Diese Aspekte wurden aber im Rahmen dieser Diplomarbeit nicht detaillierter untersucht.

3.3.2 Laufzeitvergleich

Für den Laufzeitvergleich wurden zwei Komponenten des innersten Kerns von SiSiSi getrennt betrachtet: die Atomtabelle in ihren verschiedenen Implementierungen (Suche nach Präfixatomen) und der übrige Teil des Wortzerlegungsalgorithmus inklusive des Regelautomaten und der Dudentrennung.

Ausgangspunkt ist die bis vor kurzem in Delphi-SiSiSi eingesetzte Hash-Tabelle mit innerer Verkettung und dem enorm hohen Füllgrad von 90% (*Hash-Tabelle miV 90% [Zno]*; *[Zno]: Zugriff nicht optimiert*). Dabei ist die Atomtabelle der die Laufzeit klar dominierende Bestandteil mit einem Anteil von rund 55%. Zusätzlich wurde der Zugriff in die Hash-Tabelle so simuliert, wie er in Delphi-SiSiSi auch angewendet wird, was allerdings Mehrfachberechnungen von Hash-Werten[1] beinhaltet, wie sie in Java-SiSiSi durch eine anders gewählte Schnittstelle (nicht zuletzt, um die Möglichkeiten des Tries voll ausschöpfen zu können)

[1]Die Hash-Funktion $H(S)$ für den Index in die Hash-Tabelle basiert in SiSiSi auf der Addition einzelner, aus den Buchstaben des zu untersuchenden Wortteiles berechneter Werte. Da Java-SiSiSi alle Präfixatome eines Wortteiles auf einmal anfordert, kann die Atomtabelle bereits berechnete Werte (z.B. den Hash-Wert für den ersten Buchstaben, dann für den zweiten, usw.) innerhalb dieser einen Anfrage wiederverwenden.
In der Delphi-Implementation ist die Schnittstelle zwischen Atomtabelle und Wortzerlegungsalgorithmus anders definiert: Hier zerschneidet der Algorithmus den Wortteil selbst und lässt

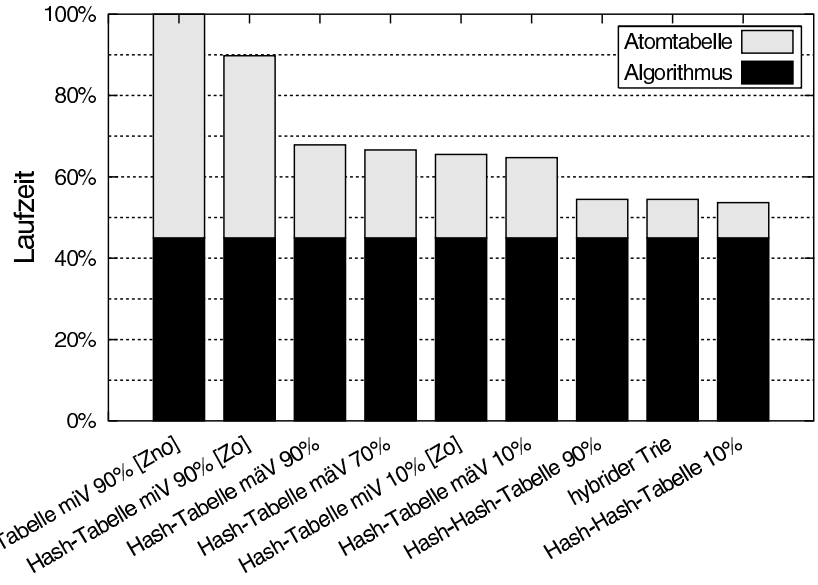

Abbildung 3.15: Laufzeitvergleich Atomtabelle (unterschiedliche Implementationen) und restlicher Algorithmus bei der Silbentrennung von SiSiSi

zwischen Atomtabelle und Algorithmus nicht notwendig sind. Alleine die Vermeidung dieser Mehrfachberechnungen reduziert den Anteil der Atomtabelle an der Gesamtlaufzeit deutlich (*Hash-Tabelle miV 90% [Zo]: Zugriff optimiert*).

Eine – vielleicht unerwartet – große Differenz kann man im Laufzeitverhalten zwischen Hash- und Hash-Hash-Tabelle (jeweils mit äußerer Verkettung: $mäV$) erkennen. Zur Erinnerung: Der einzige Unterschied zwischen diesen beiden Varianten der Atomtabelle besteht darin, dass die Hash-Hash-Tabelle statt des Atom-Strings selbst nur einen verdichteten Schlüssel (in der konkreten Implementierung eine Integer-Zahl), basierend auf den einzelnen Buchstaben des Atom-Strings, speichert. Für die Anfrage nach Präfix-Atomen bedeutet das, dass die Hash-Hash-Tabelle den zu untersuchenden Wortteil zunächst verdichten (die Hash-Funktion $V(S)$ auswerten) und diesen Wert in der Atomtabelle suchen muss, wohingegen die Hash-Tabelle gleich direkt die gespeicherten Atom-Strings mit dem Wortteil vergleichen kann. Trotzdem ist die Hash-Hash-Tabelle beachtlich schneller, was nur den Schluss zulässt, dass der Vergleich zweier Strings in Java (`String.compareTo()`) eine überaus zeitraubende Methode ist, die es nach Möglichkeit zu vermeiden gilt.

Dass die Hash-Hash-Tabelle schneller ist als die Hash-Tabelle mit äußerer Verkettung mag man vielleicht noch erwartet haben, aber dass sie mit der Lauf-

die Atomtabelle für jedes Stück einzeln überprüfen, ob es sich dabei um ein Atom handelt. Bei einem Wortteil der Länge n wird daher der für die Hash-Funktion $H(S)$ benötigte Wert des ersten Buchstaben n-mal berechnet, der Wert für den zweiten Buchstaben $(n-1)$-mal usw., was nur mit einigem Aufwand (z.B. Umstellung der Schnittstelle) unterbunden werden könnte (z.B. muss für Client-Server Applikation berücksichtigt werden, dass mehrere Clients auf eine Atomtabelle zugreifen, damit kommen Caches in der Atomtabelle selbst nicht in Frage).

zeit des hybriden Tries mithalten kann, war doch überraschend. Denn wo die Hash-Hash-Tabelle bei der Suche nach allen Präfixatomen eines Wortteiles alle Möglichkeiten durchprobieren muss, kann der Trie im Normalfall die Suche frühzeitig abbrechen, und trotzdem ist die Hash-Hash-Tabelle ebenbürtig.

Es mag vielleicht auch verwundern, dass der hybride Trie in der Grafik mit nur einem einzigen Wert vertreten ist. In Abschnitt 3.1.5 wurden noch drei unterschiedliche Implementierungen der Linked-Trie-Knoten vorgestellt. Doch die Laufzeitunterschiede dieser drei Varianten bewegten sich lediglich innerhalb der Messungenauigkeit. Allerdings war durchaus eine Tendenz erkennbar, und zwar, dass die ganz normalen Linked-Trie-Knoten (Buchstaben in alphabetischer Ordnung gespeichert und linear durchsucht) meist am effizientesten waren. Das ist wohl damit zu erklären, dass fast 80% der Trie-Knoten nur Daten zu einem einzigen Buchstaben verwalten, wodurch die anderen beiden Knoten-Varianten ihre vermeintlichen Stärken bei der Suche nicht ausspielen können.

Schön zu sehen ist auch die relative Unempfindlichkeit beim Füllgrad der Hash-Tabellen mit äußerer Verkettung gegenüber der mit innerer Verkettung. Wichtig ist dies vor allem für die Hash-Hash-Tabelle ohne Atom-Strings, denn diese kann zur Laufzeit nicht vergrößert werden, auch wenn neue Atome aufgenommen werden. Grund dafür ist die Berechnung des Index $H(S)$, die sowohl vom Atom-String als auch von der Größe der Hash-Hash-Tabelle abhängig ist. Dieser Index müsste bei einer Erweiterung der Datenstruktur auch für die bereits gespeicherten Atome neu berechnet werden. Dies ist allerdings nicht möglich, da die Atom-Strings selbst nicht gespeichert sind, sondern nur ihre verdichteten Schlüssel, aus denen keinerlei Information für die Berechnung von $H(S)$ gewonnen werden kann.

Der Anteil der Atomtabelle an der gesamten Laufzeit von SiSiSi bei der Silbentrennung konnte, wie die Grafik eindrucksvoll beweist, deutlich reduziert werden. Allerdings erkennt man auch, dass eine weitere Beschleunigung der Datenstrukturen für die Atomtabelle an der Gesamtlaufzeit des Programms nicht mehr sehr viel ändern wird. Um diese weiter zu drücken, müsste nun der Algorithmus selbst einige Veränderungen erfahren, doch zur Zeit ist auf diesem Gebiet, wenn überhaupt möglich, keine Verbesserung in Sicht.

Nach dem relativen Laufzeitvergleich der einzelnen Atomtabellen zum Abschluss noch einige konkrete Zahlen, die auf einem System mit Pentium®4 Prozessor mit 1,8 GHz, 512 MB Hauptspeicher, Linux 2.4.19 ohne grafische Benutzeroberfläche und Sun Java™ SDK 1.4.2 ermittelt wurden. Für die Messung wurden 209.544 unterschiedliche, deutsche Wörter aus einer Textdatei eingelesen, für SiSiSi aufbereitet (Ersetzung von Großbuchstaben durch ihre entsprechenden Kleinbuchstaben, Eliminierung von Akzenten, usw.), einer Silbentrennung unterzogen und mit allen gefundenen, potenziellen Trennstellen wieder in eine Textdatei geschrieben.

Sowohl die Hash-Hash-Tabelle (Füllgrad 70%) als auch der hybride Trie führten die Silbentrennung dieser Wörter innerhalb von 50 Sekunden durch, was einer Bearbeitungsgeschwindigkeit von rund 4.190 Wörtern pro Sekunde entspricht.

Mit einer Hash-Tabelle mit äußerer Verkettung als Atomtabelle arbeitete SiSiSi bereits länger als eine Minute, nämlich 63 Sekunden (3.081 Wörter/s), an dieser Aufgabe. Zum Vergleich benötigte das Programm mit der nicht optimierten Variante der Hash-Tabelle mit innerer Verkettung (Delphi-SiSiSi) 1 Minute und 34 Sekunden (2.229 Wörter/s).

Kapitel 4

Der Vortrenner

Der Vortrenner ist eine auf dem Wortanalysealgorithmus von SiSiSi aufbauende Applikation. Mit ihr können optionale Trennzeichen in einige der wichtigsten Dateiformate (darunter das Rich Text Format RTF, HyperText Markup Language HTML oder auch LaTeX) eingefügt werden. D.h. es werden alle von SiSiSi gefundenen, potenziellen Trennstellen in ein Dokument eingetragen, das entsprechende Textsystem verwendet diese dann bei Bedarf am Ende einer Zeile automatisch.

Eine direkte Integration von SiSiSi in vorhandene Textverarbeitungssysteme wie z.B. Microsoft® Word oder LaTeX gestaltet sich aus mehreren Gründen schwierig. Einmal abgesehen von der Tatsache, dass die Ressourcen des Instituts bei weitem nicht ausreichen, SiSiSi ständig an zum Teil proprietäre Software anzupassen, sind auch die Schnittstellen, die diese Programme bieten, für SiSiSi nicht ausreichend, will man wirklich alle zur Verfügung gestellten Möglichkeiten ausnutzen. Für die richtige Behandlung von Mehrdeutigkeiten, wie sie in der deutschen Sprache durch Wortzusammensetzungen entstehen können, ist ein gewisses Maß an Interaktion mit dem Benutzer unumgänglich (z.B. die schon öfter zitierte *Wach-stube* bzw. *Wachs-tube*). Und auch eine unterschiedliche Bewertung der verschiedenen Trennstellen (z.B. Haupttrennstelle an einer Wortfuge oder Nebentrennstelle, die womöglich den Sinn des zu trennenden Wortes zerstört) bietet keines dieser Systeme.

Weiters kann der Vortrenner überall dort verwendet werden, wo endgültige, sich nicht mehr verändernde Dokumente mit hoher Qualität in unterschiedlicher Breite und/oder Höhe ausgegeben werden sollen, auf dem Anzeigegerät bzw. in der Anzeigesoftware SiSiSi aber nicht eingesetzt werden kann. Beispiele hiefür sind u.a. Texte für Taschencomputer (Palmtops), die nicht die entsprechende Rechenleistung und Speicherkapazität mitbringen, aber auch Webseiten, wo niemand verlangen kann, dass ein Webbrowser für jede nur erdenkliche Sprache einen eigenen Silbentrennalgorithmus implementiert hat, um HTML Dokumente in ansprechender Form zu präsentieren. Der Speicherplatz der mit optionalen Trennstellen erweiterten Dokumente wächst nur unwesentlich, die Lesbarkeit der Texte kann aber mitunter deutlich gesteigert werden (siehe auch [KS03]).

4.1 Der Java-Vortrenner Si3Hyph

Im Folgenden wird der im Rahmen dieser Diplomarbeit entstandene Java-Vortrenner *Si3Hyph* kurz vorgestellt, eine Adaptierung des bereits existierenden Delphi-Vortrenners für Windows®.

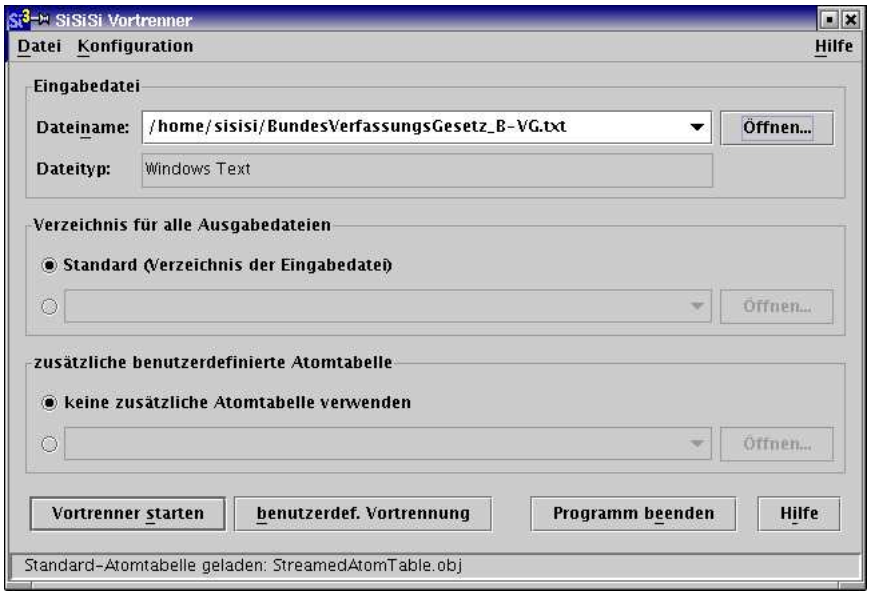

Abbildung 4.1: Startfenster von *Si3Hyph*

Begrüßt wird der Benutzer mit dem Startfenster, in welchem die zu trennende Eingabedatei wie auch eine optional zu verwendende Benutzer-Atomtabelle gewählt werden können.

Während der Vortrennung werden mehrere Dateien erzeugt: das Dokument mit den zusätzlich eingefügten optionalen Trennstellen (das Originaldokument wird nicht verändert), eine Logdatei mit selbst wählbarem Inhalt (z.B. alle gefundenen, unbekannten Wörter), eine Datei mit dokumentspezifischen Trenninformationen und gegebenenfalls eine Datei, die vorgenommene Korrekturen am Originaltext beinhaltet. Möchte man diese Fülle an Dateien nicht direkt zu der Eingabedatei speichern, so kann ein eigenes Verzeichnis für die Ausgaben gewählt werden.

Für die Vortrennung selbst existieren zwei Betriebsarten:

- Die automatische Vortrennung ist der schnelle Weg. Es werden alle bekannten Wörter gemäß den zuvor getroffenen Einstellungen bezüglich Trennprofil (welche Trennstellen sollen neben Haupttrennstellen noch verwendet werden) und Trennvariante (Trennung der Wörter nach den Regeln ihrer Herkunftssprache oder nach Silben; siehe dazu auch Abbildung 4.3) mit optionalen Trennstellen versehen.

Abbildung 4.2: Hauptfenster des Vortrenners

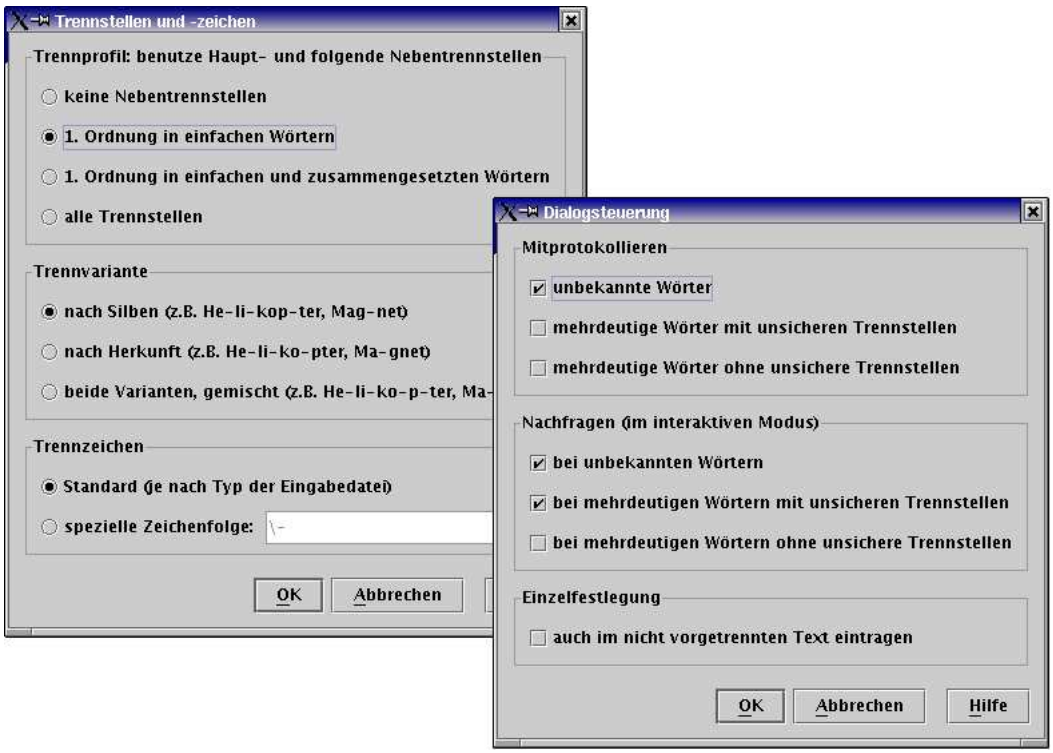

Abbildung 4.3: Optionen für die Vortrennung

- Die deutlich interessantere Variante ist allerdings die interaktive Vortrennung mit erweiterten Eingriffsmöglichkeiten für den Benutzer. In diesem Modus wird – wenn gewünscht (siehe Abbildung 4.3) – die Verarbeitung des Dokuments bei unbekannten und/oder mehrdeutigen Wörtern angehalten. Bei mehreren Trennvarianten aufgrund von unterschiedlichen vom Wortanalysealgorithmus gefundenen Zerlegungen eines Wortes kann die gewünschte Trennung entweder direkt ausgewählt oder auch selbst vorgegeben werden, auf Wunsch nur für dieses eine Vorkommen des Wortes oder für das gesamte noch folgende Dokument. Weiters können etwaige Tippfehler korrigiert und auch neue Stämme, z.B. Eigennamen, in die Benutzer-Atomtabelle aufgenommen werden.

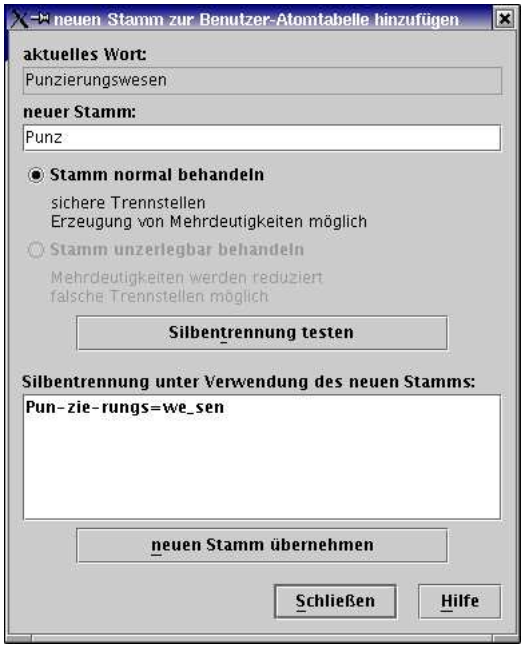

Abbildung 4.4: Dialog, um einen neuen Stamm in die Benutzer-Atomtabelle aufzunehmen

Das Hinzufügen eines neuen Stammes zu einer Benutzer-Atomtabelle kann mitunter gefährlich sein, der entsprechende Dialog ist daher streng strukturiert. Zunächst kann vom Benutzer ein Stamm angegeben werden, der eine gewisse Mindestlänge haben und Bestandteil des gerade zu trennenden Wortes sein muss. Im Zuge dieser Eingabe können auch eine Haupt- oder maximal zwei Nebentrennstellen (Ausnahmetrennstellen, wenn die Dudentrennung falsche Ergebnisse erzielt) spezifiziert werden. Danach muss die Trennung durch den Wortanalysealgorithmus erst mit Hilfe des neuen Stammes auf Basis des aktuellen Wortes getestet werden, bevor dieser endgültig, nach vielleicht notwendigen Korrekturen, in die Benutzer-Atomtabelle aufgenommen werden kann.

Standardmäßig wird ein Stamm als normal, d.h. zerlegbar, betrachtet und auch so in die Atomtabelle aufgenommen. Dabei kann es zu keinen Fehlern bei der

Silbentrennung kommen, es werden aber vielleicht unnötig viele Mehrdeutigkeiten, d.h. unterschiedliche Zerlegungen, erzeugt. Besteht die absolute Notwendigkeit, diese Mehrdeutigkeiten zu unterdrücken, dann besteht die Möglichkeit, einen Stamm auch als unzerlegbar in die Benutzer-Atomtabelle einzutragen. Dies birgt allerdings die Gefahr, bisher gefundene und gültige Zerlegungen zu verhindern, und kann daher im Extremfall zu falschen Resultaten führen. Dieses Problem ist allerdings nicht mit den Bemerkungen zu Zeile 13 des Wortzerlegungsalgorithmus zu verwechseln, wo es um gleiche Atome (z.B. *abc*) aus unterschiedlichen Atomtabellen ging. Hier wird durch die Eintragung des Stammes *abcde* als unzerlegbar die Bildung eines Wortes aus den Atomen *abc* und *de* verhindert (siehe auch Erläuterungen zu Zeile 15 des Algorithmus).

Kapitel 5

Die Schnittstelle der Java-SiSiSi Bibliothek

Die Schnittstelle zur SiSiSi-Bibliothek wurde dahin gehend optimiert, mit möglichst wenigen Zeilen Code Zugriff auf alle zur Verfügung gestellten Funktionen zu erhalten und trotzdem möglichst flexibel zu bleiben.

Dieses Kapitel gibt einen sehr kurz gefassten Überblick über die von der Java-SiSiSi-Bibliothek zur Verfügung gestellten Klassen und Methoden. Nähere Details zur Schnittstelle inklusive Verwendungsbeispiele finden sich bei der Dokumentation der Bibliothek.

5.1 Einbindung der Bibliothek und benötigte Objekte

Ausgeliefert wird die Java-SiSiSi-Bibliothek in einem digital signierten Java-Archiv (JAR). Dieses Archiv enthält alle benötigten Klassen und die Atomtabelle inklusive des Regelautomaten zur Wortbildung.

Innerhalb eines Programms, das die Bibliothek verwenden möchte, muss diese zunächst mit einer `import`-Anweisung eingebunden werden:

```
import sisisi.*;
```

Damit der Java-Compiler und -Interpreter die SiSiSi-Klassen innerhalb des Java-Archivs auch findet, muss der `CLASSPATH` (Pfad, in dem Java nach Klassen bzw. Bibliotheken sucht) richtig gesetzt sein. Das kann entweder über die entsprechende Umgebungsvariable oder – wie im folgenden Beispiel – mit Hilfe das Parameters `-classpath` geschehen (unter Microsoft® Windows® muss der Doppelpunkt durch einen Strichpunkt ersetzt werden):

```
javac -classpath .:sisisi.jar MyProgram.java
```

Für die eigentliche Wortanalyse wird ein Objekt der Klasse `WordAnalyzer` benötigt. In dieser ist der Wortzerlegungsalgorithmus implementiert. Wie bereits bekannt ist, benötigt der Algorithmus für seine Aufgabe aber Zugriff auf eine Atomtabelle und den mit ihr assoziierten Regelautomaten. Diese Daten liegen im Normalfall als binär abgespeicherter Datenstrom im Java-Archiv vor und können mit Hilfe der Klasse `AtomTableSerializer` direkt in eine Referenz des Typs `AtomTable` (die Vorgängerklasse aller implementierten Atomtabellen) eingelesen werden. Weiters ist der Trennautomat für die Dudentrennung in eine eigene Klasse, `Hyphenator`, ausgelagert worden. Diese Variante wurde gewählt, um sie bei Bedarf möglichst leicht anpassen bzw. ersetzen zu können. Dass dies notwendig sein kann, zeigt die letzte Reform der deutschen Rechtschreibung aus dem Jahr 1998, wo die Silbentrennung eine der am stärksten betroffenen Komponenten war.

```
AtomTable table = null;
try {
  table = AtomTableSerializer.readJAR("sisisi.jar", "AtomTable.obj");
}
catch(Exception e) {
  /*
  Fehlerbehandlung: IOException, ClassNotFoundException
  */
}
WordAnalyzer word_analyzer =
    new WordAnalyzer(table, new Hyphenator());
```

Die explizite Deklaration und Initialisierung der Variable `table` vor dem nachfolgenden `try`-Block ist ein Zugeständnis an den Java-Compiler, der sonst bei der Übersetzung des Programms eine Warnung ausgeben würde.

Mit Hilfe der statischen Methode `AtomTableSerializer.readJAR()` wird aus dem Java-Archiv `sisisi.jar` die Datei `AtomTable.obj` gelesen, welche die Atomtabelle zusammen mit dem Regelautomaten enthält. Dabei kann es zu der bei Dateioperationen üblichen `IOException` (Datei nicht gefunden, Datei kann nicht zum Lesen geöffnet werden, ...) bzw. zu einer `ClassNotFoundException` kommen (das ist im Normalfall nicht möglich, sondern nur, wenn die SiSiSi-Bibliothek unvollständig oder beschädigt ist). Mit der Erzeugung eines `WordAnalyzer`-Objekts – der Konstruktor benötigt eine Referenz auf eine gültige Atomtabelle inklusive Regelautomaten und einen Trennautomaten – sind alle Initialisierungen abgeschlossen, um eine Wortzerlegung starten zu können.

Bei JAR handelt es sich um ein komprimiertes Format. Beim Einlesen der Atomtabelle und des Regelautomaten mittels `AtomTableSerializer.readJAR()` werden diese Daten daher intern vorher noch dekomprimiert, was die Startzeit eines Programms durchaus merklich verzögern kann (abhängig von der Prozessorleistung des Systems). Soll diese optimiert werden, so bietet es sich an, die entsprechenden Daten bereits vor dem Programmstart aus dem Archiv zu extrahieren. Mit `AtomTableSerializer.read()` kann dann die Atomtabelle zusammen mit dem Regelautomaten direkt eingelesen werden:

```
table = AtomTableSerializer.read("AtomTable.obj");
```

Liegen der Regelautomat und/oder die Atomtabelle im Klartext vor, so müssen diese Dateien beim Einlesen erst entsprechend interpretiert werden. Diese Aufgabe übernehmen die Klassen `AutomatonParser` bzw. `AtomTableParser` und stellen hierfür jeweils eine statische Methode mit dem Namen `parse()` zur Verfügung. Der erste Parameter enthält dabei das Objekt, in das die eingelesenen Daten eingefügt werden sollen. Damit ist es bei der Atomtabelle möglich, die ihr zugrunde liegende Datenstruktur näher zu spezifizieren (im folgenden Beispiel wird die Atomtabelle in Form eines Tries gespeichert). Der zweite Parameter der Methode `parse()` enthält den Namen der Datei, die interpretiert werden soll.

```
AutomatonMatrix matrix = null;
AtomTable table = null;
try {
  matrix = AutomatonParser.parse(new AutomatonMatrix(),
                                 "Name_Regeldatei");
  table  = AtomTableParser.parse(new AtomTableTrie(matrix),
                                 "Name_Atomdatei");
}
catch(Exception e) {
  /*
  Fehlerbehandlung: IOException, TableSyntaxException
  */
}
```

Zur üblichen `IOException` kommt hier als möglicher und zu behandelnder Fehlerfall eine `TableSyntaxException` hinzu, die in ihrem Fehlertext genauere Angaben zu dem Problem macht, das beim Analysieren der einzulesenden Datei aufgetreten ist (fehlende oder unvollständige Klammerung, die in der Atomtabelle angegebene Atomklasse ist im Regelautomaten nicht deklariert, etc.).

5.2 Wortanalyse und Trennvektor

Sind die benötigten Objekte generiert, steht der Durchführung der Wortanalyse nichts mehr im Wege. Den Trennvektor eines Wortes liefert die Methode `hyphenationVector()` eines `WordAnalyzer`-Objekts. Das zu trennende Wort darf zu diesem Zeitpunkt aber keine Sonderzeichen mehr enthalten und muss in Form von Kleinbuchstaben vorliegen. Ist das nicht bereits durch andere Maßnahmen gewährleistet, so steht für diesen Zweck die Methode `checkInputWord()` zur Verfügung. Dabei werden auch gleich Zeichen ersetzt, die in jedem Fall umzuwandeln sind, z.B. kann ein `é` für die Wortanalyse durch ein normales `e` ersetzt werden. Um den durch `hyphenationVector()` generierten Trennvektor in das ursprüngliche Wort einzuarbeiten, kann auf die Methode `applyHyphenationVector()` zurückgegriffen werden.

Das folgende Beispiel bereitet für ein Wort in der Variable `input` die Wortanalyse vor, generiert einen Trennvektor und schreibt das Ergebnis (das ursprüngliche Wort kombiniert mit dem Trennvektor) in die Variable `output`:

```
try {
  String word   = word_analyzer.checkInputWord(input);
  String vector = word_analyzer.hyphenationVector(word);
  String output = word_analyzer.applyHyphenationVector(input,vector);
}
catch(Exception e) {
  /*
  Fehlerbehandlung: IllegalCharacterException,
                    IllegalHyphenPositionException,
                    UnknownWordException
  */
}
```

Bei der Wortanalyse kann es zu folgenden Fehlern bzw. Ausnahmefällen kommen, die entsprechend behandelt werden müssen:

- `IllegalCharacterException`: Die Methode `checkInputWord()` generiert diesen Fehler, wenn das zu trennende Wort ein nicht behandelbares Sonderzeichen enthält.

- `IllegalHyphenPositionException`: `hyphenationVector()` meldet diesen Fehler, wenn die Methode für ein Wort aufgerufen wird, das mit einem Trennzeichen beginnt oder endet, da eine Trennung solcher Wörter nicht sinnvoll erscheint. Beispielsweise sollte das Wort *Rettungs-* in der Wortgruppe *Rettungs- und Polizeifahrzeuge* nicht der Worttrennung unterzogen werden; in dieser Form ist es nämlich ungültig, erst in der sich aus dem Kontext (der aber von SiSiSi nicht herangezogen wird bzw. werden kann) ergebenden Kombination *Rettungs-fahrzeuge* handelt es sich um ein gültiges deutsches Wort.

- `UnknownWordException`: `hyphenationVector()` hat für das zu trennende Wort keine gültige Zerlegung (und daher auch keinen Trennvektor) gefunden. Das kann mehrere Ursachen haben, u.a. dass das Wort ungültig ist (z.B. ein Tippfehler), dass es sich um einen Eigennamen handelt, dass ein Atom in der Atomtabelle fehlt oder Ähnliches.

Bei der Ein- und Ausgabe versteht bzw. verwendet SiSiSi spezielle Zeichen, die als öffentliche, statische Konstanten der Klasse `Global` definiert sind (siehe Tabelle 5.1). So können z.B. bei der Eingabe mit Hilfe von optionalen Trennstellen Mehrdeutigkeiten wie bei der schon oft zitierten *Wach-stube* bzw. *Wachs-tube* unterdrückt werden. Bei der Ausgabe liefert die SiSiSi-Bibliothek wohl auch mehr Informationen als im Normalfall benötigt werden. Der von `hyphenationVector()` gelieferte Trennvektor kann so relativ einfach den eigenen Wünschen angepasst

werden (z.B. ausschließliche Verwendung von Haupttrennstellen an Wortfugen), bevor dieser mit dem Eingabewort mittels `applyHyphenationVector()` kombiniert wird.

Eingabe	
`HY_INPUT_FORCED`	Zur Schreibweise eines Wortes gehörender Bindestrich.
`HY_INPUT_OPTIONAL`	Vorgegebene, optionale Trennstelle.
Ausgabe	
`HY_LEVEL1`	Haupttrennstelle an einer Wortfuge.
`HY_LEVEL2`	Nebentrennstelle erster Ordnung.
`HY_LEVEL3`	Nebentrennstelle zweiter Ordnung. Diese sollten nach Möglichkeit nicht verwendet werden, da sie zu unschönen (*A-der*) oder den Sinn entstellenden (*Spargel-der*) Trennungen führen können.
`HY_OLD_LEVEL{2|3}`	Nebentrennstellen erster und zweiter Ordnung nach Herkunft.
`HY_NEW_LEVEL{2|3}`	Nebentrennstellen erster und zweiter Ordnung nach Silben.
`HY_NONE`	Kein Trennzeichen an dieser Position.
`HY_AMBIGUOUS`	Mehrdeutigkeit an dieser Stelle.

Tabelle 5.1: In der Klasse `Global` statisch als Konstante definierte Trennzeichen

Zu beachten ist, dass ein Eingabewort, welches Trennzeichen enthält (bzw. enthalten kann), nicht direkt mit dem Trennvektor verknüpft werden kann, da ein Wort nur mit einem gleich langen Trennvektor kombiniert werden darf (eine Sicherungsmaßnahme), das Eingabewort aber gerade um die Anzahl der Trennzeichen länger ist. `WordAnalyzer` stellt zu diesem Zweck die Methode `prepareInputWord()` zur Verfügung, womit sich in einem solchen Fall folgender Aufruf für `applyHyphenationVector()` ergibt:

```
String output = word_analyzer.applyHyphenationVector(
                    word_analyzer.prepareInputWord(input), vector);
```

5.3 Zusatzinformationen zur Wortanalyse

Neben den bereits besprochenen Methoden stellt `WordAnalyzer` noch weitere Schnittstellen zum Wortzerlegungsalgorithmus bereit:

```
public LinkedList hyphenationVectorList(String word) throws
                      IllegalHyphenPositionException,
                      UnknownWordException;
public int        variantCount(String word) throws ...;
```

```
public boolean    hasUnsafeHyphens(String word) throws ...;
public LinkedList decompositionList(String word) throws ...;
```

Die bereits vorgestellte Methode `hyphenationVector()` liefert den kombinierten Trennvektor eines Wortes, `hyphenationVectorList()` hingegen die Liste der einzelnen bei der Wortanalyse gefundenen Trennvarianten, deren Vereinigung zum kombinierten Trennvektor geführt hat. Mit `variantCount()` kann die Anzahl der gefundenen Trennvarianten abgefragt werden (das entspricht einem Aufruf von `hyphenationVectorList().size()`), mit `hasUnsafeHyphens()` kann man abfragen, ob der kombinierte Trennvektor unsichere Trennstellen enthält. Schließlich erhält man mit einem Aufruf von `decompositionList()` die Liste der gefundenen Wortzerlegungen. Diese ist aber *nicht* mit der Liste der gefundenen Trennvarianten zu verwechseln! Denn verschiedene Wortzerlegungen können zu ein und derselben Trennung des Wortes führen. Im Normalfall ist daher die Wortzerlegungsliste länger als die Liste der Trennvarianten, auf keinen Fall aber kürzer. Ein kurzer Blick zurück auf den Pseudocode des Wortzerlegungsalgorithmus (2.4, Algorithmus 1) verdeutlicht dies: Eine neu gefundene Wortzerlegung wird in die entsprechende Liste zwischen den Zeilen 5 und 6 eingefügt, vor der Abfrage, ob es sich um eine neue, bisher noch nicht gespeicherte Trennvariante handelt.

Aus Gründen der Effizienz verwendet die SiSiSi-Bibliothek intern einen Cache, der alle zum letzten Wort berechneten Daten zwischenspeichert, um sie bei einer unmittelbar darauf folgenden Anfrage ohne neuerliche Wortanalyse gleich verfügbar zu haben (wenn man also z.B. zu einem Wort nicht nur den kombinierten Trennvektor, sondern auch die Liste der Trennvarianten wissen möchte). Während des Wortzerlegungsalgorithmus fallen die meisten der für die gerade vorgestellten Methoden benötigten Daten automatisch an; die einzige Ausnahme bildet die Wortzerlegungsliste. Ihre Erzeugung stellt einen doch erheblichen und zeitlich messbaren Mehraufwand im Algorithmus dar, sodass sie nur auf Anfrage generiert wird. Benötigt man also zusätzlich zum kombinierten Trennvektor auch die Liste der Wortzerlegungen, so sollte man – wenn möglich – darauf achten, zuerst die Methode `decompositionList()` aufzurufen und erst danach mit `hyphenationVector()` den kombinierten Trennvektor aus dem dann bereits generierten Cache anzufordern, um unnötige Mehrfachberechnungen zu vermeiden.

5.4 Benutzer-Atomtabellen

Neben der Standard-Atomtabelle (inklusive Regelautomaten), die bei der Erzeugung eines Objekts des Typs `WordAnalyzer` zwingend notwendig ist, bietet SiSiSi auch die Möglichkeit der Verwendung von zusätzlichen Benutzer-Atomtabellen.

Benutzer-Atomtabellen werden in einem anderen Format gespeichert als die normalen Standard-Atomtabellen. Dafür gibt es mehrere Gründe, u.a. müssen deutlich weniger Daten gespeichert werden, da ein Benutzer zu einem selbst eingetragenen Stamm (Atom) nur wenige Informationen selbst spezifizieren kann

bzw. darf. Auch ist einer Benutzer-Atomtabelle kein Regelautomat fest zugeordnet.

Zum Lesen und Schreiben von Benutzer-Atomtabellen steht die Klasse `UserAtomTable` mit folgenden zwei statischen Methoden zur Verfügung (die einzelnen Parameter sollten selbsterklärend sein):

```
public static void read(AtomTable atom_table, String filename)
    throws IOException, ClassNotFoundException;
public static void write(AtomTable atom_table, String filename)
    throws UserAtomTableWriteException, IOException;
```

Verwaltet werden mehrere Atomtabellen über den `WordAnalyzer`. Prinzipiell können beliebig viele Atomtabellen hinzugefügt, ersetzt und wieder entfernt werden, nur die Standard-Atomtabelle muss erhalten bleiben. Jeder Atomtabelle wird ein eindeutiger Index als Identifikationsnummer zugewiesen (die Standard-Atomtabelle hat immer 0). Jedes Atom, das während der Wortanalyse aus einer Atomtabelle genommen wird, wird mit dem entsprechenden Index versehen, um auch im Nachhinein eine genaue Zuordnung vom Atom zur Atomtabelle treffen zu können.

Zum Abschluss noch die wichtigsten Methodenköpfe der Klasse `WordAnalyzer` zur Verwaltung von Atomtabellen (bei `addAtomTable()` wird der zugewiesene Index, bei `setAtomTable()` und `delAtomTable()` die ersetzte bzw. gelöschte Atomtabelle als Funktionswert zurückgeliefert):

```
int       addAtomTable(AtomTable atom_table);
AtomTable delAtomTable(int index);
AtomTable setAtomTable(int index, AtomTable new_atom_table);
AtomTable getAtomTable(int index);
int       getAtomTableIndex(AtomTable atom_table);
```

Literaturverzeichnis

[Bar90] Barth, Wilhelm: *Volltextsuche mit sinnentsprechender Wortzerlegung.* Wirtschaftsinformatik, 32(5):467–471, Oktober 1990.

[BN85] Barth, Wilhelm und Heinrich Nirschl: *Sichere sinnentsprechende Silbentrennung für die deutsche Sprache.* Angewandte Informatik, 4:152–159, 1985.

[dDAKu98] Dudenredaktion: Annette Klosa u.a., Wissenschaftlicher Rat der (Herausgeber): *Duden: Grammatik der deutschen Gegenwartssprache, neue Rechtschreibung, (Duden Band 4).* Bibliographisches Institut, Mannheim, 6. Auflage, 1998.

[Knu98] Knuth, Donald Ervin: *The Art of Computer Programming*, Band 3 – Sorting and Searching. Addison Wesley, Reading, Massachusetts, 2. Auflage, 1998.

[Kod01] Kodydek, Gabriele: *Automatische Wortanalyse für die deutsche Sprache.* Doktorarbeit, Universität Wien, 2001.

[KS03] Kodydek, Gabriele und Martin Schönhacker: *Si3Trenn and Si3Silb: Using the SiSiSi Word Analysis System for Pre-Hyphenation and Syllable Counting in German Documents.* In: Matousek, Vaclav und Pavel Mautner (Herausgeber): *Proceedings of the Sixth International Conference on Text, Speech and Dialogue (TSD 2003)*, Band 2807 der Reihe *Lecture Notes in Artificial Intelligence*, Seiten 66–73, České Budějovice, Czech Republic, September 2003. Springer-Verlag.

[Lia83] Liang, Franklin Mark: *Word Hy-phen-a-tion by Com-put-er.* Doktorarbeit, Department of Computer Science, Stanford University, 1983. Report No. STAN-CS-83-977.

[Ste95] Steiner, Helmut: *Automatische Silbentrennung durch Wortbildungsanalyse.* Doktorarbeit, Institut für Computergraphik, Technische Universität Wien, 1995.